McGraw-Hill Series in Supercomputing and Parallel Processing

Consulting Editor: **Kai Hwang**, *University of Southern California*

Hwang and Briggs: *Computer Architecture and Parallel Processing*
Hwang and DeGroot: *Parallel Processing for Supercomputers and Artificial Intelligence*
Kogge: *The Architecture of Symbolic Computers*
Lakshmivarahan and Dhall: *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*
Quinn: *Designing Efficient Algorithms for Parallel Computers*
Siegel: *Interconnection Networks for Large-Scale Parallel Processing*

THE ARCHITECTURE OF SYMBOLIC COMPUTERS

Peter M. Kogge
IBM Federal Sector Division

McGraw-Hill, Inc.

New York St. Louis San Francisco Auckland Bogotá
Caracas Hamburg Lisbon London Madrid Mexico Milan Montreal
New Delhi Paris San Juan São Paulo Singapore Sydney Tokyo Toronto

This book was set in Times Roman by the College Composition Unit
in cooperation with Waldman Graphics, Inc.
The editors were David M. Shapiro and Ira C. Roberts;
the production supervisor was Leroy A. Young.
The cover was designed by Carla Bauer.
R. R. Donnelley & Sons Company was printer and binder.

THE ARCHITECTURE OF SYMBOLIC COMPUTERS

2 3 4 5 6 7 8 9 0 DOC DOC 9 5 4 3 2 1 0

ISBN 0-07-035596-7

Library of Congress Cataloging-in-Publication Data

Kogge, Peter M., (date).
The architecture of symbolic computers / Peter M. Kogge.
p. cm.—(McGraw-Hill series in supercomputing and parallel processing.)
Includes index.
ISBN 0-07-035596-7
1. Computer architecture. 2. Electronic digital computers—Design and construction. I. Title. II. Series.
QA76.9.A73K64 1991
004.2'2—dc20 90-42806

ABOUT THE AUTHOR

PETER M. KOGGE is a Senior Technical Staff member in IBM's Federal Sector Division. In this capacity he serves on several technical advisory boards responsible for overseeing advanced computing technology and business directions for applying them. His present areas of expertise are highly concurrent computers, non-von Neumann architectures and programming languages, real-time logic programming and expert systems, use of VLSI in computer implementations, and the theory and design of matching parallel algorithms.

Dr. Kogge is also the author of the international bestseller *The Architecture of Pipelined Computers*, has published frequently in the technical literature, and has lectured extensively at major universities and research centers. He has participated in the editing of many computer publications, including the *IEEE Transactions on Computers* and the *International Journal of Parallel and Distributed Computing*, and was General Technical Chairman of the 1989 International Conference on Parallel Processing. Dr. Kogge received his Ph.D. in electrical engineering from Stanford University. He is a Fellow of the IEEE, and a member of the IBM Academy of Technology.

CONTENTS

Part II Function-Based Computing

Part III Logic-Based Computing

PREFACE

There is a revolution looming in computer science, not just in the speed and memory capacity of the computing equipment we use, but in the very way we think about, specify, and perform computations. Essentially, the basic von Neumann model of computing, which has ruled unchallenged for the last 40 years, is encountering not one but several potentially heavyweight challengers. This text represents an attempt to provide some insight into the conceptual seeds of the revolution, and the directions in which they are growing. As such, it was written with two overall goals in mind. First, we wish to develop in the reader a firm understanding of the mathematical roots of two trunks of these new computational models. Second, we want to demonstrate in a very concrete fashion how such models can and have been put into practice as real programming languages running on real processors, often with revolutionary design characteristics. Achieving these two goals should give the reader the ability to follow, or better yet participate in, construction of the languages and computing systems that will become the everyday computational tools of tomorrow.

VIEWPOINT

The viewpoint taken in this text is that of a practicing computer architect, where ***architecture*** is taken in its broadest meaning as "the art or science of building,..., esp. habitable structures" (*Webster's Third International Dictionary*). In our case the structures are computing systems, and their inhabitants are the sophisticated, largely nonnumeric programs found today in prototype form in highly interactive environments, artificial intelligence, intelligent databases, and similar advanced applications, but moving rapidly into tomorrow's mainstream.

When building such structures, a true architect must consider the interplay between both the hardware (machine organization) and the software (compilers, interpreters, and runtime routines) needed to sustain the inhabitants. In this text the emphasis on these structural aspects is continuous, balanced, and interleaved with discussions of the formative backgrounds of the appropriate programming languages. From personal experience, it is impossible to do otherwise. A clever hardware organization idea is simply wasted gates if the support software does not employ it efficiently, and a high-performance compiler technique is of only academic interest if the underlying hardware has no hooks with which to sustain it. Further, neither has value if the application or programming language used does not permit or encourage expressions that use them.

The flow of presentation reflects this viewpoint, with the initial focus on the underlying mathematical frameworks that drive the various computing models, followed by detailed discussions of how some key languages based on such frameworks affect the architecture of the compiler and machine hardware used to support them. In general, the transition between mathematical models and real implementations is through the use of ***abstract machines,*** i.e., computer architectures that may not normally be implemented as real hardware but that are especially easy or good targets for compilers for such languages. Once the abstract architecture and matching compiler technology has been discussed, real languages, machines, and implementations become relatively direct extensions.

Since the emphasis is on architecture, the approach taken toward the mathematics is of introducing the key ideas and notation that drive the language definitions, and not on detailed proofs and arguments. Similarly, the reader should not expect to take away the ability either to program in specific languages or to duplicate in detail particular machine designs or programming systems. Rather, he or she should expect to receive a formal introduction to sets of architectural design principles that have proved successful in actual systems.

THE NATURE OF THE BEASTS

All the major concepts discussed here revolve around the idea that much of the computation of the future will process ***symbols*** rather than just simple numbers, or aggregates of numbers. To be specific, this means creating, converting, evaluation, and interpreting strings of symbols, usually by the ***substitution*** or ***matching*** of one set of symbols into or with another set in some fashion. While this may seem to be the opposite of what one does in today's programming languages, anything that can be done in a conventional language can also be done using these new concepts, usually more easily, more concisely, and more naturally. Further, the elimination of many of the old crutches often opens up opportunities for true parallelism, and thus very high performance, something that is simply not possible with today's computing models.

The two possible directions for such processing discussed in this book are ***function-based*** computing and ***logic-based*** computing. Both are often called ***declarative languages*** as opposed to conventional ***imperative languages.***

Function-based computing involves "applying" substitutions of operands into expressions as its major computational operation. The name comes from the mathematical theory of ***functions,*** which involves the formal definitions of mappings from one set of symbols to another, with substitution serving as the conceptual implementation vehicle. This mode of computing has some similarities to the structure of classical languages in that one describes, given some specific inputs, the process to follow to build up the desired result. It differs from classical computing in that there is no direct equivalent to, or need for, such things as "variables" corresponding to specific memory locations, "assignment statements," "program counters," "sequential statement execution" to change the contents of variables in a certain sequence of steps, or "side effects" which can potentially change some object at a great distance from the current site of computation. There are no "commands" to change the values assigned to names. Instead, one simply "declares" that some symbol "is" the same object as that described by some expression, and builds new objects on top of that.

Interestingly enough, this has no effect on the computational power of function-based programming languages. Further, when done properly, it yields programming languages in which functions are truly "first-class citizens"; i.e., anything that can be done to objects such as integers in conventional languages can be done to functions. They can be passed as arguments, examined by other functions, and returned as results. This means that a piece of code can dynamically test, modify, and generate new pieces of code, which may either be stored or executed immediately.

Logic-based computing, on the other hand, involves finding substitutions that make certain expressions, or sets of expressions, have some desired properties. Its mathematical base is that of ***relations***—namely, identifying when various sets of objects have some property. Programming is again ***declarative*** in that a typical statement simply "declares" that if certain objects have some properties, then some other objects have other properties. Computation often consists of methods of generating potential substitutions, and then identifying which ones are appropriate, often by sophisticated pattern matching between properties of the proposed substitutions and properties of the problem. The underlying processing engine is responsible for finding an appropriate trace of such substitutions when presented with a specific input. This trace yields such interesting capabilities as the ability to ask "why" or "how" the answer was derived; or to run the same program two ways, either in the classical form of "given an input question, find an output answer," or, more unusually, in the form, "given an answer, what was the question?" Again, these are capabilities that are not found in conventional computing systems.

ORGANIZATION

The chapters in this text are divided into three main parts. The first part, encompassing Chapters 1 through 3, reviews the mathematical fundamentals and common notation used throughout the rest of the text. Two things are of most importance here. First is an introduction to the concept of substitution as something useful to describing computation. Second is ***abstract programming notation*** and the concept of ***abstract syntax.*** Together, they will be used throughout the text to permit very short, but extremely readable, descriptions of how to intrepret programs written in the various models.

The other two major parts each include nine chapters, and cover function-based and logic-based computing, respectively. Each part shares a common organization. First is a set of three chapters devoted to the common mathematical fundamentals of the particular computational model. The emphasis is on introducing the basic ideas needed to express computations in the model, and on those key mathematical results which define the range or limits of such computations.

Next is a set of three chapters that starts with a clean form of the programming language most widely recognized as being representative of the model discussed in that part. This is LISP for function-based computing and PROLOG for logic-based computing, Following this is the description of a simple abstract machine for which it is easy to generate efficient code from programs written in the language. This in turn permits short but complete descriptions of how to extend these simple compilers and abstract machines to real compilers for real machines.

Finally, each part concludes with a set of chapters that describe other programming languages and machines for that class of computation, with emphasis on those that have some unique characteristics (such as combinators, relational databases, or production-rule systems) or exhibit features or performance possibilities (such as extensive parallelism) that are not found in the others.

AUDIENCE

While this text will be of value to anyone interested in unconventional computing, its intended audience are those students and practicing professionals interested in the innards of computation. Specifically, this includes hardware architects responsible for new computer organizations, and their software counterparts who construct matching language-processing and system-support programs. High-performance workstation projects come to mind as a prime example of this.

Next, applications programmers faced with the task of choosing, learning, and using a non-von Neumann language will find the text useful in defining the origins and meanings of those language concepts that are foreign to conventional computing, and in understanding the key implementation issues that affect a program's performance on a real machine.

Additionally, programming language designers can use the text as a starting point for identifying new concepts in expressing computation, with emphasis on their mathematical backgrounds and on implementation techniques and trade-offs.

Finally, researchers in unconventional computing can use the material here as a basis for understanding the foundations, strengths, and limitations of the most up-to-date models, and can use that as a map to identify potential successors.

REQUIRED BACKGROUND

This text was designed to be suitable for people with a broad range of backgrounds. In most cases the basic material is compatible with an upper-level computer science or computer engineering undergraduate background. In terms of a recently proposed undergraduate computer science curriculum (cf. Alfs Berztiss, "A Mathematically Focused Curriculum for Computer Science," *Communications of the ACM,* vol. 30, no. 5, May 1987, pp. 356–365), a minimal background would include "Languages and Programs" (CS5), "Advanced Programming" (CS7), "Discrete Mathematics (I)," and "Computer Architecture" (CS150). Together, these courses will give the reader the necessary familiarity with several conventional programming languages, some advanced programming concepts such as recursion, the concept of an instruction set architecture, how machines might be built to execute them, the concept of a compiler, and an introduction to the notion of formal definitions of a language's semantics.

Although not required, there are courses at the master's level which greatly enhance a student's educational experience. In terms of a proposed master's curriculum (cf. K. I. Magel et al., "Recommendations for Master's Level Programs in Computer Science," *Communications of the ACM,* vol. 24, no. 3, March 1981, pp. 115–123), a prior course such as "Compiler Construction" (CS19) or (Berztiss, CS221) would enable a typical reader to actually attempt using the material from this text as a guide to the construction of working translators for real languages.

Likewise, other courses such as "High Level Language Computer Architectures" (Magel, CS26) and "Large Computer Architectures" (Magel, CS26) would permit a student to recognize more fully those processor design concepts that are novel to these new modes of computing. Similarly, a course such as "Formal Methods in Programming Languages" (Magel, CS20) would allow for a deep appreciation of such topics as parallelism, lazy evaluation, backtracking, and nondeterminism. Finally, a course in "Artificial Intelligence" (Magel, CS12 or Berztiss, CS160 or Berztiss, CS272) would give the student a background with which to justify by experience why the concepts described are so important, and for what typical kinds of applications they are particularly suited.

CURRICULUM USE

As implied above, the material in this text can (and was designed to) support advanced undergraduate and graduate-level computer architecture courses in several ways. First, and most important in the author's view, is its direct use in a course entitled something like "Alternative Computing," which brings together what is today several diverging tracks in most computer science curricula, namely, programming languages, computer processor design and organization, and system software. Such a course can be formatted at least two ways. A single-semester version could cover only the core chapters from the two models, with a few lessons at the end on selected topics to be chosen by the instructor. This would introduce the basic principles involved in the models selected and give, through the relevant abstract machines, an overview of their implementation techniques. This would be most compatible with an advanced undergraduate student body.

A two-semester version could cover both models of computing in more detail. With care, the semesters could be structured to be relatively independent of each other, permitting students to take the second without the first, or vice versa. One way to do this would be to cover functional computing in the first semester, and logic-based computing in the second. Again, there is sufficient material for an instructor to pick and choose topics of special interest. This would be most compatible with a graduate student audience.

The author has successfully used both formats in courses taught over the last 5 years at the State University of New York at Binghamton.

Alternatively, this text could be a valuable augmentation to several existing courses. Classes in LISP or PROLOG, for example, could use the appropriate chapters for a more in-depth look at the languages' semantics and typical implementations than can be found in most programming manuals. Advanced architecture classes could use the text as a source of information for symbolic processors, just as other texts are used as references for parallel, pipelined, or distributed processors. In this case an advantage of this text is the close mixing of computer organization techniques and the compiler technologies needed to take advantage of them. Similarly, advanced programming language courses could use this text as an integrated reference to a wide variety of languages.

Finally, the text has been designed explicitly to be largely self-contained in concepts but with extensive references to the original sources, permitting experienced professionals to use it for either reference or self-study.

NOTATION

In any book that attempts to cover as broad a range as this one does, it becomes important to pick notation that is both internally consistent and reflects at least approximately what real systems use. While somewhat arbitrary, and guaranteed to conflict with some part of each reader's background, the notation I have chosen is an attempt at standardization. For example, for lists I have adopted the dot notation and parenthesis notation of LISP. This does not agree with many standard PROLOG notations, but has proved very readable through the years. One impact of its usage on other notation, however, is in Backus Naur form (BNF) syntax descriptions. To minimize confusion, the format chosen here deliberately does not use "( )" as a basic BNF symbol. Their place is taken by "{ }."

As another example, nearly all the basic mathematics needed involves ***quantified expressions*** at some point, where a quantifier symbol of some sort (such as ∀, ∃, λ, ...) controls some aspect of an identifier's usage in some expression. Each author seems to adopt his or her own notation for such expressions, with use of ".," "( )," and the like being common. The notation chosen here puts the quantified variable after the quantifier and separated by a "|" from the expression being controlled, e.g., $\exists x|$ (**real** $(x) \wedge (1<x) \wedge (x<2)$). While this matches no particular standard, it is consistent, and it minimizes confusion with the other notations.

Finally, in the printing of this text, the first time a term is introduced it is highlighted in a ***font like this***. The names of **objects**, *variables*, **sets**, **relations**, and **functions** are as shown, while keywords from programming fragments are in a separate font, as in if-then-else. Syntactic terms defined by BNF productions are in brackets, as ⟨expression⟩.

For simplicity and ease of readability, the fonts used in figures and tables are a subset of these.

CONCLUSION

This text has been a labor of love for the author. In more than 20 years as a practicing computer architect and 13 years as a teacher, I have come to realize the fundamental importance of the intertwining of language fundamentals, compilation, and machine architecture. With such understanding comes a deep sense of rightness and completion, and the ability to see behind a snippet of code to the guiding principles and implementation mechanisms that support it. This is what I want to pass to the reader.

To those readers who have actively participated in the development of machines or languages that should perhaps be referenced here but are not, I apologize. The field is so large, and growing so fast, that it would be a monumental task to do adequate justice to all alternatives. My hope is that I have picked ones that have some longevity, and I would be overjoyed to hear from researchers who could educate me on others that should be considered for future editions.

Finally, although I have enjoyed writing every sentence of this book, it is not mine alone. My vision of what it means to be an architect comes from my Ph.D. advisor at Stanford, Harold S. Stone. A large measure of creative criticism and suggested improvement belongs with my students at SUNY Binghamton, who suffered through early versions of the material, and with the faculty at SUNY and my coworkers at IBM, without whose complete support this work would not have been possible. The editorial staff at McGraw-Hill, especially David Shapiro and Ira Roberts, also had a major impact on the final product. I would also like to mention the following reviewers who provided valuable critiques on earlier drafts of the manuscript: Anand Agarwal, Massachusetts Institute of Technology; John P. Hayes, University of Michigan; John Peterson, University of Arizona; and Keshav Pingali, Cornell University. Perhaps, however, even more credit should go to my family, particularly my son Michael for keyboarding the index, and especially my wife Mary Ellen, for putting up with lost Monday nights and extended periods of hand-keyboard attachment.

Peter M. Kogge

THE ARCHITECTURE OF SYMBOLIC COMPUTERS

CHAPTER 1

FUNDAMENTALS OF COMPUTATION

The forms of computation discussed in this book are sufficiently different from that of the FORTRAN, Pascal, C, . . . , worlds that it is worth the initial effort to step back and review quickly the underlying nature of computation. This chapter is such a review. It starts with an informal definition of problems and solutions in general, continues with how one expresses the process of computing such solutions, and ends with a summary of the mathematical underpinnings used by the computational models developed later in this book.

This latter summary actually covers three separate but very related theories, that of sets, relations, and functions. The most fundamental of these, ***set theory,*** helps define what an ***object*** is, and will lead in the next chapter to a data structure called a ***symbolic expression*** or ***list,*** which is an integral part of nearly all the languages addressed in this book.

Building on set theory is ***relation theory,*** which deals with partitioning groups of objects into subsets (called ***relations***) that have some particular (often user-defined) properties. This forms the basis for the ***conditional execution*** or ***if-then-else*** form of computation found in all models of computation. It is also the basis for formal logic, and thus of the ***logic-based computing*** model addressed later in this book.

Finally, ***function theory*** investigates a specific class of relations that have particularly useful characteristics for computation. Again, it shows up in one form or the other in most models of computation, and it is the basis of the first model of computation discussed here, ***function-based computing.***

A reader with a background in these topics should feel free to skim this chapter quickly, with a few important exceptions:

- The definition of a ***variable*** and ***substitution*** in the mathematical sense, and the dramatic difference between our usage here and conventional usage
- The concept of an ***abstract machine*** that is both simple to understand and an easy target for a compiler of appropriate languages
- The idea of a ***curried function*** that accepts less than its full complement of arguments and delivers a function result that would accept the rest at a later time

These concepts will be introduced here and developed more fully in later chapters.

1.1 PROBLEMS AND ALGORITHMS

(Garey and Johnson, 1979)

The normal reason for using computers is that we have a ***problem*** for which a ***solution*** is desired. Usually such problems are generic; there are one or more ***parameters*** for which values are not provided when the problem is stated, but when some of them are given values, they "personalize" the problem description to correspond to some specific case. ***Solving a problem*** involves providing input values to some of these parameters, and then finding matching values for the remaining parameters. Note that there is nothing here that guarantees a solution exists or that, if it does, there is a single, unique one.

Many such input assignments can be given to these parameters; each assignment specifies a special case of the problem called an ***instance,*** and it has its own matching solution. For a particular instance, or class of related instances, a problem is said to be ***deterministic*** if there is exactly one solution; it is ***nondeterministic*** if more than one solution is possible. An example of the former is computing the factorial of a number. An example of the latter is looking up in a phone book the number of someone whose last name is Smith.

1.1.1 Variables, Substitutions, and Computation

Within the context of the problem's description, these parameters are often called ***free variables,*** or simply ***variables,*** because they are not bound to a specific value in the problem's description and are "free" to be given any value in a real instance. Further, once given a value (called ***binding*** a value to a variable), such a variable becomes a ***bound variable*** that "never changes," at least for the duration of the computation of the matching solution. The name of the variable becomes synonymous with the single value given it.

Again, a variable in a mathematical sense (and for most of the languages discussed in this book) can be either unbound or bound at most once and to one value. Note the strong distinction from conventional computing. A parameter is called a variable because for *different* instances one can give it different values, and not because it has of necessity any sort of "storage" associated with it, whose value can change with time within a single computation.

Note also the idea of ***substitution.*** Creating an instance is conceptually equivalent to replacing in the problem's description all copies of the input parameters by their assigned instance values. These are called ***input substitutions.*** Likewise, the ***solution*** to the instance is a substitution of some values to the remaining parameters which together are consistent with the input values. In analogy, these are ***output substitutions.***

In addition to parameters, the ***formal statement*** of a problem usually entails a description of the ***properties*** that the output parameters must have in relation to the input parameters. Note that this description usually tells absolutely nothing about how to solve the problem, only how to recognize a solution when you have one.

Thus ***computation*** is the process of determining an output substitution which, for a specific input substitution, obeys all the specified properties of the problem.

As an example, consider the ***knapsack problem*** (Figure 1-1). Here we have a knapsack that can hold only so much contents in terms of total weight. We also have a set of objects, each of which has a certain weight and a certain measurable "value." These objects are like rolls of bolo-

Given: • A knapsack of carrying capacity C
• N objects of weights W_k and value V_k, $k = 1, \ldots, N$

Find: A fraction F_k of each object that:

- Does not overload the knapsack
- Maximizes value carried
- Where objects can be "sliced" into pieces

(*a*) Formal problem statement.

$C, N, W_1, V_1, F_1, \ldots, W_N, V_N, F_N$

(*b*) Its parameters (variables).

$C = 14\text{kg}$	$N = 3$	
$W_1 = 4\text{kg}$	$W_2 = 6\text{kg}$	$W_3 = 7\text{kg}$
$V_1 = \$30$	$V_2 = \$48$	$V_3 = \$50$

(*c*) An instance (input substitution).

100% of objects 1 and 2
4/7 of object 3
Total value = $\$106.57$

(*d*) A solution to the instance (output substitution).

FIGURE 1-1
The knapsack problem.

gna—we can slice off any amount we want, with the value of that slice being in direct proportion to its relative weight. A "solution" to the problem will be some set of slices of the objects that does not overload the capacity of the knapsack, but that maximizes the total value carried. A particular instance of the problem involves specifying the capacity of the particular knapsack of interest, and the number and characteristics of the objects available to place in it. A solution consists of the fractions of each object placed in the knapsack.

Note that adding additional constraints to the problem can change it radically. For example, one could say that the objects are indivisible, or that the knapsack has both weight and volume constraints. In such cases, instances for the old problem may remain valid instances for the new problem, but the corresponding new solutions may have nothing in common with the original ones.

1.1.2 Algorithms

To emphasize a key point again, the statement of a problem specifies only the properties of the solution, not how one finds it for a particular instance. Describing this latter process is often done via an ***algorithm,*** a step-by-step description of how, given a particular instance, one goes about locating or computing the substitution of an object or objects into the output parameters that satisfy the properties demanded of the solution.

Executing an algorithm occurs when a particular instance is available, and the steps of the algorithm are carried out with the values from the instance "substituted" for the problem's parameters in the algorithm's description.

Figure 1-2 gives the classic algorithm for the knapsack problem. This example demonstrates most of the ***properties of an algorithm.*** It is not a "crystal ball" which returns an answer by magic in zero time. Nor is it something that must run forever before the answer will be available. Instead, an algorithm is characterized as taking a finite (possibly large but still countable) number of basic steps, each of which in turn uses a finite number of well-defined "simple" operations and which again runs in finite time. Although an undefined "solve the problem" operator is not permissible, it is possible for one step or operation to invoke other algorithms to solve subsidiary problems. These other algorithms might include a copy of the current algorithm, but with a (hopefully) different instance to solve. This latter case is called ***recursion,*** and it plays a large role in the mathematics for the computational concepts developed in this text.

```
1. Sort the objects by ratio of value to weight
2. Repeat until the knapsack overflows:
   Remove highest value-to-weight object from set, and place entirely in knapsack.
3. Take out the last object placed in knapsack, and slice off just enough to fill
   knapsack.
```

FIGURE 1-2
An algorithm for the knapsack problem.

The ***validity*** of an algorithm is a mathematical proof that, given any meaningful instance of the original problem, the algorithm always returns a result that satisfies all the properties the solution must have. This is usually a very complex process, and for today's common methods of computing is done at best informally if at all. For example, to prove the validity of the above knapsack algorithm requires arguments that suppose that the value derived is not as high as some real solution, and then proving that this assumption causes a contradiction.

1.2 LANGUAGES: SYNTAX AND SEMANTICS

A ***language*** is a notation for conveying meaning or information from one person to another. For this book the only such notations that we will use are written ones formed from linear left-to-right sequences of ***symbols*** (called ***characters***), although other notations, such as two-dimensional graphs, sound waves, raised dots on a surface, finger motions, or light patterns, are certainly possible.

In any language, only certain forms of ***notation*** (certain combinations of symbols) are allowed. The language's ***syntax*** or ***grammar*** comprises rules that describe these valid combinations.

Further, in order to convey information, it must be possible to relate syntatically valid pieces of notation to particular meanings. A language's ***semantics*** describe such relationships.

This difference is important: Syntax addresses the form of a valid sequence of symbols, and semantics refers to its meaning. Interpreting a particular symbol string requires a knowledge of both.

Recalling the prior distinction between problems and algorithms, we define a ***programming language*** as a formal language for describing problems or algorithms. A ***procedural language*** is one that emphasizes the description of the steps of the algorithms; a ***declarative language*** is one that emphasizes problem descriptions, particularly the desired characteristics of the solution. Nearly all of today's conventional programming languages are procedural. In contrast, nearly all those discussed here are heavily declarative.

A ***program,*** then, is a syntatically complete description of one algorithm or problem statement. The smallest piece of notation in a program that describes an individual step in the algorithm is a ***sentence*** or a ***statement.*** There may be many possible programs in the same programming language that describe the same algorithm, differing in the number, ordering, or content of the individual statements, but all yielding substantially the same result when given an instance. In some sense each such program may itself be considered an "instance" of the set of all possible programs that can be written in the programming language.

With these definitions, we define a ***computer*** as some agent, usually electronic, that interprets a syntatically valid program according to the language's semantics, and essentially "executes" the equivalent algorithm for a particular instance. The result of the execution is a set of output substitutions that solve the problem instance given as input.

Notice again the idea of ***substitution.*** Given a particular instance of a problem, a computer solves it by conceptually substituting the specified values for the parameters into the appropriate places in the program, and then executing the individual statements with these substitutions in mind.

As an example of syntax rules, consider the process of writing "well-formed" integers in many languages. Characters are written from left to right, and may start with an optional "sign" (either "+" or "−"). Following this there may be one of more digits (from 0 to 9), terminated by a space or some other valid delimiter. By convention, leading 0's are valid, but not often written.

The "semantics" of such notation is that it corresponds to a number whose value is formed by taking the digits from right to left, multiplying each by increasing powers of 10, and adding the results. If there is a leading "−," the number is considered to be negative. Note that just this semantic meaning can be seen in typical student programs that take such character strings and convert them into equivalent internal binary notation.

As another example, consider the typical ***assignment statement*** found in many conventional programming languages. The syntax of such statements specifies that a ":=" (or equivalent) separates two smaller pieces of notation. The piece on the right specifies some computational sequence. The piece on the left essentially specifies some location in memory. The semantics of this is that one first interprets the piece on the right, and evaluates it down to some object. Then one interprets the left piece down to the point of finding a memory address. Then the former value is copied into the latter location.

As an example, the Pascal statement "x:=x+13" breaks down syntatically into a "variable" x, and a "variable x plus integer 13," which in turn "means:"

1. Get a copy of the contents of the current memory location associated with the symbol x.
2. Interpret it as a number.
3. Add the number corresponding to the character string "13" to this number.
4. Convert it back into the form expected by the type of x.
5. Place those bits back into memory at the same location found in step 1.
6. Continue with the next statement.

Notice here the time dependency in the semantics, plus the nested "meanings" of numbers, addition, conversions, etc.

In both cases above, the reader should again note the clear distinction between syntax and semantics. The former describes what character strings are valid notation; the latter describes the actions or results brought about by the computing agent when presented with the strings.

1.3 SEMANTIC MODELS

(Horowitz, 1983a; Stoy, 1977; Gordon, 1979)

The ***semantics*** of a piece of notation is its ultimate ***meaning***. This is of obvious and paramount importance to a programmer who wishes to create a piece of notation (a program) whose meaning to the computer that executes it will direct it to solve the desired problem. In turn, this means that the programming language designer must be very careful when formulating a language that its semantics are perfectly clear to all involved with the language. Likewise, the computer architect must understand the ramifications of such semantics for the way the hardware in a computer must interact.

Informal methods for transferring this information include reference manuals, sample programs, help screens, etc. Such techniques are fine when the designers are also implementers and primary users, or where the languages are very small. Many real programming languages, however, have become quite complex, with designers and implementers (let alone users) often being groups who have no direct contact. Further, the support tools for such languages are becoming increasingly more complex and themselves dependent on a precise understanding of semantics. Examples of these include languages for writing compilers for other languages, "smart" program editors, high-level debugging tools, and program verification packages.

As one very simple example of the necessity for an understanding of semantics, consider boolean expressions based on the equality test, for example, "x=y." The "normal" meaning for such an expression is that it represents a truth if both x and y have the same value. However, there are other "meanings" possible, for example:

- Are they exactly the same object (at the same memory location)?
- If they are at different locations, do all their subcomponents match in order and value?
- Can they be made equal by giving values to embedded variables?

There are programming languages, LISP for one, where there is not one but several "equal" tests, all with different meanings. Other languages, like Prolog, emphasize one of these, but not the conventional one (in Prolog's case it is the final one above).

Continuing this example, in mathematics expressions like "x=x"

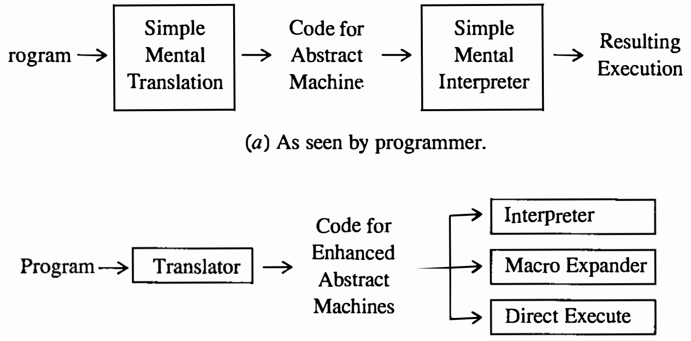
are always true, regardless of what "x" is. This is not true of most programming languages, where the idea of time-ordered execution of parts of a program is an important part of the semantics. Thus "**random( )=random( )**" is hardly ever true, because each "call" to the function **random** returns a different value. Expressions like "**foo**(x)=**foo**(x)" are likewise practically impossible to understand without a solid understanding of the semantics that governs what piece of notation is "executed" before what other piece, and what can be changed as a result of its execution (e.g., consider what happens if **foo** returns as its value the value of x, but internally increments x by 1 as a by-product.)

With these kinds of problems in mind, computer scientists have developed three major methods of defining and describing the semantics of programming languages:

- ***Interpretative semantics,*** where meaning is expressed in terms of some simple ***abstract machine***
- ***Axiomatic semantics,*** where rules describe the data values given various objects before and after execution of various language features
- ***Denotational semantics,*** where syntatic pieces of a program are mapped via ***evaluation functions*** into the abstract values they "denote" to humans

1.3.1 Interpretative Semantics and Abstract Machines

Interpretative semantics, often called ***operational semantics,*** is perhaps the one that most human programmers use at least unconsciously. In this approach we have some rather simple ***abstract machine,*** which exhibits the essential functioning of the real machines the programs of interest will run on, and about which there is absolutely no doubt (to trained humans) how they would work given real programs (see Figure 1-3). The


rogram
Simple Mental Translation
Code for Abstract Machine
Simple Mental Interpreter
Resulting Execution
Program
Translator
Code for Enhanced Abstract Machines
Interpreter
Macro Expander
Direct Execute

(*a*) As seen by programmer.

(*b*) As seen by architects.

FIGURE 1-3
Uses of abstract machines.

semantics of a real programming language are then specified by how each piece of notation would be translated into code for such a machine. Questions about what would happen to certain programs would thus be answered by "translating" the program into this abstract machine code and then "executing" it, often mentally.

Because of its natural relation to machine architecture and compiler design, this approach to semantics is featured heavily in this text. Functional languages, for example, have the ***SECD Machine*** and the ***G-Machine,*** while logic-based languages have the ***Warren Abstract Machine*** and the ***Production System Machine.*** Time and again we will start with simple variations of abstract machines and matching simple mental models of how they work, and develop simple ways of converting programs into them. Later on, more robust versions will be developed, with matching translation procedures and real agents for their execution.

The abstract machines described here are not the only examples. Other real programming languages described this way include Smalltalk (see Goldberg, 1986); Algol68 (van Wijngaarden, 1975); and PL/1 [Lucas and Walk (1969) and the Vienna Definition Language of Wegner (1972)].

1.3.2 Axiomatic Semantics

In contrast, ***axiomatic semantics*** resembles closely many of the mathematical approaches to proving the correctness of programs. Semantic "axioms" basically describe how the "state" of a program's execution changes as a result of the execution of some language feature. As an example, consider a language with a built-in procedure **SORT,** which takes as arguments an array and lower and upper indices of that array. What is supposed to happen is the "sorting" of the specified subset of the array. Figure 1-4 gives in an English format what a semantic rule for this con-

For statement SORT(A,m,n)

Before execution:

- m≤n.
- m,n both > 0 and within dimensions of A.
- A[m] through A[n] all have valid values.

After execution:

- m,n are unchanged.
- A[i]≤A[i+1] for m≤i≤n.
- For each original A[i] value, m≤i≤n, there is some j, m≤j≤n, where the new A[j] has the same value.
- All other values of A are unchanged.
- No other object is referenced or changed.

FIGURE 1-4
Sample axiomatic rule.

struct might look like. Note that this describes the "before" and "after." Unlike the interpretative model, it gives no idea of what happens "during" execution.

In this text we will use a very low-level form of axiomatic semantics to describe how instructions in the various abstract machines change machine states. Because of the nature of such axiomatic statements for large applications, they are often written in some more formal notation from mathematical logic, permitting very precise and rigorous reasoning about them.

1.3.3 Denotational Semantics

Finally, ***denotational semantics*** takes the approach of defining how each piece of syntatic notation in a programming language "maps" into, or "denotes," its true or "abstract" values as viewed by a human. This mapping is expressed by ***semantic evaluation functions,*** whose output is "understood" by humans but whose actual implementation may or may not be spelled out in any real detail. The "meaning" of a program statement is thus the translation of that statement into a nested composition of the appropriate semantic functions.

As an example, consider the expression "*A*[*I*]+3." The meaning as expressed in a denotational form might look like this:

add(select(*A*,value(*I*)),value(3))

where value, for example, is a semantic function which, given a character string, returns its current value. Thus value(3) returns the third positive integer, and value(*I*) returns the value associated with the variable *I*.

Stoy (1977) is the fundamental text describing the mathematics of the denotational approach. Gordon (1979) uses it in a complete description of several small but realistic programming languages.

The concepts of ***abstract programs, abstract interpreters,*** and ***abstract syntax*** introduced in the next chapter have much of the flavor of this approach, and will be used throughout the text to describe conceptual (and real) programming languages.

1.4 THE VON NEUMANN MODEL OF COMPUTATION

A ***model of computation*** is a description of how a computer interprets and executes a program written in some programming language. The purpose of the description is usually to provide a basic interpretative semantic model that is easily comprehensible to the programmer and yet reasonably accurate in overall details. As such, models are normally somewhat simplistic, although it is certainly possible to formalize them to the point of precisely defining both the hardware and and support software.

Perhaps the most well-known example is the "classical" ***von Neumann model*** used to describe conventional programming languages and machines (see Figure 1-5). This model has several key characteristics that show up in one form or another in nearly all current computing systems:

1. A ***memory unit*** to hold values and instructions in specific locations
2. A ***central processing unit*** (CPU), which sequences access to the memory
3. An ***instruction,*** which specifies some particular sequence of activities in the CPU that modifies the contents locations in memory or in CPU ***registers***
4. A ***program,*** which consists of an ordered sequence of instructions placed in the memory
5. A ***Program Counter*** in the CPU to select an instruction from the memory
6. A ***programming language,*** which specifies how to assemble instructions into memory

A key characteristic of this model that shows up in virtually all real implementations is that all access to the data in the memory must be done more or less sequentially under the control of the CPU. This tends to limit the peak performance of the computer to something related directly to the "width" and "speed" of the path between the CPU and the memory. For obvious reasons, this limitation goes by the popular name of the ***von Neumann bottleneck.***

Performance of a particular computer for a particular program is of-

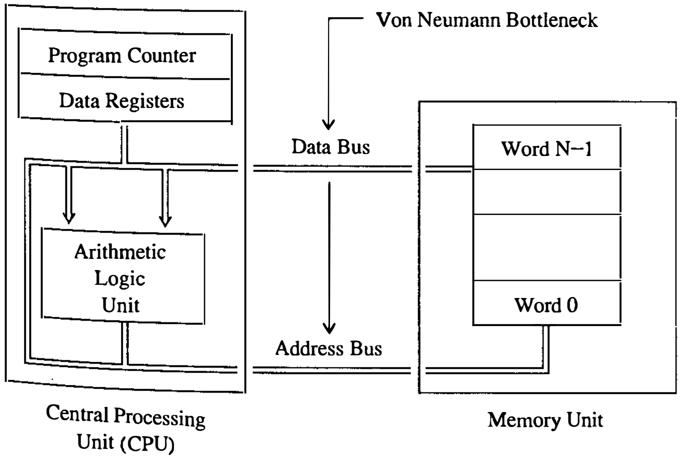


FIGURE 1-5
The von Neumann model.

ten measured in terms of how many ***instructions per second*** the machine can execute. Today's machines often run in the range of millions of instructions per second, or ***mip.***

1.4.1 Comments on the Model

This model has several very important characteristics. Above all, the major purpose of executing a program in such a system is to "change memory." One must know what locations hold initial data before the program is executed, and what locations to look in after execution is complete. This is reflected throughout the syntax and semantics of conventional programming languages, where perhaps the single most important type of statement is the ***assignment statement.*** Such statements specify some specific memory locations to be read, some computations to be performed on them, and some other locations to receive the results. These locations are often designated symbolically by giving them "names" and letting the compiler find exact locations for them. By convention these symbols are also called ***variables.*** Use of the same name at different parts of the program refers to the same memory location with its time-varying contents, and not (as in mathematics, for example) with some "value" that sits still. In fact, this process of allocating storage to variables and then binding and rebinding values to them becomes so important that much of the surrounding syntax and semantics of a typical programming language and much of the actual compilation time quickly devolves into deciding where to put things in memory, and not on the original problem of performing the desired computations.

This concept of "changing memory" also means that two programs cannot be composed with one using the output of the other, without guarantees that the memory map of outputs produced by the first program exactly matches the needs of the input to the second program. Similarly, it means that unless one has a very detailed understanding of the execution of a particular program on a particular computer, it is virtually impossible to relate the contents of memory back to the original computation if one stops the program in midstream for any reason.

In this book the ***assignment*** notation ⟨register/location⟩ ← ⟨expression⟩ denotes that the specified machine register or memory location on the left has its contents replaced by the value on the right. When multiple assignments are strung together, they are assumed to be executed in that sequence.

Finally, as mentioned before, introducing the idea of time-sensitive locations is the basic cause of the performance limits of the von Neumann bottleneck. By returning to the purer idea of a variable that receives a value exactly once, the computing models described in this text have opportunities for significantly breaking this bottleneck while at the same time greatly simplifying the overall semantics.

1.5 OBJECTS

When discussing problems and algorithms, the instances and results must all relate to something that is "real" in some sense, that is an object of some sort. For the purposes of this text, an ***object*** is anything that can be described, discussed, or distinguished from some other object. Thus the aircraft named the *Spirit of St. Louis* is an object, as is each integer number, or the colors **red, white,** and **blue.**

The key property that makes something an object is that it has a ***value*** associated with it that can be compared in some sense with the "values" associated with other objects. The ability to distinguish objects permits us to give them ***names*** (distinct from the values) by which we can refer to them. In many cases this name is either commonly accepted shorthand for the value or some formal representation of the value's structure or origins. Thus "2" is the Arabic name of the object whose value is the second positive integer, while "II" is the Roman name for exactly the same object. Anything that "2" implies is implied by "II" also. Note in particular that the Roman name represents very concretely the object's value. In either case, the object discussed is always distinguishable from the third integer from 0, which again can have a variety of names, but never the same as that for two.

The above definition of an object is not limited just to simple things like numbers. Collections of objects are objects, algorithms which take objects as instances and identify objects that are results are themselves objects, and even the ***characteristics*** or ***types*** of object values are valid objects. As an example of the latter, the concept of **color** itself can be considered an object because it can be distinguished from similar definitions of **size, shape, smell,** or even **number.** Likewise, one can consider as separate objects individual integers, the collection of all integers, individual rational numbers, the collection of all rational numbers, the collections of all sets of numbers where each set has elements divisible by some prime, algorithms that compute such primes, etc.

When used to represent its value, the name of an object is often called a ***symbol.*** The symbol **color** is the name of the object that represents all possible colors and can be used interchangeably as its value.

1.6 SETS

(Bavel, 1982; Gellert et al., 1975; Mendelson, 1979)

As mentioned above, very often in mathematics (and computing) we are interested in objects whose values are collections of other objects. When combined with the constraint that no object may appear more than once, such a collection is called a ***set,*** with the individual objects making it up called its ***elements.*** There are two standard ways of writing sets. First is simply to list all the elements separated by "," and surrounded by "{" "}." Thus the set of U.S. coin denominations would be:

{penny, nickel, dime, quarter, half-dollar, dollar}

The other notation is of the form $\{x|$"properties of x"$\}$. Here x is a ***placeholder*** or ***representative name*** for all the objects that have the properties identified to the right of the "|." It is also called a ***set variable,*** but with none of the connotations of variables from von Neumann computing. There is no sense of assigning storage to the symbol, nor is there any way of assigning values to it that might change with time.

Very often we indicate the name of a set by writing the name to the left of a "=" and the set description to its right. Examples include:

Z = $\{x|x$ is an integer$\}$
Coins = $\{x|x$ is a U.S. unit of money made out of metal$\}$
Even = $\{y|y$ is a member of **Z** that is evenly divisible by 2$\}$
Color = $\{z|z$ is set of light waves visible to the human eye$\}$
RGB = **{red,green,blue}**

There is no particular ordering assumed for the elements of a set. Thus if two sets have exactly the same elements but are written in different orders, they are in fact still the same set.

A set is itself an object because it can be distinguished from other sets by looking at its elements. Because of this, it is perfectly permissible to construct sets of other sets to any level, each of which is a different object from the previous one. Thus if **a** is an object, then {**a**}, {{**a**}}, {{{**a**}}}, . . . are all different sets of one element, and all are different from **a** alone.

The ***size*** of a set is a count of its elements. This may range from 0 (the set is ***empty***) to infinity. The ***empty set*** in particular plays a very important role in mathematics, and is denoted in this text as either "{ }" or "∅." Even though it has no elements in it, it itself is a perfectly valid object, with all the capabilities of being compared to and embedded in other sets.

1.6.1 Set Comparisons

Of all the questions that might be asked about sets, the most important deal with comparing one with other objects. For this the most basic test is ***membership.*** We say that the object x is a member of the set **A** (written $x \in$ **A**) if x is one of the elements making up **A**. For mathematical reasons, the null set ∅ is a member of all other sets, including itself.

With this definition there are a variety of ways of comparing two sets. For example, the set **B** is a ***subset*** of the set **A** (written **B** ⊆ **A**) if every object that is a member of **B** is also a member of **A**. If **B** ⊆ **A** but there is some element of **A** that is not a member of **B**, then **B** is a ***proper subset*** of **A**, written **B** ⊂ **A**. If **B** ⊆ **A** and **A** ⊆ **B**, then all the members of either set are members of the other, and thus the sets are ***equal,*** written **A**=**B**. Finally, two sets are ***disjoint*** if the only object they have in common is ∅.

Note that equality of two sets is independent of the written order of the elements in the sets.

1.6.2 Combinations of Sets

Sets can also be combined with sets to produce other sets. Figure 1-6 lists the most common such operations.

1.6.3 Tuples

Throughout this text we will often have the need to discuss setlike objects where the order of the elements is important. Such objects will be called ***ordered n-tuples*** or ***tuples,*** and will be written in "( )" as $(x_1,\ldots,x_n)$, where each x_k is a member of some set $\mathbf{D}_k$. The order here is critical; while the sets {1,2} and {2,1} might be equal (and thus the same object), the tuples (1,2) and (2,1) are not.

Tuples of two elements are very common, and are often called ***ordered pairs*** or ***pairs*** for short.

Elements of a tuple are also called ***components*** or ***arguments,*** and are accessed by their position.

We will often construct sets of tuples, i.e., where each member of the set is itself an ordered tuple object. As such, all the set operations described above are applicable. In addition, however, an operation called ***Cartesian product*** × will take n separate sets and use them to construct a new set of all n-tuples constructable from them:

A∪B = union of sets A and B
= {x|x∊A or x∊B}
e.g., {1,2,3,4}∪{3,4,5,0} = {0,1,2,3,4,5}

A∩B = intersection of sets A and B
= {x|x∊A and x∊B}
e.g., {1,2,3,4}∩{3,4,5,0} = {3,4}

A−B = set difference of sets A and B
= {x|x∊A but not in B}
e.g., {1,2,3,4}−{3,4,5,0} = {1,2}

P(A) = power set of set A
= {x|x⊂A}
e.g., P({0,1,2}) = {{0}, {1}, {2}, {0,1}, {0,2}, {0,1,2}}
Note that again there are no duplicates, and that ∅ is an implicit member of this set.

FIGURE 1-6
Common ways of combining sets.

$$\mathbf{A}_1\times\mathbf{A}_2\times\cdots\times\mathbf{A}_n=\{(\mathbf{a}_1,\ldots,\mathbf{a}_n)\mathbf{a}_k\in\mathbf{A}_k\}$$

As an example, if **A**={3,6}, **B**={**red,green,blue**}, **C**={**hi,lo**}, then

$$\begin{aligned}\mathbf{A}\times\mathbf{B}\times\mathbf{C} = \{&(3,\textbf{red,hi}),(3,\textbf{green,hi},)(3,\textbf{blue,hi}),\\ &(3,\textbf{red,lo}),(3,\textbf{green,lo}),(3,\textbf{blue,lo}),\\ &(6,\textbf{red,hi}),(6,\textbf{green,hi}),(6,\textbf{blue,hi}),\\ &(6,\textbf{red,lo}),(6,\textbf{green,lo}),(6,\textbf{blue,lo})\}\end{aligned}$$

Also, the notation $\mathbf{A}^n$ stands for the Cartesian product of **A** with itself n times. Thus $\mathbf{Z}^3$ equals the set of all 3-element tuples of integers.

1.7 RELATIONS

Given the ability to describe objects as combinations of sets and tuples, the next useful facility is the ability to express relationships between them. We have already discussed simple relationships such as equality and membership, but these are inadequate for such things as expressing that *x* "is the next number after" *y*, *x* "is a part of" *y*, or *x* "is a parent of" *y*. This is the purpose of ***relation theory.***

Mathematically, an ***n-place relation*** **R** over sets $\mathbf{A}_1,\ldots,\mathbf{A}_n$ is some subset of tuples from the Cartesian product $\mathbf{A}_1\times\cdots\times\mathbf{A}_n$. **R** is thus itself a set where each tuple in it represents a collection of objects that together has the desired property **R**. The use of tuples rather than sets is important here because it permits the same object to be used more than once, and permits specification of "ordering" within the relation.

Some examples of relations over different sets include:

parents-of over $\mathbf{ANIMAL}^3$ = $\{(f,m,c)|f$ and m are parents of $c\}$
1+ over **Z**×**Z** = {...,(0,1), (1,2), (2,3),...}
+ over **Z**×**Z**×**Z** = $\{(x,y,z)|z=x+y\}$

An n-place relation is often said to have ***arity*** n. When it is important to a discussion to know what a relation's arity is, it is appended to the relation's name with a "/." Thus for the above relations we would write **parents-of/3**, **1+/2**, and **+/3**.

There are two primary ways of writing a relation: as a set as above, and as a table (see Figure 1-7) where each row describes one tuple of the relation. Note in either case that the definition of a relation as a set prohibits duplicate tuple entries, while the definition of tuples permits duplicate components, but only in a specific order.

Because a relation is a set of tuples, the key test on which all other operations are based is membership—namely, is a particular tuple an element of the relation set? Previous notation expressed such tests as $(x_1,\ldots,x_n)\in\mathbf{R}$. However, for a variety of reasons a more convenient form of the same test lists the relation name first (as a ***prefix***), and the tuple components following surrounded by "( )," as in $\mathbf{R}(x_1,\ldots,x_k)$.

parents-of (f,m,c)			1+(a, b)		+(a, b, c)		
Julius	Matha	Roy	0	1	0	0	0
Roy	Louise	Peter	1	2	0	1	1
Jim	Mary	MaryE	2	3	0	2	2
Peter	MaryE	Michael	3	4		...	
Peter	MaryE	MaryB	4	5	1	0	1
Peter	MaryE	Tim	5	6	1	1	2
Adam	Eve	Able	6	7		...	

FIGURE 1-7
Sample relations.

This notation can be used in the same two ways that the equivalent ∈ notation was: either as a statement that a particular token is in the relation, or as a test that returns true or false depending on whether or not the tuple is in the relation. The choice should be obvious from context.

As examples, the following all describe tuples that are members of their respective relations: **parents-of**(Peter,MaryE,Tim), 1+(8,9), +(4,7,11). Under normal conditions, however, any test of the form **parents-of**(...,...,Adam) or +(4,7,3) should return false.

1.7.1 One- and Two-Place Relations

Relations whose tuples have only one or two components (i.e., arity 1 or 2) have special names. One-place relations usually represent sets of simple objects that have the same property, and thus are called ***properties.*** In such cases we often drop the ( ) surrounding each object as redundant. Examples include:

Positive = $\{x|x\in\mathbf{Z}$ and $x\geq 0\}$
Sweet = {candy,fresh-apples,MaryBeth,...}
Human = $\{x|x\in$ {Adam,Eve} or the parents of x were human$\}$

The prefix notation for relations becomes particularly handy here, because it allows us to write such expressions as **Human**(*x*) and mean that the expression is true only if *x* is human.

A ***binary relation*** is one whose tuples are all pairs. Common examples include set membership, equality, 1+, **father-of,** >, <, ⊆ ....Other important examples (from the artificial intelligence community) include ***is-part-of*** and ***is-a.*** The former indicates if something is part of the construction of something else, where "construction" is dependent on context. Thus wheel **is-part-of** bicycle, and memory **is-part-of** computer.

The latter indicates if something has some particular ***type,*** where the type is an object expressed as the second component of the tuple. Thus we might have an IBM PC **is-a** computer, or a Cortland **is-a** apple. In a real sense this is a generalization of a property.

The two components of a binary relation are important enough to distinguish with their own names. Given a relation **R** over **A×B,** the ***domain*** of **R** is the set **A,** and the ***range*** of **R** is **B.** That subset of **A** which actually shows up in some tuple of **R** (i.e., $\{x|(x,y) \in \mathbf{R}$ for some y}) is the ***active domain*** of **R,** while the corresponding subset of $\mathbf{B}=\{y|(x,y) \in \mathbf{R}$ for some $x\}$ is the ***active range.*** The first element of any pair in **R** is thus the ***domain value,*** and the second is the matching ***range value.***

Binary relations can be described in any of the three ways presented earlier: set, tabular, and prefix. In addition, however, there is another notation unique to binary relations that is particularly useful. This is ***infix notation,*** where the relation name is placed between the two components of the tuple, as in $4 < 5$, Roy **father-of** Peter, 3=3, or the **is-a** examples given above. Such notation is particularly meaningful for humans because it is the same format used in both English and most written mathematical expressions.

Given a relation **R,** it is often possible to talk about its ***inverse relation*** $\mathbf{R}^{-1}$, where the tuples are all reversed. Mathematically, if **R** is over the set **A×B,** $\mathbf{R}^{-1}$ is over **B×A,** and has a set of tuples of the form $\{(b,a)|(a,b) \in \mathbf{R}\}$. For example, for the relation **is-parent-of** (all the pairs where the first element represents a parent of the second element) has the inverse relation **parent-of**$^{-1}$ **(is-child-of),** where the first element is a child of the second.

Even though they are so directly related, such sets often have radically different properties.

1.7.2 Properties of Binary Relations over *A*×*A*

Very often, the domain and range sets of a binary relation are the same; that is, the relation is a subset of $\mathbf{A}^2$ for some set **A.** This commonality permits binary relations to be categorized on the basis of their tuple values in several ways:

- A ***reflexive relation*** is one where for all a ∈ **A,** (a,a) ∈ **R.**
- A ***symmetric relation*** is one where for any $(x,y) \in \mathbf{R}$, (y,x) is also in **R.**
- An ***antisymmetric relation*** is one where the only way both $(x,y) \in \mathbf{R}$ and $(y,x) \in \mathbf{R}$ is if x=y.
- A ***transitive relation*** is one where if $(x,y) \in \mathbf{R}$ and $(y,z) \in \mathbf{R}$, then so is (x,z).

If the only tuples in **R** are of the form (a,a) then **R** is an ***identity relation,*** which is reflexive, symmetric, and transitive.

As shown in Figure 1-8, it is possible for a single relation to have any combination of the three properties. In particular, a relation that has all three is an ***equivalence relation.*** Further, if we divide the set **A** into subsets so that all, and only, the elements of each subset are related to each other through **R,** the result will be a disjoint set of subsets that have no elements in common and cover the set **A** like a blanket. These clusters are called ***equivalence classes.***

Relation	Reflexive	Symmetric	Transitive
+1	no	no	no
>	no	no	yes
ancestor	no	no	yes
≠	no	yes	no
sibling	no	yes	yes
≥	yes	no	yes
subset	yes	no	yes
=	yes	yes	yes ←equivalence

FIGURE 1-8
Sample binary relations.

Computing the elements of an equivalence class is a common operation in many symbolic programs. In its simplest form, one starts with an element of **A,** adds to it all other elements of **A** that are related to it by **R,** and then repeats the process until exhaustion. This is called computing the ***transitive closure,*** and can be repeated to find another equivalence class by starting with an element outside the set just found.

1.8 FUNCTIONS

A ***function*** **f** is a special binary relation over some set **A×B** where each domain value from **A** appears in exactly one tuple of the relation. For each value **a** from the domain **A** there is exactly one **b** from the range **B** such that **(a, b)** ∈ **f.** In terms of tuples this means that the only way the tuples **(a, b)** and **(a, c)** can both be in **f** is that **b=c.** There is no constraint on how many times a range value can be used in tuples of **f.**

Conceptually this type of restricted relation gives a clean "black box" concept for a function as a computing device (see Figure 1-9). Given an object **a** from its domain, the agent in the black box looks for a tuple **(a, b)** from the function **f** and returns the matching object **b** from its range. The process of finding the right result is often called ***function evaluation*** or ***function application.***

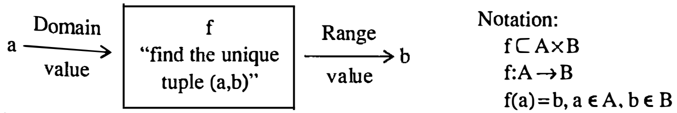


FIGURE 1-9
A function.

1.8.1 Domains and Ranges

Classical mathematical notation describes a function **f** on the set $\mathbf{A}\times\mathbf{B}$ as $\mathbf{f}:\mathbf{A}\rightarrow\mathbf{B}$, with **A** called the ***domain*** set and **B** the ***range,*** or as $\mathbf{f(a)=b}$, where $\mathbf{(a, b)} \in \mathbf{f}$. This latter notation is especially convenient for computing languages where we have the value **a** and wish the computer to find the matching value **b** via the expression **f(a)** (pronounced "**f** of **a**"). It is also a somewhat debased form of the prefix notation for relations given earlier ("**f(a, b)**").

Dropping the parentheses around the domain value gives a variant of this notation that will be of real use later on for lambda calculus. Thus **f(a)** can be written as **fa,** where the object on the left (**f**) is the function and the first object to its right (**a**) is the ***argument.*** There are examples from normal arithmetic where this notation is used, such as the unary minus and the square-root operators.

The above definition of function requires that each domain value show up in exactly one tuple. The equivalent term, ***total function,*** is often used in contexts where it is important to remember this coverage of the domain. The term ***partial function*** will refer to a functionlike relation (which we still often call a function) for which there may be some domain values that do not occur in any tuple. The reciprocal operator over real numbers is an example of this; $1/x$ exists for each domain value x except for $x=0$.

Note that a partial function **f** over $\mathbf{A}\times\mathbf{B}$ is a total function over $\mathbf{A'}\times\mathbf{B}$, where **A′** is the active domain subset of **A.**

In real life, most functions of practical interest have more than one argument, while the above definition seems limited to one. Mathematically, this is not a problem, because we are perfectly free to define a domain set as a Cartesian product of several other sets. The single "argument" can actually be a tuple of "argument components" from several separate domains, with notation being a direct map. As before, the number of argument components in the tuple is called the function's ***arity,*** and when it is of value to a discussion, it is often attached to the function name with a "/."

For example, the function "+" has a domain of $\mathbf{Z}\times\mathbf{Z}$ and a range of **Z.** It could be denoted "+/2" (in contrast to the earlier three-argument relation "+/3"). A more explicit description would be $+:\mathbf{Z}\times\mathbf{Z}\rightarrow\mathbf{Z}$. When expressed as a set it would be written as $\{((x,y),z)|z=x+y\}$. Note that this is still a set of tuples where the first element happens to be a tuple itself.

Another common function is a ***predicate,*** where the range is the set {T, F}. Such predicates are constructed from general relations of the same name defined over sets that make up the domain of the function equivalent. Thus if **R** is a relation over $\mathbf{A}\times\mathbf{B}$, the ***predicate function form*** of **R** is **r(a, b)**, $\mathbf{r}:\mathbf{A}\times\mathbf{B}\rightarrow\{\text{T, F}\}$, which in turn returns true if and only if the tuple **(a, b)** is a member of the relation **R.** As with prior notation, whether we are dealing with a function or a relation should be clear from the context.

Because functions are relations, relations are sets, and sets are valid elements of other sets, there is nothing in the above theory that prohibits us from constructing functions whose domains or ranges are essentially ***sets of functions.*** Thus, for example, we could define a function whose domain is **Z,** the integers, and whose range is the set of functions from **Z** to **Z.** Giving an argument **n** to the first function might return a function that itself adds **n** to any of its inputs. Notationally, we might write this function as $+':\mathbf{Z}\rightarrow(\mathbf{Z}\rightarrow\mathbf{Z})$, where $+'\{(\mathbf{n}, \mathbf{f_n})|\mathbf{f_n}=\{(x,x+\mathbf{n})\}\}$. Such strange creatures will become quite important to function-based computing.

Finally, any of these functions have one very important property that in a sense is a hallmark on most non-von Neumann-based computing: namely, regardless of time or the past history of prior inputs, applying a function **f,** to an object **a** will always give the same result object **f(a).** This property is called ***referential transparency*** (references to the function are transparent to time) or ***semantic invariance*** (the "meaning" of the function as a set of tuples never changes). More detailed definitions of this will be given later.

1.8.2 Properties of Functions

As with the more general binary relations, functions can be grouped into several categories. Assuming a function $\mathbf{f}:\mathbf{A}\rightarrow\mathbf{B}$, **f** is an ***injective function*** if the only way that **f(a)** can equal **f(b)** is for **a** to equal **b.** This is also called the ***one-to-one*** property, since you cannot get the same result (range value) from two different input domain values. The functions $1/x$ and $1+x$ are both injective. Integer addition is not [e.g., +(3,4) = +(6,1) but (3,4) ≠ (6,1)].

A function is a ***surjective function*** if every possible range object **b** is a member of some tuple of **f** (i.e., an "output" of at least one **a**). This property is also called ***onto,*** since it maps the domain **A** onto the entire range **B.** Addition is surjective, while squaring over the integers is not. For any integer there are multiple combinations (actually infinite) combinations of other integers whose sum equals the first one.

A function is a ***bijective function*** if it is both injective and surjective. Every element of the domain maps into exactly one element of the range, and vice versa. In a sense the function is ***renaming*** all its domain values as equivalent range values. The reciprocal function $1/x$ over the reals$-\{0\}$ is bijective.

A bijective function is also well behaved in regard to inverses. If $\mathbf{f}:\mathbf{A}\rightarrow\mathbf{B}=\{\mathbf{(a, b)}\}$, $\mathbf{f}^{-1}:\mathbf{B}\rightarrow\mathbf{A}=\{\mathbf{(b, a)}|\mathbf{(a, b)} \in \mathbf{f}\}$. There is exactly one such inverse, and it simply renames the range back into the domain.

If $\mathbf{A}=\mathbf{B}$ and the function is bijective, then it is also called an ***isomorphism.***

If the domain itself is the Cartesian product $\mathbf{A}^2$ for some set **A,** a function may have yet another set of properties in addition to the ones specified above. First, it is a ***commutative function*** if for all $(x,y) \in \mathbf{A}^2$, $\mathrm{f}(x,y) = \mathrm{f}(y,x)$. Next, it is an ***associative*** if for all $x,y,z \in \mathbf{A}$, $\mathrm{f}(x,\mathrm{f}(y,z)) = \mathrm{f}(\mathrm{f}(x,y),z)$. Finally, a function f has an ***identity element*** i if for all $x \in \mathbf{A}$, $\mathrm{f}(x,\mathrm{i}) = \mathrm{f}(\mathrm{i},x) = x$. Addition is both commutative and associative with an identity element of 0; division is neither.

Also, if $\mathrm{f}:\mathbf{A}^k \rightarrow \mathbf{A}$ is bijective, then f is *closed;* that is, we can combine any k elements of A together and get another element of A.

Finally, a function f whose domain and range overlap often have one or more common domain-range values **a** with the property that f(a)=a. Such values are called ***fixed points,*** and often play an important part in mathematical systems of interest to computing.

1.8.3 Application and Composition

The "computational model" of giving a function **f** some element **a** from its domain is defined as an ***application***; that is, the function **f** "is applied to" **a**. "Executing" the application is equivalent to finding the matching range value. In practice, this could be done by table lookup, searching the tuple set, or some more complex series of operations leading to the correct result. The mechanism is totally immaterial; in fact, we show later that all that is needed is the ability to substitute one character string into another (lambda calculus). There is no need for storage, explicit sequential instruction execution, assignment statements, variables, or any other reference from von Neumann computing.

Given this definition, an obvious direction of inquiry is what happens when we try to combine several applications together, that is, when the range value resulting from one application, say **g(a)**, is used as a domain value for some other function **f**. The result, **f(g(a))**, equals **c** from the range of **f** as long as there is some **b** in the range of **g** and the domain of **f** is such that $(\mathbf{a}, \mathbf{b}) \in \mathbf{g}$ and $(\mathbf{b}, \mathbf{c}) \in \mathbf{f}$.

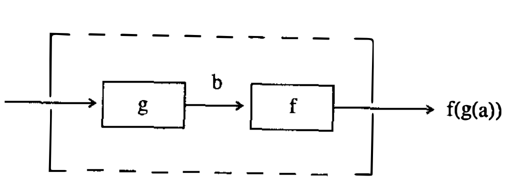
In general, we say that **g** is ***composable*** with **f** if $\mathbf{g}:\mathbf{A}\rightarrow\mathbf{B}$, $\mathbf{f}:\mathbf{C}\rightarrow\mathbf{D}$, and $\mathbf{B} \subseteq \mathbf{C}$. Any output of **g** is a valid input for **f**. Actually performing the combination is termed forming the ***composition*** of **f** and **g**.

Figure 1-10 gives the "black-box equivalent" of a composition and an example. Data travels through **g** and then into **f**. This agrees with both the above discussions and our normal intuition as formed by classical programming languages.

There is, however, an alternative view that is of more fundamental importance. Instead of concentrating on the data flow through functions, we can look at the function that results from the composition, without regard to specific inputs. In particular, the act of composing two functions can itself be treated as the ***composition function*** ∘ that accepts a pair of functions as input and delivers a function as a result. Its definition might look like this:

$$\circ : (\mathbf{C}\rightarrow\mathbf{D})\times(\mathbf{A}\rightarrow\mathbf{B})\rightarrow(\mathbf{A}\rightarrow\mathbf{D})$$

$$\circ = \{(\mathbf{a}, \mathbf{c}) \mid (\mathbf{a}, \mathbf{b}) \in (\mathbf{g} \in (\mathbf{A}\rightarrow\mathbf{B})) \text{ and } (\mathbf{b}, \mathbf{c}) \in (\mathbf{f} \in (\mathbf{C}\rightarrow\mathbf{D}))\}$$



(*a*) ∘(f,g).

x	g(x)	f(g(x))
(0,0)	0	2
(0,1)	3	1
(1,0)	2	0
(1,1)	1	3

$$\mathrm{g} = \{((0,0),0),\ ((0,1),3),\ ((1,0),2),\ ((1,1),1)\}$$
$$\mathrm{f} = \{(0,2),\ (1,3),\ (2,0),\ (3,1)\}$$

$$\mathrm{f}\circ\mathrm{g} = \{((0,0),2),\ ((0,1),1),\ ((1,0),0),\ ((1,1),3)\}$$
$$\mathrm{f}\circ\mathrm{g} = \{(x,y) \mid y = \mathrm{f}(\mathrm{g}(x))\}$$
$$\mathrm{f}\circ\mathrm{g} = \{(x,y) \mid (x,z) \in \mathrm{g} \text{ and } (z,y) \in \mathrm{f}\}$$

(*b*) An example.

FIGURE 1-10
Composition of functions.

Thus **f ∘ g** yields a function such that if **f(g(a))=c**, then **(f ∘ g)(a)=c** also. As an example, if $\mathrm{f}(x)=2\times x$ and $\mathrm{g}(x)=1+x$, then $(\mathbf{f}\circ\mathbf{g})(x)=2\times(1+x)$. The function **f ∘ g** is the function $2\times(1+x)$.

The key concept to bring from this approach to composition is that " ∘ " is itself a full-fledged function, with a domain consisting of the Cartesian product of two function sets and the range also consisting of a function set. Its inputs are functions and its result is a function. It is the first example of a special class of function-processing functions called ***combinators*** that form the basis for several function-based models of computing and have no analog whatsoever in more conventional models.

1.8.4 Curried Functions

(Stoy, 1977, p. 40; Seldin and Hindley, 1980)

The concept of a function as an object that can be processed by another function can be carried one step further. Consider, for example, a functionlike piece of code in a conventional programming language that has several input arguments, one of which is some kind of a "selection parameter" that chooses from among a set of code fragments within the program's body. Very often it would be quite convenient if we could provide this piece of code with just a value for the selection parameter and get back as a result a "tailored" subroutine to which the remaining arguments could be applied later. This could be a real performance boost if the selection process is lengthy, and the same selection value is used with multiple other argument values. Of course, in most conventional languages this is wistful thinking.

Going back to function theory, however, we find that this is not only a rational thing to discuss, but actually has a sound mathematical base developed in the 1930s and 1940s by the American mathematician

Haskell B. Curry and others. The basic idea is to invent a function “'” called the ***Curry function*** that takes as its single argument a function of n arguments (n > 1). The result of an application of this to a real function is a ***curried form*** of the function which, if given a value for the first of the n arguments returns a function which accepts the rest of the n−1 arguments, and returns the same result that the original function would have.

More formally, we define the function “'” (pronounced ***curry***) as ':(**A**×**B**→**C**)→(**A**→(**B**→**C**)). The argument here is itself a function whose domain is **A**×**B** and whose range is **C**. The range elements are also functions which, when given single arguments, return yet another function. There no constraints on any of the sets **A**, **B**, or **C**. In particular, **B**, for example, could be a Cartesian product of multiple other sets, permitting multiple curries.

In operation, ' has the property that if **f** ∈ (**A**×**B**→**C**), **a** ∈ **A**, then ('(**f**))(**a**)= $\mathbf{f}_a$ (a function from **B** to **C**) such that for any **b** ∈ **B**, $\mathbf{f}_a$(**b**) = **f**(**a**, **b**). More concisely:

$$'(\mathbf{f}) = \{(\mathbf{a},\ \mathbf{f}_a) | \mathbf{f}_a \text{ a function where for all } \mathbf{a} \text{ and } \mathbf{b},\ \mathbf{f}(\mathbf{a},\ \mathbf{b})=\mathbf{f}_a(\mathbf{b})\}$$

Consider as an example standard addition over the integers +:**Z**×**Z**→**Z**, or + = {...,((3,0),3), ((3,1),4), ((3,2),5),...}. Currying this function gives another function which if applied to a number (say, 3) produces a third function which adds 3 to its single argument. Figure 1-11 gives an example of applying ' to the function **g** of Figure 1-10.

As yet another example, consider a "generic" sorting function written in some high-order programming language that can sort an array of any kind of objects. It has two arguments, one a predicate that can be used to compare two objects of the type found in the array, and the other the array to be sorted. The desired output is a sorted version of the array.

Currying this package produces a function whose sole argument is a comparison predicate. Providing this function with a specific predicate yields a new routine that accepts one argument, an array of just that particular type, and produces as a result a sorted version of the array. Providing the curried function with a different comparison predicate would produce a sort routine tailored for another data type.

g={((0,0),0), ((0,1),3), ((1,0),2), ((1,1),1)}

'(g) = g' = {(0,{(0,1),(1,3)}), (1,{(0,2),(1,1)})}

x	g(x)	x	g'(x)	(g'(x))(0)	(g'(x))(1)
(0,0)	0	0	{(0,1),(1,3)}	1	3
(0,1)	3				
(1,0)	2	1	{(0,2),(1,1)}	2	1
(1,1)	1				

FIGURE 1-11
A curried function.

Function: log:R × R→R = log(n,x) = logarithm of x to base n

Curried version: '(log) = 'log:R→(R→R)
where 'log(b) = $\log_b$:R→R = base b log function

(('(log))(b))(a) = ('log(b))(a) = $\log_b$(a) = log(b,a)

FIGURE 1-12
Another curried function.

Figure 1-12 gives another example of the curry process.

Finally, we need not stop at currying a function only once. Consider what would happen if the generic sorting package were augmented with an argument that described how big an object in the array is, and how how many elements will be in the array. This might be useful to optimize performance. Small arrays could mean that a fast, totally in-memory routine could be used; large arrays might demand some clever sorts that use main memory as much as possible and minimize paging in and out of mass memory.

Currying once as performed before would give a routine that could sort any size array of the specified type. However, if we curried this latter routine again, the result would be a routine optimized not only for the specified type, but also for a specific array size.

To this point we have not given any indication on how such a curry function might be implemented in a real programming language. Indeed, for most conventional languages it is probably impossible. However, the lambda calculus that forms the foundation of function-based computing implements currying as a normal part of its definition and supports such concepts quite naturally.

1.9 PROBLEMS

1. Consider the problem of finding the sum and product of the k-th through k+m-th largest numbers from a set of numbers. Identify the parameters in this problem. Which ones receive input substitutions when forming an instance? Which ones receive output substitutions when a solution is reached? Characterize the problem more formally in terms of constraints on the parameters. Give (in English) a sample algorithm.
2. Consider the above problem. What happens when we switch the roles of input and output parameters. Give a simple example. (*Hint:* Are more than one solution possible?) How, if at all, is the algorithm solving such instances affected?
3. Describe (in English) the syntax and semantics of an if-then-else statement in Pascal.
4. The axiomatic rule of Figure 1-4 has a subtle problem. Consider an input substitution where *m*=1, *n*=3, *A*[1 2 3]=5 4 5, and a "valid" solution *A*[1 2 3]=1 4 5. Fix the problem.
5. For the following piece of "conventional" code, how many variable loca-

tions are there and how many different combinations of values may be bound to them during the code's execution? Comment on how often different locations are changed.

```
array A[1::200];
for I:=1 to 100 do
for J:=1 to 1 do
begin K:=I+J;
A[K]:=B[I]+C[J]
end;
```

6. Define a tuple which identifies a calender date. Include descriptions of the sets that each element may be drawn from.
7. Consider the set **I** of intervals over integers:

   $\mathbf{I} = \{(x,y)|x,y \in \mathbf{Z} \text{ and } x \leq y\}$

   Define precisely the relations **edge-touch, overlap, follow, equal,** and **contained-in** between pairs from **I.** For each relation, is it reflexive, symmetric, antisymmetric, and/or transitive?
8. Consider the following relations:

   $\mathbf{R1}$:$\{1,2,3\}^2=\{(1,1),(1,2),(1,3)\}$

   $\mathbf{R2}$: $\{1,2,3\}^2 = \{(1,1),(2,1),(3,1)\}$

   $\mathbf{R3}$: $\{1,2,3\}^2 = \{(1,3),(2,2),(3,1)\}$

   $\mathbf{R4} = \{(a,b)|a \in \textbf{Reals}, b \in \mathbf{Z}, \text{ and } b=a^2\}$

   $\mathbf{R5} = \{(a,b)|a \in \mathbf{Z}, b \in \textbf{Reals}, \text{ and } b=a^2\}$

   $\mathbf{R6} = \{(a,b)|a \in \mathbf{Z}, b \in \mathbf{Z}, \text{ and } b=a^2\}$

   $\mathbf{R7} = \{(a,b)|a \in \mathbf{Z}, a \geq 0, b \in \mathbf{Z}, \text{ and } b=a^2\}$

   $\mathbf{R8} = \{(a,b)|a \in \textbf{Reals}, b \in \mathbf{Z}, b \geq 0, \text{ and } b=a^2\}$

   Which are functions (total or partial), and for those which are functions, are they injective and/or surjective?
9. Describe $\mathbf{f} \circ \mathbf{g}$ and $\mathbf{g} \circ \mathbf{f}$, where:

   f = {**(green,sunday)**, **(blue,monday)**, **(red,friday)**}

   g = {**(sunday,red)**, **(monday,blue)**, **(tuesday,green)**, **(wednesday,blue)**, **(thursday,green)**, **(friday,green)**, **(saturday,red)**}
10. For $\mathbf{f} = \{((x,y,z),r)|$ where $x,y,z,r \in \{0, 1, 2\}$ and $r=((x\times y)+z)$modulo 3}:
    a. List **f** in a tabular format.
    b. What is the result **f1** of applying ′ to **f**? (Show as a set description).
    c. What is the result **f2** of currying it to accept a pair of arguments instead of a triple (somewhat like currying **f** twice)?
    d. What is **f1**(1)? What is result of applying (1,1) to this result?
    e. What is **f2**(1,1)?
11. Is the knapsack problem of Figure 1-1 deterministic or nondeterministic? How would you change it to get an example of the other?

CHAPTER 2

EXPRESSIONS AND THEIR NOTATION

What distinguishes one object from another is its ***value.*** Often we can write this value down directly as a character string whose meaning as a value is unambiguous, such as "3.1415." Often, however, expressing the value is more complex: We need to develop a mathematical statement which is equivalent to the desired value, but for which computation is needed to convert the statement into its simplest form.

Such a formal description of a value is termed an ***expression,*** and its notation is the subject of this chapter. For the most part, these expressions will represent functions applied to arguments. Not surprisingly, there is more than one way to write down such applications as strings of characters for interpretation by both humans and computers. While these variations may seem irrelevant, different forms in fact influence heavily the form and flavor of most of the programming languages addressed in this book.

As with the previous chapter, a reader with a good background can simply skim this chapter, paying most attention to the concept of reducing an expression to find a simpler but equivalent one, the notation used for substitutions, and to the exact variation of BNF (Backus Naur form) that we will use here for describing such expressions in real languages.

2.1 EXPRESSIONS

An ***expression*** is a formal description of an object's value. This may be anything, including a value or a function. The purpose of an expression is to describe the object, not necessarily to give an algorithm telling how to compute it. Thus an expression is neither an equation nor a program. There is no sense of assigning anything to anything, nor of executing any code.

There are two types of expressions: ***simple expressions*** and ***composite expressions***. An example of the former is a ***constant***, where the value is obvious and unchanging, for example, 2345. The latter is a construction involving the application of a function to some arguments, where in turn any of these arguments may themselves be arbitrary expressions, for example, **g**(3,**h**(4,7×16)).

An expression with no unbound variables (symbols with no values yet) in it is called a ***ground expression***, because all objects in it are rooted to real values. In contrast, expressions with unbound variables are ***nonground expressions***. For the rest of this section we assume that expressions are all ground.

Evaluating an expression corresponds to reducing it to a simpler but totally equivalent expression. Each such ***reduction*** corresponds to taking a function application and replacing it by its equivalent value (i.e., the value in the function's range which is mapped to by the argument values). Clearly, a simple expression is fully reduced and evaluates to itself.

Figure 2-1 diagrams several evaluations. Note in particular that more than one reduction sequence is possible, but all lead to the same value. Note also that this figure shows reductions graphically. A more common notation is one where two expressions are written side by side with the symbol "⇒" between. This means that the expression on the left can be reduced to the expression on the right. Thus (3+4)⇒7.

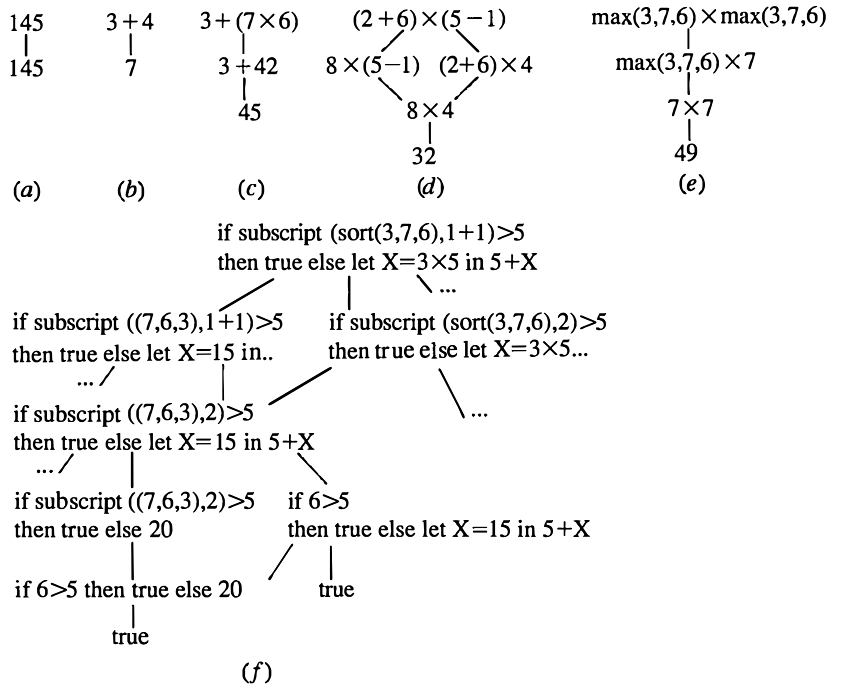

145
145
3+4
7
3+(7×6)
3+42
45
(2+6)×(5−1)
8×(5−1)
(2+6)×4
8×4
32
max(3,7,6)×max(3,7,6)
max(3,7,6)×7
7×7
49
(a)
(b)
(c)
(d)
(e)
if subscript (sort(3,7,6),1+1)>5
then true else let X=3×5 in 5+X
...
if subscript ((7,6,3),1+1)>5
then true else let X=15 in..
if subscript (sort(3,7,6),2)>5
then true else let X=3×5...
...
...
if subscript ((7,6,3),2)>5
then true else let X=15 in 5+X
...
if subscript ((7,6,3),2)>5
then true else 20
if 6>5
then true else let X=15 in 5+X
if 6>5 then true else 20
true
true
(f)

FIGURE 2-1
Some expressions and their reductions.

It is important to realize that each reduction step always returns a "totally equivalent value." Once an expression is written, no evaluation will change its "meaning"; only the complexity of its representation will change. Likewise, if the same expression is used in several places of a larger expression, each occurrence will reduce to exactly the same value.

This constancy of value over time and repetition is a hallmark of a true expression. It has what is called ***referential transparency***. To quote Stoy (1977, p. 5):

> The only thing that matters about an expression is its value, and any subexpression can be replaced by any other of equal value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs.

The expressions we will deal with here are all constructed from strings of written characters, written from left to right. With this constraint there are exactly three places where a series of characters representing a function can reside relative to those characters representing its actual arguments: to the left, in the middle, and to the right (see Figure 2-2). The following subsections address each of these. The kinds of questions that will be answered for each method include how one identifies the functions in the expression; what subexpressions should serve as the arguments to what functions; and if several applications are possible at one point, which is done first.

2.1.1 Prefix Notation

Placing the name of a function to the left of its arguments forms what is called ***prefix notation***. The arguments are to the right and, in common usage, are surrounded by "( )" and separated by ","—such as "**f**(3,67)." This will be the preferred style for most of the mathematics and many of the programming languages addressed in this book.

Notation	Position of Function	Examples
Prefix	Left of argument(s)	sqrt(16), max(3,17,4), +(4,5)
Infix	Between two arguments	4+5, 2×16
Postfix	Right of argument(s)	4 5 +, 3 17 4 max, 16 sqrt

FIGURE 2-2
Possible locations for a function in an expression.

For functions that take only one argument, the "( )" are often dropped. We thus write simply "**f***x*" for "**f**(*x*)." Samples from normal mathematics where this is common practice include unary minus and square root.

While eliminating the "( )" for this case does reduces the symbol count of an expression, it also tends to make ambiguous what expressions like "**fgh***x*" mean. Which are to be treated as functions and which as arguments? By convention, the leftmost object in any self-contained expression is the function, and the next object to the right of it is its singular argument. The purposes served by objects farther to the right cannot be determined until this first function application has been fully reduced, although they usually are other arguments. Also by convention, if the above interpretation is not the one desired, "("can be put back in to the left of each function object and ")" to the right of its single argument.

Under these conventions **fgh***x* is equivalent to ((**fg**)**h**)*x*, where **f** is a function applied to **g**. The result of this application should be a function that accepts **h** as its argument and in turn produces another function that absorbs *x*.

In contrast, the notation **f**(**g**(**h***x*)) involves applying **h** to *x*, using the result as an argument for **g**, and the result of that as an argument for **f**. This is totally different from the prior sequence of applications.

This type of notation easily extends to functions of n arguments, (i.e., arity n), n>1. Conceptually, we could treat $\mathbf{fa_1a_2a_n}$ either as an automatic currying of **f** n−1 times or simply as a function which "looks to its right" for the next **n** objects. The programming language LISP is one that normally uses the second interpretation but can be made to work like the first.

Also, our identification of a tuple like "(3,4)" as a single object allows notation like **f**(3,4) to be in agreement with both the above and standard usage. The function **f** is to be applied to a single argument which just happens to be a tuple of several components.

2.1.2 Infix Notation

Many common functions have exactly two arguments. In such cases a very natural place to write the function is between the argument objects. This is ***infix notation.***

As with prefix notation, conventions are often needed to resolve more complex expressions using infix notation. Consider, for example, "**afbgc,**" where each character is some object. This could mean to apply the infix operator **f** to **a** and **b,** and then use the result as the right argument for **g** and its other argument **c.** The reverse is also possible: We could apply **g** to **b** and **c** and then **f** to **a** and the result. In addition, however, there is a third alternative, namely, treating **b** as a function that absorbs **f** and **g** and produces a new function that in turn uses **a** and **c.**

Unlike prefix notation, the conventions for resolving such ambiguities in infix are not simple. First, all functions are given a ***precedence*** value, which is a numeric rating of its "stickiness." A higher-precedence function will be applied to its neighboring objects before a lower-precedence one. If the ordering dictated by this is not the one desired, "( )" can be used as before to force it. Figure 2-3 gives a common precedence table used in many programming languages.

This is not the end of syntatic problems for infix notation. What happens when there are "ties" in the precedence, such as in 2-5-17-8? There are six different orderings, each yielding a different result. The resolution of this usually involves other conventions such as left-to-right (FORTRAN) or right-to-left (APL).

Just as a note, in classical programming languages it is important to specify such issues even when the order seemingly does not matter, such as in **a+b+c+d.** The various objects in this expression could involve calls to code that changes the values used in the other objects. (Consider what would happen if *a, b,* and *d* are variables with initial values of 3, and **c** is a subprogram which when executed has the ***side effect*** of changing *a* and *b*'s value to 16.)

2.1.3 Postfix Notation

There is one other possible place to put a function: to the right of its arguments. This is called ***postfix notation.*** It also goes by the name ***reverse Polish notation*** after the Polish mathematician Lukasiewicz, who invented it in the 1920s. Typical notation is of the form "**abcf,**" where **f** is a function of three arguments, which in this case takes on the values **a, b,** and **c.**

More complex expressions can be constructed by throwing in more functions and objects, but as before we quickly become faced with a dilemma as to which of two or more applications should be done first, as in "**agbhcf,**" where we know that **g** and **h** are functions of one argument and **f** is a function of three. Precedence could be used, but it gets very messy when multiple functions with the same precedence are near each other. This leaves only a scan-directed ordering, left-to-right or right-to-left.

Consider right-to-left ordering first. Here we take the rightmost function (it should also be the rightmost object in the written expression),

Function	Precedence	Expression	With "( )"
**	3	3+4×5	3+(4×5)
×	2	4×5+3	(4×5)+3
/	2	3+4**2×8	3+((4**2)×8)
+	1	1+2+3	(1+2)+3 or 1+(2+3)
−	1	3/4×5	(3/4)×5 or 3×(4/5)

FIGURE 2-3
Typical infix operator precedence assignment.

and apply it to the k objects to its left, where k is the function's arity. After application we replace all k+1 objects by the result, and continue the process. This is fine until we reach the point where the results of several small function applications are needed to form the arguments for a larger one. Now we are forced into using "( )" to indicate the order, as in **(a g)(b h)c f.**

Surprisingly, the left-to-right ordering does not have this problem with the need for "( )." In this order (now called reverse Polish), the leftmost application in an expression always goes first, with processing as in Figure 2-4(*a*). Unlike any of the other notations, "( )" are never needed. It is always possible to write the desired series of function compositions without them. In the example, "**agbhcf**" **g** is applied to **a, h** then applied to **b,** and finally **f** applied to the first two results and **c.**

The primary disadvantage of reverse Polish notation is that it does not permit implicit currying of functions. If a function needs k arguments and there are k arguments to its left, it takes them. A related problem is the difficulty of telling such a system that a certain object that happens to be a function should be treated as a value for some later function.

The primary advantage of the notation in many circumstances far outweighs the above disadvantages. The semantic model is an excellent match to a ***Stack Machine*** model of a von Neumann computer. A reverse Polish expression can be translated on a one-to-one basis into a sequence of simple instructions that manipulate a stack [cf. Figure 2-4(*b*)]. If an object in an expression is a value and not a function, then its matching instruction "pushes" a copy of the value onto the top of the stack. An object representing a function translates into code that pops off the required number of values, computes the result, and pushes a copy back on the stack.

This match is so good that most real compilers for programming languages using any of the other notations will first translate from those notations into equivalent reverse Polish, and generate code from there. The ***SECD Machine*** is an example of such a machine that we will study in detail later.

```
1. Start with the leftmost object in the expression.
2. If it is not a function, move right, and repeat.
3. If it is a function of k arguments:
   a. Take the k objects to the left.
   b. Apply the function.
   c. Replace all k + 1 objects by the result.
   d. Move right and repeat step 2.
```

(*a*) Evaluation algorithm.

```
                                    ;Stack is initially empty
   3 5 + 8 6 − × SQRT               Push 3 ;Stack = 3
       ^                            Push 5 ;Stack = 3 5
 = 8 8 6 − × SQRT                   Add    ;Stack = 8
         ^                          Push 8 ;Stack = 8 8
 = 8 2 × SQRT                       Push 6 ;Stack = 8 8 6
       ^                            Sub    ;Stack = 8 2
 = 16 SQRT                          Mpy    ;Stack = 16
       ^                            Sqrt   ;Stack = 4
 = 4
```

(*b*) Sample reduction. (*c*) Equivalent Stack Machine code.

FIGURE 2-4
Postfix notation.

2.2 FUNCTION DEFINITIONS

The prior sections have gone into fair detail about the properties of functions and rough syntax for their use in expressions, but have only tangentially addressed how one actually defines a function. Specifically, this includes how one identifies the arguments that the function expects, how specific domain values get mapped into these arguments, and how those in turn are to be mapped into range values.

There are at least three major syntatic variations for defining a function:

1. As an equation
2. As an expression representing a function object
3. As a composition of more basic functions

Although all are equally powerful mathematically, the semantics they assume for the various mappings are slightly different, and when generalized into real programming languages, provide slightly different models of computation. The following subsections address each of these approaches briefly, with some discussion on the one common denominator that distinguishes any of them from a more von Neumann view. We will see detailed examples of the use of each in real languages later.

2.2.1 Functions as Equations

The most common method of describing a function is in ***equational form,*** where the name of the function is explicitly identified along with a tuple of ***formal arguments*** and some sort of expression that relates those arguments to matching range values. For example:

$$\mathbf{f}(x,y) = x+3\times y$$

This notation, or something very close to it, is what people use every day in ordinary mathematics, and often looks like features found in many programming languages. Its meaning is intuitive. If one writes something of the form "**f**(1,2)," one really wants the value $1+3\times2$, or 7. The equation itself is not an expression and has no intrinsic "value"; it is simply a statement that the symbol **f** is to be treated as a function of several arguments whose value is an expression involving the actual values given those arguments.

The assumed computational model here is that of a ***rewrite rule.*** Whenever one sees an application of the form "**f**(...)," one "rewrites" it by ***matching*** the actual arguments listed with the **f** with the formal ones in the equivalent places in the equation, ***substituting*** these values into the right-hand side of the equation, ***evaluating*** the result, and ***replacing*** the original text with this result.

2.2.2 Functions as Expressions

An alternative method of describing functions combines the tuple of formal arguments with a slight modification to the right-hand side of the equational form to give an expression representation for the function. The name of the function and the "=" can then be dropped. The remaining expression now can be used anywhere that any other object can be used, with a "meaning" that matches the original function. Function application occurs whenever an object that happens to be a function expression is placed where we would expect a function in an application. Actual arguments are then passed into the function's body as before. If there are insufficient arguments, those that can be present are still absorbed, but what comes out of the function application is a "curried" form of the original function. This curried version is again an object just like any other, and can be used accordingly.

The basis for this approach to describing functions is a mathematical theory called ***lambda calculus,*** from which most of modern mathematics can be derived. It has served as the basis for the world's second-oldest programming language, LISP, and is a primary driver for much of the function-based computing models discussed later.

Informally, the notation used by lambda calculus in this book is as follows. A function object takes a form like $(\lambda xy|x+3\times y)$, where the "λ" indicates that what is inside the "( )" is a function object, and is followed by the list of formal argument names (terminated by the "|"). Unlike the equational form, the function is given no name.

Function application is prefix form, is inherently curried, and involves only one argument at a time. The function is on the left and the argument it uses first is directly to its right. The result is often another function that replaces the original one and its argument. This new function in turn now has access to the next object to the right for another application. Thus:

$$(\lambda xy|x+3\times y)\ 1\ 2 \Rightarrow (\lambda y|1+3\times y)\ 2 \Rightarrow 1+3\times 2 \Rightarrow 7$$

2.2.3 Functions as Compositions

In equational form both the function and its formal arguments had names. In the expression form only the arguments had names. It is possible to go one step further and develop a method for describing functions and their applications that gives names to only the most basic "built-in" functions. No other functions have names, and no formal names are given to their arguments. The basic idea is to build up larger functions from smaller functions, where the tools available for doing this construction are themselves functions. The ***composition operator*** " ∘ " described earlier is one such function. Its use might result in (+ ∘ ×) expressions of the form "(+ ∘ ×)1 2," where the function (+ ∘ ×) receives its domain values much as lambda calculus expressions do.

The programming language APL has many functions of this kind built into it.

This model of computation is a direct outgrowth of a branch of mathematics called ***combinator theory,*** which itself is an offshoot of lambda calculus. Here, all functions are built up from various combinations of functions called ***combinators*** from a small predefined library. A later chapter will define several such libraries and ways of constructing arbitrary functions from them.

2.2.4 Formal Arguments and Substitutions

In any of the expression formats described earlier, the key step in their evaluation has been that of identifying an application and then of ***substitution*** of actual domain values for the formal arguments in the function's definition, regardless of how this definition was originally formed. This substitution process involves four distinct steps:

1. Pair each actual argument value with its matching formal one.
2. Find all occurrences of the formal argument in the body of the function.
3. Relace by the paired actual value.
4. Evaluate the resulting expression.

This process of pairing and substituting will occur so often in our discussions that it is worth introducing some notation. "$[\mathbf{A}/x]\mathbf{E}$" means "take the expression **E**, find all appropriate occurrences of the symbol x in **E**, and replace it by the expression **A**." Thus the evaluation of the expression "$((\lambda xy|x+3\times y)1)2$" could be described by the process

$$[2/y]([1/x](x+3\times y))$$

Each such substitution is termed a ***reduction.***

When several substitutions could, or should, go on at the same time, the natural extension "$[\mathbf{A}_1/x_1,\ldots,\mathbf{A}_n/x_n]\mathbf{E}$" will be used.

Later discussions on lambda calculus will give a very formal set of rules for when and how to apply such substitutions.

2.3 FORMAL SYNTAX NOTATION

(Lewis et al., 1978; Horowitz, 1983; Wirth, 1977)

Syntax relates to the rules and form in which symbols are assembled into language statements. It is the hook on which a programmer and a programming language both hang "meanings" about exactly what the statements will do.

Up to this point we have been somewhat loose about such syntax. The statements and notational conventions have been so small that there has been no real need for precision in their descriptions. For the most part this has been satisfactory, because we have been looking at individual concepts and not at whole chunks of languages. This will change in later chapters, and it will be important to quickly and precisely define the syntax of something unambiguously.

The following subsections address some basic concepts about syntax and a formal method of defining syntax that will be used throughout this book. This will permit short and accurate descriptions of well-formed programs. The method described here is the well-known ***Backus Naur form*** or ***BNF***. It is itself a language whose programs are descriptions of the syntax of other languages. "Executing" such programs is equivalent to computing all possible valid programs in those other languages, with no regard as to whether or not such programs actually compute anything.

2.3.1 Strings and Languages

(Bavel, 1982)

Informally, a ***language*** is a notation that permits us to describe objects. It has a ***syntax*** that consists of rules for constructing valid descriptions (which usually are ordered collections of ***symbols***), and ***semantics*** that for each piece of syntatic notation indicate how to interpret it back into a "meaning" or value.

Formally, a ***character*** is an object that represents some valid symbol in a language. For our purposes, we will assume that all characters are written, although other forms such as sound tones or light patterns are possible.

An ***alphabet*** is a finite set of characters for a particular language.

A ***string*** is an ordered sequence of characters from an alphabet. Again we will assume that such strings are written horizontally from left to right, although clearly other combinations and directions are possible.

There are three rules defining the set of all valid strings. First, the string of no characters (the ***null string***) "" is a valid string. Second, appending any character from the relevant alphabet to the right or left of a string produces a new string. Finally, the ***concatenation*** of two strings is the appending of one string to the front (left) of the other.

Some strings from the alphabet {0,1} include ∅, 0, 1, 00, 01, 10, 11, 000, 001,. . . .

Concatenation may be considered a function over strings that delivers a string result. It is often written by simply juxtaposing the two strings next to each other. Thus "HI-" "THERE" is the same as "HI-THERE."

The ***set closure*** (or ***Kleene closure***) of an alphabet **A** is denoted as **A***, and represents the set of all possible strings that can be constructed from **A.** In many ways it is similar to the ***power set*** operation defined earlier over sets. As an example, for the alphabet **A**={+,−,0,1},

$$\mathbf{A}^*=\{\varnothing,\ +,\ -,\ 0,\ 1,\ ++,\ +-,\ +0,\ -1,\ 001011,\ +0-11+000,\ldots\}$$

In this context, a ***language*** over an alphabet **A** is simply some subset of **A***. For the above example, one language might be that of signed binary numbers. A syntatically different language might be signed binary numbers where all leading 0's have been removed.

With the above definitions, we can now define a ***syntax*** or ***grammar*** as a set of rules for describing or generating a language. For the binary integer language this might be an optional "+" or "−," followed by any non-null strings of 1s and 0s.

Again note that none of this addresses the semantics of what the strings mean. That is left for other mechanisms.

2.3.2 Backus Naur Form (BNF)

The ***Backus Naur form*** or ***BNF*** is a standard language for describing the syntax of typical programming languages. While there are several forms of BNF, the one described here seems quite common, with a syntax that can be described in terms of itself.

There are several major components of a BNF description: At the lowest level, a ***terminal character*** is a character from the alphabet for the language being described. Its use in a BNF statement means that a language statement constructed according to that BNF rule must have that character embedded properly in it.

A ***metasymbol*** is a special character from the BNF alphabet itself, not the language being defined. For this book the metasymbols include {:=, <, >, |, {, }, *, $^+$, "}. In many cases the terminal alphabet includes symbols that are also in the metasymbol set; in such cases, surrounding the metasymbol by " " denotes the terminal equivalent.

At the next level is a ***nonterminal*** that represents some sublanguage in the language being described. In BNF itself, a nonterminal is the name of the sublanguage (expressed as a character string) surrounded by "⟨"

and "⟩." For example, the nonterminal ⟨number⟩ might stand for all valid character strings in the language that can represent a number.

A ***production rule*** or ***rewrite rule*** is a basic BNF statement that indicates one way of generating part of the valid sublanguage for some nonterminal. Its format is as follows:

⟨nonterminal⟩ := ⟨BNF-expression⟩

There may be multiple production rules with the same left-hand side nonterminal. Each such rule identifies a separate (but possibly overlapping) subset of strings which are part of the valid set of strings represented by the nonterminal.

Finally, a ***BNF program*** consists of a set of production rules. The order of rules within the program is immaterial; all orderings produce/define the same language.

The right-hand side of a production rule is a ***BNF expression*** indicating the appropriate subset of valid strings for this nonterminal. This expression may take any of the following forms:

- Any string of characters from the terminal alphabet. Meaning: This particular string of characters. Example: 3.1415.
- Any concatenation of nonterminals. Meaning: Any string constructable from concatenating arbitrary valid strings from each of the nonterminals. Example: ⟨sign⟩⟨pos-number⟩. Equivalent set = {1,+1,−1,...,123,+123,−123,...}.
- Any metasymbol surrounded by ".'' Meaning: The equivalent character from the language's terminal alphabet. Example: "{."
- Any BNF expression surrounded by { }. Meaning: Either the null string or any string represented by the expression within the { }. Example: {⟨sign⟩}. Equivalent set ={∅, +, −}.
- Any BNF expression surrounded by $\{\}^+$. Meaning: Anywhere from 1 to an infinite number of possibly different strings generatable by repeating the contents of the { }. Example: $\{\langle\text{digit}\rangle\}^+$. Equivalent set = {0,...,9,01,11,21,4567881,...}.
- Any BNF expression surrounded by $\{\}^*$. Meaning: Anywhere from 0 to an infinite number of possibly different strings generatable by repeating the contents of the { }. Example: $\{\langle\text{digit}\rangle\}^*$. Equivalent set = {∅, 0,..., 9, 01, 11, 21, 4567881,...}.
- Any of the above concatenated with any of the above. Meaning: Concatenate substrings arbitrarily chosen from each of the subexpressions. Example: ⟨pos-number⟩"+"⟨pos-number⟩.
- Any two BNF expressions separated by |. Meaning: Strings expressible by either subexpression are permissible. Example: ⟨letter⟩ | ⟨number⟩. Equivalent set = {A,..., Z, 0,..., 9}.

This notation is somewhat different from that found in many other texts, particularly in the use of { } instead of ( ) to surround expressions.

Alphabet = {+, −, ×, /, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Production rules:
<digit> := 0| 1|2|3|4|5|6|7|8|9
<pos-number> := $\{\text{<digit>}\}^+$
<number> := <pos-number> | −<pos-number>
<term> := <number> | <term>×<term> | <term>/<term> | (<expr>)
<expr> := <term> | <expr>+<expr> | <expr>−<expr>

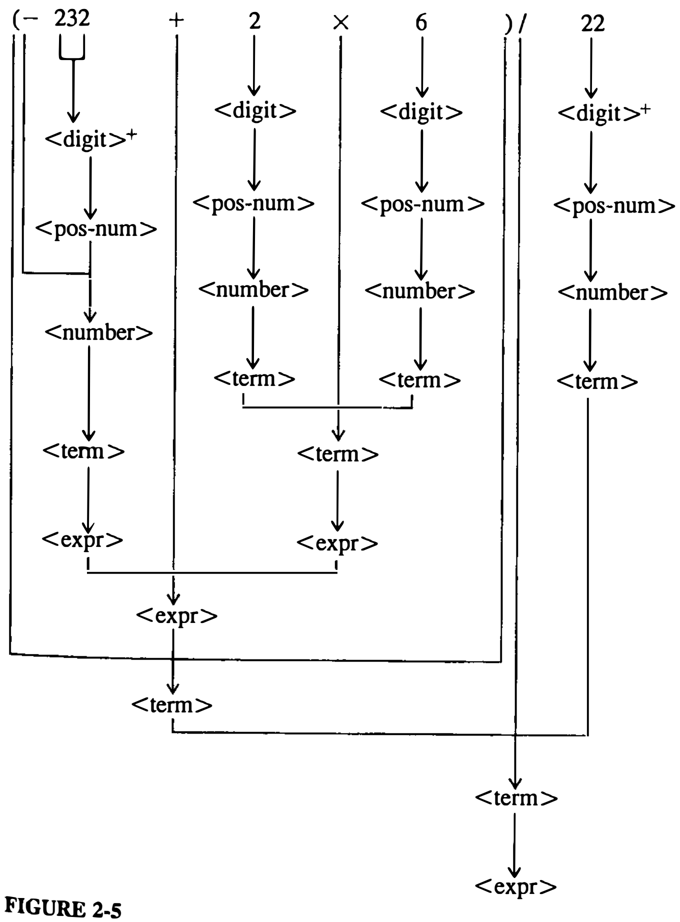


FIGURE 2-5
BNF for integer infix expressions.

In all cases the choices made here were for improved clarity; for example, the heavy use of ( ) in real languages that we will discuss later would have created longer and more confusing BNF statements, with many " to designate which ( ) are part of the language versus part of its syntax description. Although this now conflicts with the use of { } for set definitions, in most cases the distinction is obvious.

Note that the use of "|" is optional. One could get exactly the same effect by separate rules for each of the BNF expressions separated by the |, all with the same nonterminal left-hand side. Again, the order of such rules is immaterial.

Figure 2-5 diagrams a simple language that describes arithmetic expressions involving integers, the four standard operators, and parentheses. Also included is a sample derivation (called a ***parse***) from a real character string in the language's alphabet. The first rule says that a ⟨digit⟩ is any character 0 to 9. The second rule defines a ⟨pos-number⟩ as an arbitrary string of digits. The next rule defines a ⟨number⟩ as a ⟨pos-number⟩ or a leading "−" followed by a ⟨pos-number⟩. A ⟨term⟩ is either a number, two terms multiplied or divided together, or an expression surrounded by ( ). Finally, an ⟨expression⟩ is a term by itself or two expressions added or subtracted.

2.4 PROBLEMS

1. Convert the following infix expression to both prefix and reverse Polish:

   $(b+\mathrm{sqrt}(b\times b-4\times a\times c))/(2\times a)$

2. Consider a function that accepts three arguments a, b, and c, and returns a value equaling that from the above problem.
   a. Write such a definition as a rewrite rule, a lambda function, and a composition.
   b. For each of these forms, show the reductions that result when the function is applied to the three arguments 2, 6, and 8.
3. Reduce each of the following:
   - $[2/y]([3/x]\ (x+y))$
   - $[2/y,z/x]\ (x+y)$
   - $[2/y]([y/x]\ (x+y))$
   - $[2/y,y/x]\ (x+y)$
4. Describe using BNF a language of signed integers (+ is considered optional) with all unnecessary leading 0s eliminated.
5. Write a set of BNF statements that accurately describes the syntax rules of BNF itself (as defined in this chapter).
6. Along the lines of Figure 2-5, write a set of BNF statements that accurately describes the syntax rules for integer reverse Polish postfix expressions. Show a parse for "−232 2 6 × + 22 /."

CHAPTER 3

SYMBOLIC EXPRESSIONS AND ABSTRACT PROGRAMS

Accurately describing computing models requires formalisms for both syntax (the form of an expression) and semantics (its meaning). The prior chapter addressed syntax; this chapter addresses semantics. The primary notation is called ***abstract programming*** and will be used to express short "functions" which when mentally executed will return the value expected by the appropriate computational model.

To a programmer practiced in a block-structured language such as Pascal or C, this notation will be quite natural and easy to read unambiguously. Further, as will be shown later, the notation also maps directly into lambda calculus, meaning that we could make it a real programming language without much difficulty.

Before defining abstract programming, however, we will first review a notation that is useful for defining a data structure usually termed a ***list.*** Virtually any other data structure can be simulated by some form of list, meaning that it can serve as the sole data structure defining mechanism in abstract programming.

Next, this chapter will also include a discussion of a special class of functions, termed ***recursive functions,*** where each function definition is expressed in terms of itself. Nearly all the functions of interest in this book are recursive and use a relatively standardized format to express them. This chapter will introduce this notation and a matching mental process to use in reasoning about them.

Finally, part of our most frequent use of abstract programming will be in describing simple "interpreters" and "compilers" for languages of interest. To avoid the complexity of lexical scanners, parsers, tokenizers, etc., as found in real implementations, we will augment abstract programming with a set of ***abstract syntax*** functions that will do the equivalent work but without the gory detail on their implementation.

Unlike the previous two chapters, the material in this chapter is likely to be new for most readers. The only exception might be the section on s-expressions. However, given the way the material permeates the rest of this book, it is recommended that all readers review this chapter in total.

3.1 SYMBOLIC EXPRESSIONS

(Henderson, 1980; McCarthy et al., 1965)

By this point it should be obvious that the idea of a tuple as an ordered collection of other objects would be an excellent descriptive mechanism for many objects of interest to a computer scientist. This is so true that the entire structure of the programming language LISP was designed around it in the early 1960s, and variations of the mechanism used in that implementation have persisted to this day in one form or another in many important languages.

The term ***symbolic expression,*** or ***s-expression*** for short, was coined by LISP's inventor, John McCarthy, for both the notation and its implementation. Coupled with this, McCarthy also defined a set of functions which can perform useful work on objects written in the notation, and a method for describing arbitrary prefix expressions in it.

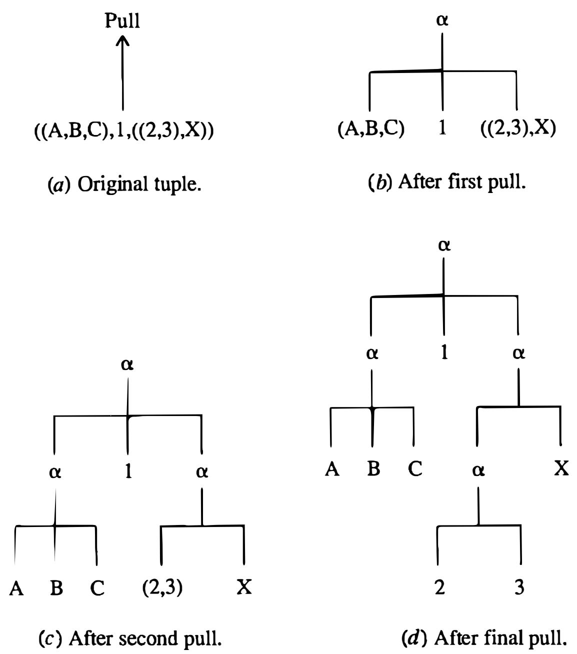
The following subsections briefly describe both the notation, a simple implementation, and the major operators. Because of their historical significance and widespread use in the computer science community, much of the original LISP terms will be used in this text, even though in some cases a more modern notation might make the exposition clearer. We will try to point out places where such confusions might occur, and augment them with extra discussion.

3.1.1 Graphical Representation—Trees

If one were to take an arbitrary tuple and repetitively "pull" up on it, leaving behind at each pull only those components that are themselves tuples, the shape very quickly takes on the two-dimensional appearance of a ***tree*** (see Figure 3-1).

Graphically, such trees are structured interconnections of ***nodes*** of three different types:

- A ***terminal node, leaf node, atomic node,*** or ***atom*** is one that has no further internal structure (namely, it is not a tuple, set, or other complex object). It typically has associated with it a constant, literal, character string, etc.



(*a*) Original tuple.

(*b*) After first pull.

(*c*) After second pull.

(*d*) After final pull.

α represents a nonterminal node.

FIGURE 3-1
Growing a tree from a tuple.

- A ***nonterminal node*** is one that does have some internal structure, and usually corresponds to a tuple. The internal structure can be represented as arcs that join it to its ***children nodes,*** that is, other nonterminals or terminals. It is a ***parent*** to these child nodes. Each child corresponds to a component or element of the tuple represented by the parent.
- A ***root node*** is a nonterminal node that has no parents.

Not all graphs in the normal mathematical sense correspond to valid objects which are expressible as tuples. Specifically, to be a tree a graph must have the following properties:

- There is either a unique root node or the tree is ***empty.***
- No nonterminal can be a parent or child of itself.
- All simple values are at the leaves.

The first rule simply guarantees that we are discussing a single object which is a tuple. Note that an object whose value is expressible as a simple constant ***is not*** a tuple, and is excluded by this rule. However, a tuple of exactly one terminal is. The second rule simply prevents loops in the tree, and the third reinforces the relationship to the tuple form.

Note that none of these rules excludes the possibility of a node having an arbitrary mixture of leaves and nonterminals as its children. This again is in agreement with the concept of a general tuple.

Although it would seem natural to draw trees from the root up, historical reasons have led most people to draw them upside down. The root node is at the top. Fanning downward from this node are arcs (like limbs) that lead to the node's children. Typically all the direct children (leaves and nonterminals) of any node are drawn on the same horizontal level. Any of the children that are themselves nonterminals fan similarly outward to their children on the next lower level. This process continues as long as there are nonterminals to expand. When complete, terminal nodes terminate all paths.

3.1.2 Historical s-Expression Notation

Our notation for tuples consists of sequences of expressions separated by "," and surrounded by "( )." This is a perfectly valid notation, and one that will be used heavily later on. However, the reader should be aware that the language that originated s-expressions, LISP, uses a slight variation, namely, one where the "," separator is replaced by one or more spaces. Thus ((A,B,C),1,((2,3),X)) would be written as ((A B C) 1 ((2 3) X)). In addition, the term ***tuple*** is renamed a ***list.***

Some modern languages, especially function-based ones, use both notations. The "," form is used when the number of elements in a particular tuple is known in advance and does not change. The " " form is used when the length of the list may vary unpredictably and dynamically.

We will follow similar guidelines (cf. Figure 3-2); context should indicate why one form was used over the other.

3.1.3 Dot Notation

A special notation, called ***dot notation,*** exists for the case where a nonterminal has exactly two components. If all nonterminals in a tree were such, the result would correspond to a ***binary tree.*** The notation involves using the infix operator "." (pronounced "dot") to specify that exactly two subtrees are to be joined together at a new nonterminal. The s-expression that is written to the right of the "." is the same one that graphically is on the right leg. The same holds for the left. "( )" surround a "." and its two arguments. Thus, if **a** and **b** are s-expressions, then **(a.b)** (pronounced "**a** dot **b**") represents a tree whose root has two arcs to **a** and **b,** respectively. Figure 3-3 diagrams several examples.

<s-expression> := <terminal> | <nonterminal>

<terminal> := "appropriate character strings"

<nonterminal> := <tuple> | <list>

<tuple> := (<s-expression> {, <s-expression>}*)

<list> := (<s-expression> {" " <s-expression>}*)

FIGURE 3-2
Partial BNF for lists and tuples.

A single BNF addition to Figure 3-2 extends s-expressions to include dot notation:

⟨nonterminal⟩ := (⟨s-expression⟩.⟨s-expression⟩)

3.1.4 Implementation of Dot Notation

One of the beauties of dot notation is the directness and simplicity of its implementation in the memory of a conventional computer. A typical approach involves dividing memory up into multiple ***cells,*** each of which corresponds to a single node and contains within itself enough storage to hold several pieces of information. First of these is a ***tag field,*** which tells what kind of node this cell represents, a nonterminal or terminal, and if the latter, what kind of value [integer, floating-point number, character string, etc.; see Figure 3-4(*b*)]. The rest of the cell's contents depends on the tag. For a terminal node it is a ***value field*** holding a standard binary representation of the node's value. For a nonterminal, there are two ***pointer fields*** whose contents can address other cells in the memory [see Figure 3-4(*a*)]. These pointers correspond to arcs to the node's children, one for the left child and one for the right.

Given the address of a nonterminal cell, a standard operator called

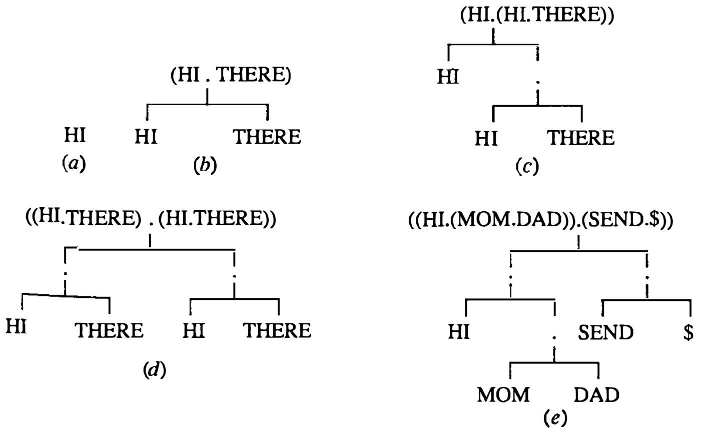
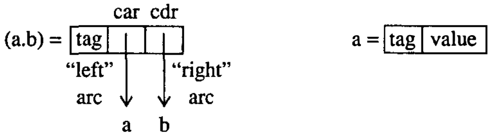


FIGURE 3-3
Sample use of dot notation.



(*a*) tag = "nonterminal."

(*b*) tag = "terminal."

FIGURE 3-4
Dot notation and its equivalent.

head, first, left, or ***car*** returns the contents of the first pointer. Another standard operator called ***tail, last, right,*** or ***cdr*** takes the same cell address and returns the other pointer. This text will use **car** and **cdr,** respectively. Thus, if **c** is the cell representing (**a**.**b**), then **car(c)** returns **a** and **cdr(c)** returns **b.**

Applying **car** or **cdr** to a terminal node is not defined.

The terms **car** and **cdr** have their origins in early implementations of LISP on the IBM 7090. This machine had a 36-bit word with two halves, the "address" half and the "data" half, each of which could hold a pointer. This was an excellent match to the cell structure described above. The 7090 instructions to access the two halves were "contents of the address register" and "contents of the data register." Thus the names.

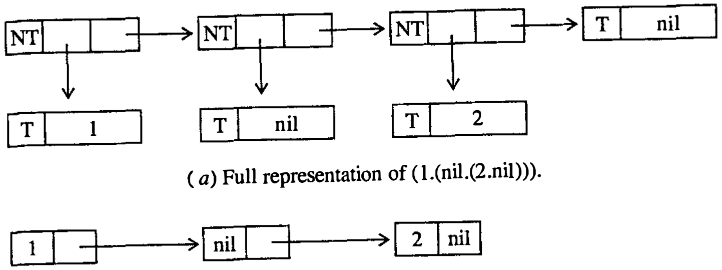
3.1.5 Shorthand Cell Drawing

Drawing s-expressions as connections of cells can often get quite tedious and consume considerable amounts of space. To reduce this we will use several shorthand drawing notations in this book.

First, in nearly all circumstances the tag value of a cell is obvious from the diagram. A cell with two subfields is a nonterminal; a cell with only one is a terminal. In the latter case the actual tag value is obvious from the cell value. Consequently, many of our diagrams will not show explicit tag fields.

Next, one kind of node not mentioned yet corresponds to the empty tree. Although it is possible to define a special tag value for this node, a more common implementation encodes any pointer that should point to an empty tree node as a special value, usually zero. This value is called ***nil,*** and is usually semantically indistinguishable from a normal pointer to a cell with the empty tree type tag. Thus we do not need to expend a unique memory cell for the nil object. In a drawing we will usually write "nil" in a cell's field when that field should point to the nil object.

Finally, very often the **car** or **cdr** of a nonterminal cell will point to another cell with a terminal tag. For brevity in notation, we will often draw such diagrams in a fashion similar to that for nil. The nonterminal cell is drawn with the value from the terminal cell in its appropriate field [see Figure 3-5(*b*)]. Even though such a diagram shows only the one nonterminal cell, it is important to remember that in reality two cells are used, with the one not shown containing a tag of type terminal and the value shown in the nonterminal's field.



(*a*) Full representation of (1.(nil.(2.nil))).

(*b*) Condensed drawing.

FIGURE 3-5
Drawing condensed s-expression cell diagrams.

3.1.6 Implementation of Lists with Dots

The dot notation is easy to implement, but it supports only two children. Most general tuples and lists have an arbitrary number of children, and it would be useful to use the cell implementation to support it.

The most common approach is quite simple:

- A zero-element list is the nil pointer discussed above.
- To construct a general n-element list, start with a nonterminal cell whose **car** part points to the first element of the list and whose **cdr** part points to the remaining (n−1)-element list. (see Figure 3-6).

The net result is n cells, the **cars** of which point to the n children and the **cdrs** of which point to the next cell, except for the last entry, which contains a nil. This last nil serves as a wall to indicate the end of the line for children of this node. Figure 3-7 diagrams a node with five children and its construction as a list of cells. Note also from this figure the now natural conversion from a pure s-expression notation to a dotted form. An s-expression of the form

$$\text{``}(\mathbf{a}_1\ \mathbf{a}_2 \ldots \mathbf{a}_n)\text{''} \quad \text{or} \quad \text{``}(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)\text{''}$$

has an equivalent dot version as

$$\text{``}(\mathbf{a}_1 \,.\, (\mathbf{a}_2 \,.\, (\ldots(\mathbf{a}_n \,.\mathbf{nil})\ldots)\text{''}$$

Note the cases for n=1 and 2:

$$(\mathbf{a}_1) = (\mathbf{a}_1 \,.\, \mathbf{nil})$$

$$(\mathbf{a}_1\ \mathbf{a}_2) = (\mathbf{a}_1, \mathbf{a}_2) = (\mathbf{a}_1 \,.\, (\mathbf{a}_2 \,.\, \mathbf{nil})) \qquad \text{(THIS IS NOT } (\mathbf{a}_1 \,.\, \mathbf{a}_2))$$

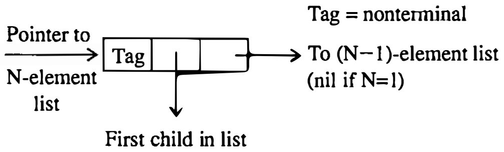


FIGURE 3-6
Linked-list equivalent of dot notation.

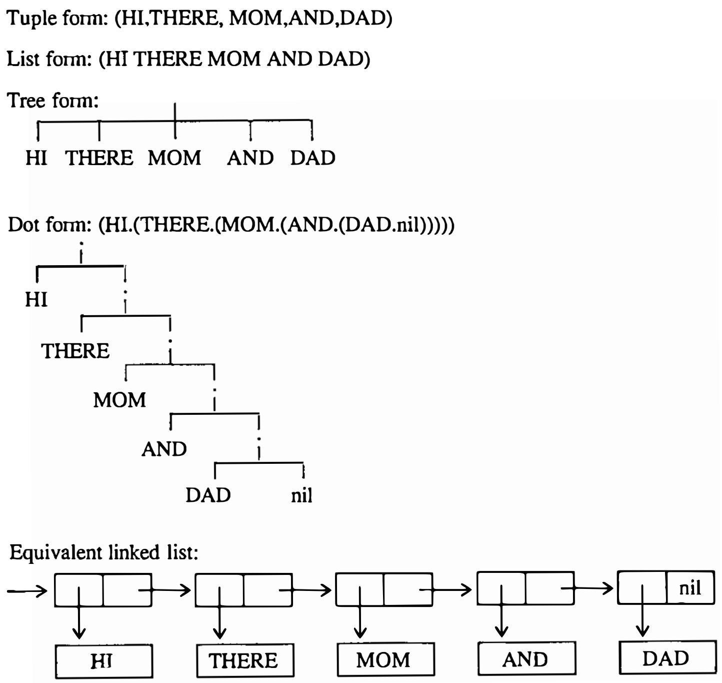

Tuple form: (HI,THERE, MOM,AND,DAD)
List form: (HI THERE MOM AND DAD)
Tree form:
HI THERE MOM AND DAD
Dot form: (HI.(THERE.(MOM.(AND.(DAD.nil)))))
HI
THERE
MOM
AND
DAD nil
Equivalent linked list:
nil
HI
THERE
MOM
AND
DAD

FIGURE 3-7
Multiple children in dot notation.

It is possible for any of the children of a node to be an empty tree itself by simply placing a nil in the **car** of the appropriate cell. Since it is the **car** pointer being used, there is no danger of confusion with the terminating nil in the final **cdr**. Thus "(( ),( ),A,( ))" in dot notation is (nil . (nil . (A . (nil . nil)))).

In all these cases the equivalent list form has significantly fewer parentheses, and thus is nearly always easier to write and read accurately. Further, the general idea can be carried forward to cases where the last **cdr** is not nil. In general, we will permit all but the leftmost dot (and the matching parenthesis) at any level of an s-expression to be removed. Thus:

$$(a_1 . (a_2 . (\ldots (a_{n-2} . (a_{n-1} . (a_n . a_{n+1}) \ldots)$$
$$=(a_1 \; a_2 \ldots a_{n-2} \; a_{n-1} . (a_n . a_{n+1}))$$

Figure 3-8(*a*) diagrams a variety of totally equivalent ways of writing the list (1 2 3); Figure 3-8(*b*) diagrams a variety of similar expressions that are nevertheless totally different.

There is nothing in the equivalence of dot notation and list notation that prevents any child of a nonterminal from being a nonterminal itself. The **car** of the appropriate cell in the first list simply points to the first cell in the chain that represents the second. Here, of course, the outermost ( ) of the inner object must be present or there is no way to identify the beginning and end of the sublist. Figure 3-9 diagrams the cell equivalent of the s-expression from Figure 3-8. Note that the shorthand form in (*b*) really corresponds to all the cells shown in (*a*).

Figure 3-10 diagrams several other examples of list notation and their equivalents.

3.1.7 Common Functions and Predicates

Of all the functions used to process s-expressions, perhaps the most common are **car** and **cdr.** When applied to an s-expression they return whatever object is indicated by the appropriate subfield. Thus **car** of ((1 2.3) 3 4) is (1 2.3); **cdr** of the same list is (3 4).

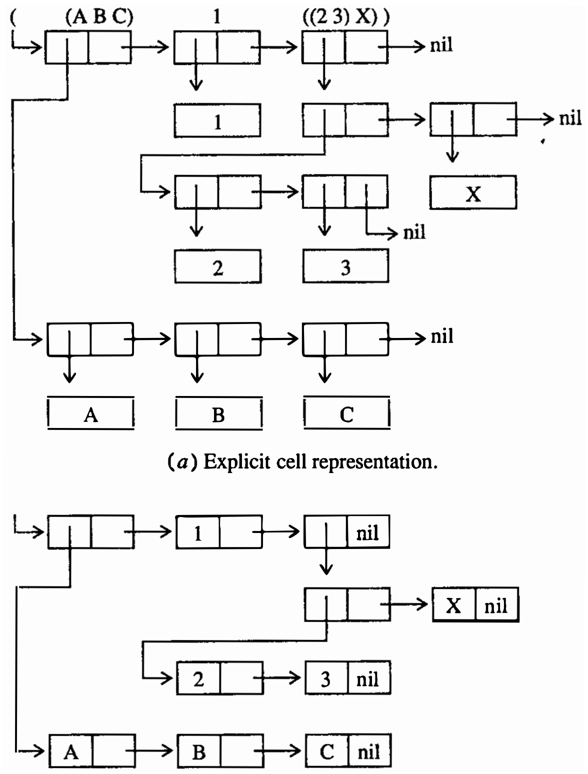
Not only are these common functions, they are often used in long strings of compositions of each other to pick out various elements of objects. For example, **car(cdr(cdr(***x***)))** picks out the third element of a list *x*; if *x* is the above list ((1 2.3) 3 4), this is 4. Likewise, **cdr(car(***y***))** picks out a list representing all but the first child of the direct children of the first child of *y*, namely, (2.3).

Because such compositions are so typical, a common convention eliminates from the written form all the middle "r(c"s and matching ")"s, leaving an initial "c," a final "r," and an intermediate string of "a"s and "d"s. Thus, **car(cdr(cdr(***x***)))** is shortened to **caddr(***x***)**, and **cdr(car(***y***))** reduces to **cdar(***y***)**.

Besides **car** and **cdr,** there are several standard functions typically found in systems that support the cell implementation of dot notation (Figure 3-11 gives some examples):

(*a*) Equivalent representations.	(*b*) Different representations.
(1 2 3) = (1, 2, 3)	(1.(2.3))
= (1.(2.(3.nil)))	((1.2).3)
= (1.(2.(3))	((1.2) 3)
= (1.(2 3))	((1 2) 3)
= (1 2.(3.nil)))	(1 (2.3))
= (1 2.(3))	(1 2 3 nil)
= (1 2 3.nil)	(1 2 (3.nil))

FIGURE 3-8
Three-element s-expressions.



(*a*) Explicit cell representation.

(*b*) Shorthand representation.

Note: A,B,C,X assumed to be terminal atomic constants

FIGURE 3-9
Pointer structure of ((A B C) 1 ((2 3) X)).

cons (*x*,*y*) constructs an s-expression (*x*.*y*). It finds an unused cell in memory and modifies it so that *x* is its **car,** *y* is its **cdr,** and its tag is nonterminal, and then returns a pointer to the cell.

list (x_1, x_2,... ,x_n) creates a list where the i-th element is x_i;=**cons** (x_1, **cons** (x_2,..., **cons** (x_n,nil)...).

length (*x*) returns a count of the number of toplevel elements in *x*; e.g., **length**(((A.B) 3 (3 4 5) 6))=4.

append (*x*,*y*) concatenates a copy of list *x* to s-expression *y*; e.g., **append**((A (B B) C), ((1.4) 2 3))=(A (B B) C (1.4) 2 3).

The general execution followed by this function involves making a copy of the list *x*, finding the last cell in the list, and replacing its **cdr** by a pointer to the root cell of *y*.

This function is also called *concatenate* or *concat.*

(A.(B.C)) = (A B.C)

(A.nil) = (A)

(A.(B.nil)) = (A B)

(A.(B.(C.nil))) = (A B C)

((A) (B.C) (D (E.F)))

= ((A) (B.C) ((D (E.F)).nil))

= ((A) ((B.C).((D (E.F)).nil)))

= ((A).(B.C).((D (E.F)).nil))

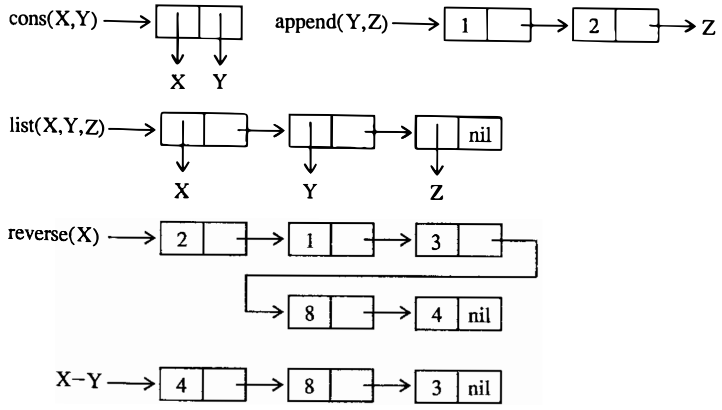
= ((A).(B.C).((D ((E.F).nil)).nil))

= ((A).((B.C).((D.((E.F).nil)).nil)))

= ((A.nil).((B.C).((D.((E.F).nil)).nil)))

FIGURE 3-10
Sample list and dot expressions.

X = (4 8 3 1 2)
Y = (1 2)
Z = (A B C)



Note: All cells shown here are different from those implementing lists X, Y, and Z.

FIGURE 3-11
Cell form of typical s-expression functions.

reverse (*x*) reverses the order of top-level children of list *x*; e.g., **reverse**((A (B C) (D E)))=((D E) (B C) A).

difference list *x*−*y* returns a list whose concatenation with *y* yields *x*. It is a partial inverse of ***append***; e.g., if *x*=(4 8 3 1 2), *y*=(3 1 2), *x*−*y*=(4 8).

Also ***append*** (*x*−*y*, *y*)=*x* and ***append*** (*x*,*y*)−*y*=*x*.

In addition to these functions there are several common predicates:

atom (x) returns true only if x is a terminal.
null (x) returns true only if x=nil.
eq (x,y) returns:

- True if x and y both terminals and $x = y$.
- False if x and y both terminals and $x \neq y$.
- Undefined otherwise (a partial function).

member (x,s) returns true if s-expression x is some first-level child of s.

3.2 ABSTRACT PROGRAMS

(Henderson, 1980)

An ***abstract program*** is a method of describing an expression built out of compositions of functions applied to arguments in a simple standardized equational form. Its beauty is that for the most part the meaning of the notation is relatively obvious to anyone who has had a moderate amount of programming experience. As such, it will be used throughout this book to describe the semantics of various constructs in other programming languages. This section gives enough of a description to permit reading and understanding these later semantic definitions. A more formal definition and mathematical basis can be found later, where the close relationship between this notation and lambda calculus is explored.

Figure 3-12 gives the major syntax for abstract programs.

The key syntatic unit is an ***expression*** which can take several forms. First is simple arithmetic and function applications on the arguments. Since this notation is primarily for human use, whatever form of expression seems most appropriate at the time will be acceptable. This includes infix, prefix, or postfix, use of standard arithmetic functions, the s-expression operators discussed earlier, and the like, all without need for more detailed definitions. A simple example might be

$x + 3 \times \textbf{length}(\textbf{cdr}(y))$.

In addition to composition of functions, an if-then-else form is also acceptable. In this form an expression following the if (usually a ***predicate***) returns either true or false when evaluated. When the overall function is applied to an argument, if this expression evaluates to true, then the expression following the then is reduced. Similarly, the else expression provides a value if the predicate is false. There is nothing prohibiting nesting of an if-then-else inside the expressions or predicate of another if-then-else. The interpretation is obvious. Figure 3-13 gives two examples.

The final bit of syntax to be described here is the ***let expression.*** This has two parts, the first of which (the definition part) looks like one or more assignment statements in a conventional programming language (coupled by the keyword ***and***), and the second (the body) which is a conventional expression. Each definition either defines a function or equates a variable name with some expression. Typically these function definitions are ones that are used inside the expression following in. Each function is defined by giving it a name and appending to it a tuple of formal argument names. These argument names are placeholders for the components of a single argument tuple that the function would accept as input.

```
<abstract-program> := <expr>

<expr> := " any normal mathematical expression"

<expr> := <if-expr> | <let-expr> | <where-expr>

<if-expr> := if <expr> then <expr> else <expr>

<let-expr> := let <definition> in <expr>
            | letrec <definition> in <expr>

<where-expr> := <expr> where <definition>
              | <expr> whererec <definition>

<definition> := <function-eqtn> | <arg> = <expr>
              | <definition> {and <definition>}*

<fcn-eqtn> := <function-name>(<arg>{,<arg>}*) = <expr>

<arg> := <identifier>

<function-name> := <identifier>
```

FIGURE 3-12
A partial BNF syntax for abstract programs.

```
reverse((A B C D)) whererec reverse(x) =
  if null(x)
  then x
  else let y=reverse(cdr(x)) in append(y,cons(car(x),nil))
```

(*a*) Reversing the list (A B C D).

```
letrec Ack(i,j) =
  if i=0 then j+1
  elseif j=0 then Ack(i-1,1)
    else Ack(i-1,Ack(i,j-1))
  in Ack(2,1)
```

(*b*) Ackerman's function.

FIGURE 3-13
Sample abstract programs.

To the right of this name and list is an "=" followed by an expression that denotes the value returned by the function when it is applied to a real argument. The "=" really means equality in the mathematical sense, and not an assignment, storage, or other memory-changing operation. Whenever one sees the left-hand side in an expression being evaluated, it can be replaced by something that is totally equal to it, namely, the right-hand side.

In the second form of a definition an identifier (without any arguments) is equated to an expression. The purpose is to simplify complex expressions where the same subexpression is used several places in some deeper expression. For example, instead of "$(3\times z+6)^2+3\times(3\times z+6)+4$," we could write "let $x=(3\times z+6)$ in $x^2+3\times x+4$."

Again, it is important to realize that any such definition is not an assignment statement in the classical sense. The "variable" in the definition part receives a value only once and has no concept of allocated storage to which values may be written or read many times. It is more like a ***macro substitution*** in a sophisticated assembler system, and actually defines an ***anonymous function*** with the let variable as a formal argument name and the in expression as the function body. The expression in the definition is then the actual argument being applied to this function. In the substitution notation defined earlier, an expression of the form "let x=**A** in **E**" is equivalent to "[**A**/x]**E**."

The range of text over which the definitions in the let part have effect is limited to the expression in the matching in part. It is permissible to nest let expressions inside the expression parts of other let expressions, creating a hierarchy of definitions very similar to the standard scoping rules in block-structured languages such as Pascal. Thus, in

let $x=3$ in let $x=x+1$ in let $x=x\times2$ in $7-x$

the only x to take on the value 3 is the one in "$x+1$," which means that the x in "$x\times2$" takes on the value 4, and the x in "$7-x$" the value 8 (yielding a value of -1 for the whole expression).

Syntatically, the ***letrec*** statement is the same as the let. Semantically the difference is that the scope for the definitions being made includes not only the expression in the in part, but also the expression in the definition part. This is a subtle but important difference and permits abstract programs to define functions which are defined in terms of themselves. Such functions are called ***recursive functions*** and are of great importance in computing. The next section will address such functions more fully, and give several detailed examples.

The ***where*** and ***wherereс*** statements are identical to let and letrec, except that the definitions come after the expressions over which the definitions apply. It often simplifies notation somewhat for humans, but has no semantic differences from the let forms.

Finally, the typical abstract program will often consist of several relatively high-level function definitions followed by a call to one of them as the ultimate expression to be evaluated. In such cases we will invoke a standard bit of shorthand by deleting the outermost let, and matching ands and in. The function definitions and final expression will be written independent of each other and on separate lines.

3.3 RECURSION

(Bavel, 1982, pp. 304–332; Burge, 1975, pp. 38–40)

A close look at many of the BNF statements and abstract programs used in this book will reveal that they often have the strange property of embedding in their defining expression an application involving themselves. Both Ackerman's function and the definition of list reversal given earlier have this property. In general, a function defined thus is termed a ***recursive function.***

While it is possible to define functions that lock themselves up in a never-ending sequence of applications of themselves, all the recursive functions of practical interest have the property that when they use themselves within their own definitions, they always do it with "simpler" arguments than they started with. Each such call of a function to itself is termed a ***recursion*** or ***recursive call.*** Eventually, these arguments become simple enough for the function to solve without recursion, and a value gets returned. Such functions are ***effectively computable***; that is, even though they are defined in terms of themselves, they can be computed mechanically in less than infinite time. This means that we can consider writing computer programs that compute them.

Recursion is an incredibly powerful tool that will be used throughout this book in discussing computational models. Indeed, most of the languages studied here have at the very heart of their semantic descriptions mechanisms that can only be expressed recursively, and usually quite compactly.

In general, a recursive definition of a function will follow a common outline somewhat like the following:

```
⟨function-name⟩(⟨arg⟩{,⟨arg⟩}*):=if ⟨basis-test⟩
                                 then ⟨basis-case⟩
                                 else ⟨recursive-rule⟩
```

The ⟨***basis test***⟩ is a predicate expression that tests if the arguments are "simple enough." If so, the ⟨***basis case***⟩ expression provides the result directly without further recursion. If not, the ⟨***recursive rule***⟩ or ***generating rule*** expression determines two things:

- How to compute the result for the current set of arguments if the answer to a simpler version of the same problem is known
- How to compute the argument values for that simpler version from the current ones

It is possible to have more than one basis test, basis case, and recursive rule by appropriate nesting of if expressions. However, from a computational viewpoint it is important that no recursive rule be invoked before being sure that none of the basis tests applies. Failure to do this could cause a computer executing the function to chase forever down unnecessary blind alleys.

In addition, it is important that all recursive rules really do generate "simpler" cases; otherwise the function's description is of no use to anyone who wants to use it as an outline for a computer program.

Perhaps the most famous recursive definition is that of the ***factorial function.*** Figure 3-14 describes the various parts of a recursive definition for it, its expression as an abstract program, and a sample "execution."

The classical implementation technique for recursive functions on conventional von Neumann computers involves use of a ***stack.*** Each time the function refers to itself, the program executing it saves on this stack a complete set of information as to where it was and what the values assigned to all formal arguments were. Such a collection of information is often called a ***frame.***

The program computes the new argument values and then jumps back to the beginning of the program code for the function. Then, if the basis test is passed, the code at the basis case computes the desired answer. Further, it checks the status of this internal stack. If the stack is empty, the answer is returned directly. If the stack is not empty, the top frame is popped back into the arguments, and the program is restarted at the point it suspended itself, but with the just-computed result now available. Again, when this code completes itself, it tests the stack before quitting. A nonempty stack triggers another popping sequence.

Failure of all basis tests leads to the recursive rule code, where a new frame is built and a new call to the function takes place.

Many recursive definitions, such as that for factorial, can be written as classical ***iteration loops*** in conventional programming languages that simply repeat the same code for a certain number of iterations, changing various variable values each time. The most common case of this occurs when the only recursive call of a function to itself occurs once at the very end of the recursive rule, and where it is not necessary to do any computation on the result of the recursive call. Such recursion is called ***tail recursion,*** and is automatically detected by many good compilers for languages that support recursion. An example of the factorial function which more clearly shows such tail recursion is

factorial(n) $= n\times(n-1)\times(n-2)\ldots\times 1$

Basis test: Does $n=0$?

Basis case: Value is 1 if $n=0$

Recursion rule: $n\times$ factorial$(n-1)$ if $n>1$

Thus, as an abstract program definition:
factorial(n) = if $n=0$ then 1 else $n\times$ factorial$(n-1)$

E.g., factorial(3) $\rightarrow 3\times$ factorial$(3-1)$
$\rightarrow 3\times(2\times$ factorial$(2-1))$
$\rightarrow 3\times(2\times(1\times($factorial$(1-1)))$
$\rightarrow 3\times(2\times(1\times 1))$
$\rightarrow 6$

FIGURE 3-14
Recursive factorial.

factorial(n) = **fact**$(n,1)$
whererec **fact**(n,z) = if $n=1$ then z else **fact**$(n-1,n\times z)$

Here, inside the function **fact,** the argument n takes on the status of a loop counter that counts down to 0. The argument z takes on the status of a variable modified each time through the loop, with the value at the end of the loop the desired result.

Not all recursively defined functions can be optimized into a conventional loop like this. Mathematicians have shown, for example, that Ackerman's function (see Figure 3-13) can only be computed recursively.

Another example of a non-tail-recursive definition is the standard one for a **Fibonacci function fib,** namely,

fib(n) = if $n<2$ then 1 else **fib**$(n-1)+$**fib**$(n-2)$

Each of the recursive calls in the recursive rule must return its value to the rule as an input to the addition, which in turn will compute the final result. This function is unlike Ackerman's, however, because it is possible to define a tail-recursive form.

3.3.1 Common Examples

Many of the functions mentioned so far, particularly those that process s-expressions, are recursive. Figure 3-15 gives typical definitions assuming that other functions such as **car, cdr, cons,** +, −, ×, **atom, null,** etc., are all "built-in" and defined separately.

3.3.2 Accumulating Parameters

In many cases the inherent computational complexity of a function stated recursively is not related to the simplicity of its definition. For example, in functions dealing with s-expressions, a common unit of "work" is the number of **cons** operations performed (this is because in real implemen-

```
From prior sections:
  factorial(x) = if x=0 then 1 else x×factorial(x−1)

  length(x) = if null(x) then 0 else 1+length(cdr(x))

  append(x,y) = if null(x) then y else let z=reverse(x) in
                   append(reverse(cdr(z)), cons(car(z),y))

  reverse(x) = if null(x) then nil
               else append(reverse(cdr(x)),cons(car(x),nil))

  member(x,s) = if null(s) then F
                  else if eq(x,car(s))
                      then T
                      else member(x,cdr(x))
Other interesting functions:
  count(x) counts the number of non-nil atoms in x.
  count(x) = if atom(x)
               then if null(x) then 0 else 1
               else count(car(x))+count(cdr(x))
  E.g., count((A (B C) D)) = 4

  equal(x,y) is true iff x and y are identical lists.
  equal(x,y) = if atom(x)
                 then if atom(y)
                      then eq(x,y)
                      else F
                 else if equal(car(x),car(y))
                      then equal(cdr(x),cdr(y))
                      else F

  union(s,t) returns a list containing every element of s or t.
  union(s,t) = if null(s) then t
               else   addelt(car(s),union(cdr(s), t))
               where addelt(x,t) = if member(x,t) then t else
               cons(x,t)
  E.g., union((A (D E) C), (A (B C) D)) = (A (D E) C (B C) D)

  intersect(s,t) returns common elements of s and t.
  intersect(s,t) = if null(s) then nil
                   else let z=intersect(cdr(s),t) in
                       if member(car(s),t)
                       then cons(car(s),z)
                       else z
  E.g., intersect((A B C D),(B C D E F)) = (B C D)
```

FIGURE 3-15
Some common recursive functions.

tations **cons** requires a relatively expensive memory allocation process to be invoked). Counting these up as a function of the size of the input often gives some valuable insight into how that function might perform if translated into a program for a conventional computer.

A good example is the definition of the **reverse** function from Figure 3-15. Figure 3-16 gives one reduction sequence for the case where the input has four elements. If we count the number of **cons**'s done by each **append,** the total is 10. A simple generalization leads to the conclusion that for an N-element list as input, this **reverse** takes (N+1)×N/2 **cons** operations to complete.

Can one do better than this? Consider, for example, a program to do the same function in a conventional programming language where loops are common [cf. Figure 3-17(*a*)]. This program does exactly one **cons** per element in the input argument, for a complexity of N. The key difference from the prior definition of **reverse** is that we have a variable *z* which "accumulates" the partial answer as it is built up.

Modifying the abstract program **reverse** to have the same lowered complexity is not only possible but a valuable lesson in a general technique for recursive function optimization. The basic idea is to define the desired function in terms of a new function which has all the same arguments (unchanged) from the original function, plus one more. This additional argument is called an ***accumulating parameter,*** and takes the place of the loop variable *z* in the prior figure. The new function is itself recursive, and in each of its recursive rules the new value to be computed by that rule is placed in the accumulating parameter position. When a basis case is reached, the current value of the accumulating parameter is returned as the value of the function application.

For **reverse** this process would yield a definition of the form of Figure 3-17(*b*). At each recursive call to **rev,** the accumulating parameter is

```
reverse((A B C D)) =

→ append( append[ append{ append(nil,D.nil),C.nil} ,B.nil],A.nil)
                                         ↑      ↑       ↑     ↑
                                         1      1       1     1
→ append( append[ append{(D),(C)} ,(B)],(A))
                   ↑
                   1  more
→ append(append[(D C),(B)],(A))
          ↑
          2  more
→ append((D C B),(A)) → (D C B A)
   ↑
   3  more
```

FIGURE 3-16
Counting **cons** operations in reverse.

```
procedure reverse(x)
  z:=nil;
  while x≠nil do
    begin
    z:=cons(car(x),z);
    x:=cdr(x);
    end;
  return z;
```

(*a*) An imperative reverse.

```
reverse(x) = rev(x,nil)
wherec rev(x,z) = if null(x) then z
                  else rev(cdr(x),cons(car(x),z))
```

(*b*) An abstract program using accumulating parameters.

FIGURE 3-17
More efficient reverses.

computed in a form that looks just like that used in the iterative loop program. The original argument takes on the role of an iteration variable which signals when the "loop" is over. Note also that the call to **rev** in the definition of **reverse** includes an initial value for this accumulating parameter.

As another example, consider a function ***sumproduct***(x), where x is a list of numbers. This function returns a dotted pair of numbers, where the **car** element is the sum of all the numbers and the **cdr** is the product. A straight recursive definition without the use of an accumulating parameter can get quite complex, in contrast to the following:

```
sumproduct(x) = sp(x,0,1)
whererec sp(x,y,z) = if null(x)
                     then cons(y,z)
                     else let p=car(x)
                     in sp(cdr(x),p+y,p×z)
```

This definition uses exactly 1 **cons**, in contrast to the approximately 2^n **cons**'s needed by an approach without an accumulating parameter.

3.3.3 The Towers of Hanoi

The ***towers of Hanoi*** is a classic example of a problem with an elegant recursive solution. In this problem, there is a board with three pegs, labeled "L," "M," and "R" (for left, middle, and right). Initially, there are n disks on peg L, all of different sizes. A property of this initial configuration is that the largest disk is on the bottom, and no disk of any size rests on top of a smaller one.

The goal of the problem is to move all disks to the R peg, so that the result is in the same order as initially. During the solution only one disk can be moved at a time, and no disk can be placed on top of a smaller one. Any of the pegs can be used at intermediate steps.

A recursive solution to this problem is the essence of simplicity. First the problem is generalized so that we can try to move a stack of disks from any of the three pegs (called the *from* peg) to any of the others (called the *to* peg), where the third peg (called the *other* peg) may have disks on it, but they are all larger than any on the first. The basis test is whether there is exactly one disk on the *from* peg. If so, the basis case simply moves it to the *to* peg, and is done. If not, the recursive rule now assumes that there are n>1 disks on the *from* peg, and they are to be moved to the *to* peg, with the *other* peg available for intermediate steps. It then performs the following steps:

1. Recursively call for the solution of moving the top n−1 disks from the *from* peg to the *other* peg, with the *to* peg free.
2. Move the now exposed largest disk from the *from* peg to the *to* peg.
3. Again recursively call for the solution of moving n−1 disks, but this time from the *other* peg to the *to* peg.

Note in both recursive calls that the problem gets "simpler," that is, the number of disks to move gets smaller than the original one (n−1 disks moved versus n). This guarantees that eventually the basis case will be invoked and the recursion process will stop.

Figure 3-18 gives an abstract program of a function that solves this problem recursively. As a sidelight it also uses s-expressions in an accu-

```
Towers-of-Hanoi(n) = reverse(toh(n,Left,Right,Middle,nil))
   whererec toh(n, from, to, other, list-of-moves) =
      if n=1 then ((from.to).list-of-moves)
      else toh(n−1, other, to, from,
         ((from.to).toh(n−1,from,other,to,list-of-moves)))
                     ↑                                      ↑
                     └─────── Recursive call ───────────────┘
```

Sample reduction sequence:

```
towers-of-hanoi(2) ⟶ reverse(toh(2,Left,Right,Middle,nil))

  ⟶ reverse(toh(1, Middle, Right, Left,
                 (Left.Right).toh(1,Left,Middle,Right,nil)))

  ⟶ reverse(toh(1, Middle, Right, Left,
                 ((Left.Right) (Left.Middle)))

  ⟶ reverse(((Middle.Right) (Left.Right) (Left.Middle)))

  ⟶ ((Left.Middle) (Left.Right) (Middle.Right))
```

FIGURE 3-18
An abstract program for solving the towers of Hanoi.

mulating parameter position to return as the result of the function a list of all the moves that had to be performed. Thus for the two-disk problem the result is ((L.M) (L.R) (M.R)), meaning that the first step involved moving a disk from L to M, then from L to R, and finally from M to R.

3.4 ABSTRACT SYNTAX

(McCarthy et al., 1965)

A common use of abstract programs in this text is to describe functions which in turn describe the semantics of other programming languages. Our approach will be through an ***abstract interpreter,*** which takes expressions or statements in this other language and performs the appropriate computational actions. By observing the actions of the interpreter on different kinds of language constructs, we can quickly understand the essential semantics of the language. This is a combination of ***interpretative semantics,*** where we model some abstract computer, and ***denotational semantics,*** where we describe the "meaning" of a construct by defining a ***semantic function*** that translates it into a real value.

To define such interpreters requires some recourse to the syntax of the other language, at least to the point of identifying what construct of the language is being discussed at what point in the interpreter function. Ideally one would like to divide the actual character strings of an actual program down into the exact syntatic units, and then move out to semantic functions. Such a process is called ***parsing,*** and would be needed if we were actually going to code such interpreters in a real program. However, given the somewhat informal nature of most of the descriptions in this text, it would be desirable to avoid the rigorousness that a full BNF analysis would offer and instead simply "invent" functions which will do whatever parsing we want without great programming effort. What we give up in detail, we gain tenfold in clarity. This is the purpose of ***abstract syntax.***

The general approach of abstract syntax is to write functions whose arguments and/or results "represent" pieces of program text that correspond to some major syntatic structure. The inner workings of such functions are never defined but should be obvious to the reader without further explanations. If the function were ever executed (which for our purposes is only performed "mentally" by the reader), the actual arguments would be real character strings from some real program.

There are three distinct kinds of activities that we will need these abstract functions to perform:

1. Test arguments (fragments of program text) to see if they fall in some syntatic category; for example, "Is x a let expression?"
2. Break a fragment of program text of known syntatic type into pieces; for example, "Get the definition text from the let expression x."

3. Put pieces back together again; for example, "Create a character string corresponding to the number 3.1415."

Functions of the first type are called ***abstract predicates,*** typically have names of the form ***is-a-zzz***(x), and return true or false depending on whether or not the actual argument for x is a piece of program text that corresponds to syntatic type ⟨zzz⟩ in the language under study. For an example, an **is-a-assignment**(x) function might accept arbitrary statements as its domain, and return true only if they are syntatically valid assignment statements.

A function which breaks down pieces of text is called an ***abstract selector*** and by convention has a name of the form ***get-zzz***(x), where x is a piece of text and ⟨zzz⟩ is the syntatic name of some component of x. Usually such functions are used in an abstract interpreter only after an earlier abstract predicate in an if-then-else expression has verified that x is of the proper type. An example might be a **get-operator** function which returns the operator from an expression of the form "⟨expr⟩⟨operator⟩⟨expr⟩."

Very often the results returned by such **get** functions are assumed to be translated into their actual meaning, rather than being left as a character string. Thus, for example, **get-number**(x) takes a piece of text which corresponds syntatically to a number in the programming language under investigation, and returns the actual number represented. This is very close to a low-level semantic function as discussed earlier.

The final type of functions used in programs employing abstract syntax are called ***abstract creators,*** and typically they have names of the form ***create-zzz***(x). In many ways these are the inverses of **get** functions, since they go from some "real" meaning back into a syntatically valid text form of type ⟨zzz⟩. An example might be a **create-term** function which takes two other terms and an operator and combines them back into a syntatically valid term in the language under study. Likewise, **create-number**(x) would create the character string representation for the number bound to x.

As a reasonably complex example, Figure 3-19 gives a set of such functions for the simple-integer expression language described in the previous chapter. Figure 3-20 then uses these functions in an actual abstract interpreter that gives the "meaning" of an integer expression by showing how such expressions are evaluated. The input to the function ***eval*** is a character string from the integer-expression language, and the output is an equivalent character string that represents the number to which the expression corresponds when fully reduced.

Note the advantages this notation gives us. First is simplicity; the entire interpreter fits on a half sheet of paper and is readily comprehensible. The second is that the interpreter is equally valid, not just for the specific language described, but also for a wide range of syntatically similar ones. For example, a change in number representation from standard

Predicate functions:

- is-a-number(x)—true if x has syntax of a number
- is-paren(x)—true if x has outer ( )
- is-a-×(x)—true if x is the × operator
- is-a-+(x), is-a-/(x),...—similar

Selector functions:

- get-value(x)—returns numeric value of x (assuming x is a number)
- get-body(x)—returns x without outermost ( )
- get-operator(x)—returns outermost ×,/,+,−
- get-left(x)—returns left term from outermost function
- get-right(x)—returns right term from outermost function

Creator functions:

- create-number(x)—converts x into syntatically valid number string
- create-infix(left, operator, right)—creates an infix expression

Note: For the above predicates and selectors, the domain of the argument x is the set of valid statements in the integer-expression language. This same set is the range for the creator functions.

FIGURE 3-19
Abstract syntax functions for integer expressions.

base-10 Arabic notation to Roman numerals would have no effect on the interpreter. We need only remember that **get-value** and **create-number** translate to and from the alternate representation. This is as it should be, since the purpose of the interpreter is to describe meaning, not form.

As an example, consider what this function would do if applied to the expression “3×(8−2).” The argument *e* receives the value “3×(8−2)” and is passed to the function **eval.** “3×(8−2)” is not a number, and it does not have surrounding parentheses, so it is taken apart by the selector functions in the let, yielding *f*=“×,” *a*=**eval**(“3”), and *b*=**eval**(“(8−2)”). “3” is a number in this language, so **eval**(“3”) returns the numeric value 3. “(8−2)” does have outside ( ), so **eval**(“(8−2)”) becomes **eval**(**get-body**(“(8−2”)), which in turn becomes **eval**(“8−2”), and which by similar processes finally reaches the appropriate leg of the cascade of ifs, where the number 2 is subtracted from the number 8. The resulting 6 goes back up to the above point, where it is finally multiplied by 3. The resulting number 18 then goes through **create-number** to return the character string “18” as the “meaning” of the expression.

```
evaluate(e): e an "integer expression"
            return its value in same syntax
 = create-number(eval(e))
   where eval(e) = if is-a-number(e)  <---- Basis test
      then get-value(e)  <--------------- Basis case
      else if is-paren(e)  <-----------------|
      then eval(get-body(e))                 |
      else let f = get-operator(e)           |
          and a = eval(get-left(e))       Doubly nested
          and b = eval(get-right(e)) in   recursion rule
          if is-a-×(f) then a×b              |
          else if is-a-/(f) then a/b         |
          else if is-a-+(f) then a+b         |
          else if is-a-−(f) then a−b <-------|
```

FIGURE 3-20
Abstract interpreter for integer expressions.

3.5 PROBLEMS

1. Find equivalent forms for the following s-expressions that minimize the number of dots.

   (((1 . (2 . (3 . nil))) . (1 . (2 . nil))) . nil)
   ((1 . 2) . (2 . 3))
   ((1 2 (3 . nil) . (4 5 6)))

2. Draw a picture of the cells needed to implement the s-expressions of Figure 3-7(*b*), showing all cells.

3. How many memory cells are needed for each of the following? Draw a picture (use condensed form).

   ((1 2)(3 4)(5 6))
   ((1 . 2)(3 . 4)(5 . 6))
   (((1 . 2) . (3 . 4))(5 . 6))
   (((1 . 2) . (3 . 4)) . (5 . 6))
   ((1 . 2) . ((3 . 4) . (5 . 6)))
   ((1 2) . ((3 4) . (5 6)))
   (( ) ( ) ( ) . (( ) ( )))

4. Find the **car, cdr, cadr, cdar,** and **caddr** (if possible) of each of the expressions from the preceding problem.

5. Evaluate Ackerman's function A(2,1) showing all reductions (see Figure 3-13).

6. Write an abstract program that returns the difference of two lists.

7. Rewrite the **sumproduct** function without using accumulating parameters.
8. Neither **union** nor **intersect** as defined in the text guarantee that the elements of their output lists are all unique, that is, there is no duplication of elements. Write some versions that guarantee uniqueness, regardless of whether or not the original input lists had duplicates in them.
9. Write a tail-recursive form of the Fibonacci function as an abstract definition. (*Hint:* Consider using two extra accumulating parameters.)
10. Expand Figure 3-20 to handle if-then-else expressions where the then and else expressions are integer expressions, and the if test is a comparison (<, =, >, ≤, ≥) between two integer expressions. Add any abstract functions you feel are necessary. Indicate which are recursive.

CHAPTER 4

LAMBDA CALCULUS

Lambda calculus is a mathematical language for describing arbitrary ***expressions*** built from the application of functions to other expressions. It originated with an attempt to find a coherent theory from which one could derive the fundamentals of mathematics, and it ended up with the capability of describing any "computable expression." Thus it is as powerful as any other notation for describing algorithms, including any conventional programming language.

The major semantic feature of lambda calculus is the way it does computation. The key (and only) operation is the application of one subexpression (treated as a function) to another (treated as its single argument) by substitution of the argument into the function's body. The result is an equivalent expression which in turn can be used as either a function or an argument. Thus ***currying*** of multiple-argument functions is not only possible it is the normal mode of execution.

Syntatically, lambda calculus is very simple. It is essentially a prefix notation where functions have no names and can be differentiated from "arguments" only by their positions in an expression. The only names given to things are the formal parameters of a function, and there are well-defined rules as to what the value of a formal parameter is after a function has been applied to an argument. These rules mirror closely the lexical scoping rules of conventional block-structured languages such as Pascal.

The following sections address the syntax and semantics of lambda calculus, with particular emphasis on the rules for formal parameters and their replacement under substitution. Also addressed is what is possible when an expression has several different internal function-argument pairs that could be applied at the same step. These discussions will lead in later chapters to opportunities for parallelism that are not found in conventional computing.

Finally, we will also discuss how to formulate out of lambda calculus many of those objects and facilities that we have grown to expect of any notation that has pretensions of being a "programming language." These include functions with multiple arguments, support for standard arithmetic, boolean truth values, logic connectives, conditional "if-then-else" operations, and recursion. The next chapter will use these mechanisms as the formal basis for abstract programming.

From a historical perspective, the interested reader is referred to Church (1951) or Rosser (1982). Other good references include Landin (1963), Burge (1975), Stoy (1977), Turner (1979b), and Henderson (1980).

In closing, the first-time reader of this chapter should not feel distressed if the material seems overwhelming. While none of the basic concepts is individually difficult, there are a lot of them, and the rational for their inclusion will not always be obvious until later chapters. A suggestion to such readers is to complete the chapter, try some of the simpler problems at the chapter's end, read the next chapter in particular, and then come back for the rest of the problems and a rereading as necessary.

4.1 SYNTAX OF LAMBDA CALCULUS

Figure 4-1 diagrams in BNF the basic syntax of lambda calculus. Although later sections will discuss minor extensions, for the most part this syntax is a complete description of the language.

The key points to remember are that lambda calculus is a language of expressions, where a function definition is a valid expression, and that a function application involves one and only one argument.

The alphabet for this language consists of the set of lowercase letters, plus the characters "(," ")," "|," and "λ" (the Greek character ***lambda***). The nonterminal "⟨expression⟩" describes all valid lambda calculus expressions, and comes in one of three forms: an application, a function, and an identifier. An ***application*** expression consists of the concatenation of two other expressions, surrounded by a set of parentheses. Syntatically, the leftmost of such expressions [the one to the immediate right of a "("] represents a function object for which the expression is to be used as its actual and sole argument in a functional evaluation.

```
<identifier> := a|b|c|d|e....

<function> := (λ<identifier>"|"<expression>)

<application> := (<expression><expression>)

<expression> := <identifier> | <function> | <application>
```

FIGURE 4-1
BNF syntax for lambda calculus.

The most basic form of a function is as a character string surrounded by "( )" and with "λ" as its leftmost character. In a sense, λ is the one and only keyword needed by the language to distinguish between a function object and other types of expressions.

The rest of a function expression consists of two parts separated by a "|." The part on the left is a single lowercase letter (an ⟨identifier⟩) representing the "name" of the single ***formal argument*** for the function. The part on the right is the body of the function and may itself be an arbitrary expression of any kind. For obvious reasons, this body usually includes copies of the formal argument's name in it.

An identifier is simply a placeholder in the body of a function for an argument that has not yet been provided. It is given a name so that it can be used several times within such an expression and still be related back to the function expression that contains it.

For pure lambda calculus we assume that all identifiers are single characters. Thus two lowercase characters written together without separating spaces is totally equivalent to the same string of characters with intervening spaces, namely, an application.

The expression "*x*" is thus an example of an identifier. "(*yx*)" is an example of an application, as is "((λ*x*|(*yx*))*a*)." The subexpression "(λ*x*|(*yx*))" is an example of a function. "(λ*y*|(λ*x*|(*y*(*yx*))))" is a nested function expression.

Finally, we will have frequent need to discuss general lambda expressions where parts of the expression can be any valid lambda expression itself. In such cases we will use uppercase single letters to represent expressions. Thus **(AB)** represents the application of two arbitrary lambda expressions **A** and **B,** where **A** is the function and **B** is its argument.

4.2 GENERAL MODEL OF COMPUTATION

In lambda calculus, what an expression "means" is equivalent to what it can reduce to after all function applications have been performed. Thus we can "understand" a lambda expression by an ***interpretative semantic model*** that describes exactly how an application is transformed into an equivalent but simpler expression. In simple terms, this model of computation might run as follows. For any lambda expression:

1. Find all possible application subexpressions in the expression (using the third syntax rule of Figure 4-1).
2. Pick one where the function object [the one just to the right of "("] is neither a simple identifier nor an application expression, but a func-

tion expression of the form "(λ*x*|E)," where **E** is some arbitrary expression.

3. Assume that the expression to the right of this function is some arbitrary expression **A**.
4. Now perform the ***substitution*** [A/*x*]E (that is, within certain limits, identify all occurrences of the identifier *x* in the expression **E** and replace them by the expression **A**).
5. Replace the entire application by the result of this substitution and loop back to step 1.

Figure 4-2 gives a simple example of one such computation.

As defined above, this model of lambda calculus leads to several natural questions. First, is there not some way of minimizing the parentheses and simplifying the notation in general? Next, given that an expression has several possible applications that could be performed, which one should be done first, and what difference does it make to the final answer? Finally, how exactly do we decide which copies of a function's formal argument in its body are replaced by the current argument? (Consider, for example, "((λ*x*|((λ*x*|(*xx*))(*xx*)))*a*)"—the first "*xx*" is not an immediate candidate for substitution by *a*.) The following sections tackle each of these questions in more detail.

Given: ((λx|(xi)) ((λz|((λq|q)z))h))

Possible first applications:
- (λx|(xi)) to ((λz|((λq|q)z))h)
- (λz|((λq|q)z)) to h
- (λq|q) to z

Pick one: Apply (λx|(xi)) to ((λz|((λq|q)z))h)
  Replace x in (xi) by ((λz|((λq|q)z))h),
  i.e., [((λz|((λq|q)z))h)/x](xi)
  → (((λz|((λq|q)z))h)i)

Remaining possible applications:
- (λz|((λq|q)z)) to h
- (λq|q) to z

Pick one: Apply (λq|q) to z
  Replace q in q by z, i.e., [z/q]q
  → (((λz|z)h)i)

Only possible application: apply (λz|z) to h
- Replace z by h, i.e., [h/z]z
- → (hi)

FIGURE 4-2
Sample computation.

4.3 STANDARD SIMPLIFICATIONS

As can be seen from Figure 4-2, the basic simplicity of a lambda expression can quickly become obscured by a multitude of nested parentheses. This section gives some standard conventions that can minimize these parentheses without loss of precision.

The first convention simplifies the expression of a series of applications. If we let E_i be either an identifier or an expression that is still enclosed by "( )," then, given an expression of the form

$$(\ldots((\mathbf{E}_1\mathbf{E}_2)\mathbf{E}_3)\ldots\mathbf{E}_n)$$

we will feel free to write it as

$$(\mathbf{E}_1\mathbf{E}_2\mathbf{E}_3\ldots\mathbf{E}_n)$$

Note that the second expression can be interpreted in only one way. The only application that can be made is that which treats $\mathbf{E}_1$ as a function and $\mathbf{E}_2$ as its argument. Only after this application is performed can we consider $\mathbf{E}_3$ as an argument to the function that results from the first application. *Under no conditions* can we consider any other $\mathbf{E}_k$ as a function object in an application until it has been absorbed by the left-to-right process and appears in a true function position.

Further, if there is no opportunity for confusion, we will also feel free to drop even the remaining set of outer "( )." This happens most often when this is the whole expression, or when the expression is the body of a function definition. Thus:

$$(\lambda x|(\lambda y|(\lambda z|\mathbf{M})))=(\lambda x|\lambda y|\lambda z|\mathbf{M})$$

The next simplification deals with nested function definitions. Given an expression of the form (λ*x*|λ*y*|λ*z*|**M**), it is permissible to cascade all the formal parameter identifiers into a single string, as in (λ*xyz*|**M**). Again, however, the meaning of this is very precise. An expression of the form

$$(\lambda xyz|\mathbf{M})\mathbf{E}_1\mathbf{E}_2\mathbf{E}_3$$

still means that the function is $(\lambda x|\lambda yz|((zy)x))=(\lambda x|(\lambda y|(\lambda z|zyx)))$, and takes only the single argument $\mathbf{E}_1$ for the formal *x*. The result will be a function $(\lambda yz|[\mathbf{E}_1/x]\mathbf{M})$, which in turn will process $\mathbf{E}_2$, and so on. (A later section will, however, address a multiple simultaneous substitution convention).

Another simplification is to use standard notations for numbers and infix arithmetic expressions whenever it makes sense. This will be justified in a later section where we give exact lambda equivalents to standard integer arithmetic.

Even when all the above simplifications have been performed, expressions still may have a significant number of parentheses. In such cases use of different kinds of parenthesis such as "{ }" or "[ ]" (paired appropriately, of course) often helps to identify the boundaries of different subexpressions.

Finally, the notation we have chosen here is not the only one found in the literature. Most of the differences are in how we identify the formal parameter names. Church's original notation deleted the "|" (as in "λ*x***A**"); other common notations use an extra set of parentheses around a tuple of identifiers (as in "(λ(*x*,*y*,*z*)**A**)"). Still others (cf. Allen, 1978) use square brackets and ";" (as in "λ[*x*;*y*;*z*]**A**"), or even like ours but with a "." instead of a "|".

The reason for this notation is threefold. First, it minimizes the number of parentheses of any kind in an expression. This is a real boon in complex expressions. Second, it provides a unique symbol to separate the list of formal argument names from the function body. Finally, the choice of "|" agrees with similar uses in other mathematical notations, as in set definitions "{*x*|*x* is...}," and as such, will be used in later logic expressions. It also avoids confusion with our use of "." in s-expressions.

4.4 IDENTIFIERS

An ***identifier*** as used in lambda calculus as a placeholder to indicate where in a function's definition a real argument should be substituted when the function is applied. It is specified by its appearance (at most once) in an argument list to the left of "|" and is used (an arbitrary number of times) to the right.

Such an identifier is often called a ***variable*** because it takes on a value at the time of function application and obeys scoping rules like variables in classical block-structured languages. However, it is not an ***imperative variable*** like those found in such languages, because there is no reference and no sense of storage allocation and, most important, at no time does a particular identifier "change its value."

A more appropriate term for an identifier in lambda calculus is as a ***binding variable,*** because during a function application a value is **bound** to it.

Each use of an identifier symbol to the right of an "|" in an expression is called an ***instance*** of that identifier. Thus in the expression (λ*xy*|*y*(*yx*)) there are two instances of *y* and one of *x*. These instances are bound to values when the function is given arguments and reduced.

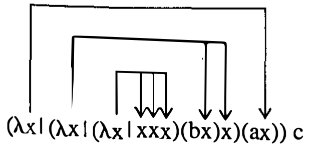
Although no single instance can have more than one value, it is possible for multiple instances of the same symbol in the same expression to have different values. This occurs when multiple function definitions are embedded within each other, all using some common identifier character in their list of formal arguments. Each of these function definitions controls a different, nonoverlapping part of the expression. Figure 4-3 gives an example of such an expression and the values taken on by different instances of the identifier *x*.

Formally, the ***scope*** of an identifier with respect to some expression is the region of the expression where instances of it will always have the same value. For lambda calculus the rules for identifier scoping are much the same as in statically scoped conventional languages. With respect to any expression, an instance of an identifier symbol may be either a ***bound instance*** or a ***free instance.*** In the former case an instance is "within the scope" (i.e., within the body) of some function object having that identifier in its argument list. As shown in Figure 4-3, the particular function object that actually binds it is the "closest" such object in terms of surrounding parentheses. In contrast, an instance is free if it is not bound, that is, it is not within the body of any function object having that symbol in its argument list.

The same identifier can have both bound and free instances in the same expression. For example, in (λ*x*|*xx*)*x* the last instance of *x* is free while the first two are bound.

Besides talking about specific instances, we can also address the properties of an identifier in general. Given an arbitrary expression **E,** we say that an identifier *x* ***occurs free*** in **E** if there is at least one free instance of *x* in **E.** More precisely, this is true if any one of the following is true:

1. **E**=*x*.
2. **E**=(λ*y*|**A**), *y*≠*x*, and *x* occurs free in **A.**
3. **E**=(**AB**) and *x* occurs free in either **A** or **B.**



→ [c/x](λx | (λx | xxx)(bx)x)(ax)

→ (λx | (λx | xxx)(bx)x)(ac)

→ [(ac)/x]((λx | xxx)(bx)x)

→ ((λx | xxx)(b(ac))(ac))

→ [(b(ac))/x](xxx)(ac)

→ (b(ac))(b(ac))(b(ac))(ac)

Note: All instances of a, b, and c are free.

FIGURE 4-3
Sample nested functions with same identifier.

Similarly, x ***occurs bound*** in **E** if there is at least one bound instance of x in **E**, that is, if one of the following is true:

1. **E**=(**AB**) and x occurs bound in either **A** or **B**.
2. **E**=($\lambda y|$**A**), y=x, and there is an instance of x in **A**.
3. **E**=($\lambda y|$**A**), $y \neq x$, and x occurs bound in **A**.

Again, an identifier can occur both free and bound in the same expression.

Figure 4-4 diagrams some sample expressions, and indicates for each which instances are free and which are bound.

4.5 SUBSTITUTION RULES

The last example of Figure 4-4 (modified from Stoy, 1977, p. 61) is of particular interest. Figure 4-5 shows two possible reduction sequences. The first (a proper one) yields 10 as a result. The second (an invalid sequence of reductions) ends up in a strange state.

The difference is in the first substitution of the second sequence. If we blindly apply the function $(\lambda q|(\lambda p|p(pq))(\lambda r|+\ p\ r))$ to its argument $(+\ p\ 4)$, we substitute the latter for q inside the function's body. However, the p in $(+\ p\ 4)$ is free in that expression, while the p's in the function body subexpression $(\lambda p|p(pq))$ are bound. A simple substitution will change this formerly free p to a bound one, changing the meaning of the expression. In a proper substitution, the free p should remain free even after the application, because its "value" should remain unchanged.

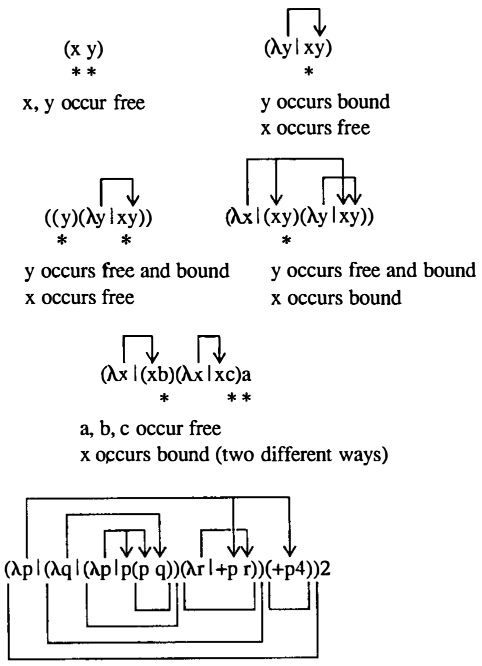


FIGURE 4-4
Sample identifier occurrence patterns.

```
Reduce: (λp|(λq|(λp|p(p q))(λr|+ p r))(+ p 4))2

Apply (λp|(λq|(λp|p(p q))(λr|+ p r))(+ p 4)) to 2 (properly)
  → (λq|(λp|p(p q))(λr|+ 2 r))(+ 2 4)
  → (λq|(λp|p(p q))(λr|+ 2 r)) 6
  → (λp|p(p 6))(λr|+ 2 r)
  → (λr|+ 2 r)((λr|+ 2 r) 6))
  → (λr|+ 2 r)(+ 2 6)
  → (λr|+ 2 r)8
  → + 2 8 → 10
```

(*a*) A valid reduction sequence.

```
Apply (λq|(λp|p(p q))(λr|+ p r)) to (+ p 4) first (Improperly)!
  → (λp|(λp|p(p (+ p 4)))(λr|+ p r))2
  → (λp|(λr|+ p r)((λr|+ p r) (+(λr|+ p r) 4))(λr|+ p r))2
  → (λr|+ 2 r)((λr|+ 2 r) (+(λr|+ 2 r) 4))(λr|+ 2 r)
  → + 2 ((λr|+ 2 r) (+(λr|+ 2 r) 4))(λr|+ 2 r)
  → + 2 (+ 2 (+(λr|+ 2 r) 4))(λr|+ 2 r)
  → ... ??
```

(*b*) An improper substitution.

FIGURE 4-5
Example of a potential substitution problem.

The solution to this lies in a more careful definition of the rules for substitution. Given an expression of the form ($\lambda x|$**E**)**A**, where **E** and **A** are arbitrary expressions, the evaluation of the expression involves the ***substitution*** of the expression **A** for all appropriate free occurrences of the identifier x in the expression **E**, denoted [**A**/x]**E**. For discussion purposes we call the expression resulting from [**A**/x]**E** as **E'**.

Formally, **E'** = [**A**/x]**E** can be computed from the rules given in Figure 4-6. Basically, these rules follow the rules for free occurrences of x in **E**. Such occurrences of x are replaced by **A**; any bound occurrences of x and any occurrences of any other symbols are left unchanged.

Rule 1 is the simplest case, where the expression **E** is a single identifier symbol. If the symbol matches the symbol being substituted for (i.e., x), it is replaced; if not, the expression is unchanged. Rule 2 handles the case where **E** is an application. Logically enough, the resulting expression is an application constructed from the substitution of **A** for x in both parts of the original expression.

Rule 3 handles the final possibility for **E**, namely, when **E** is itself a function object. There are three subcases here, based on the symbol used for the function's formal argument. The simplest case is where the formal

```
For (λx|E)A → [A/x]E → E′
Rules for substitution [A/x]E are:

1. If E is an identifier y
     then if y=x, then E′ = A
     else (in this case, y≠x) E′ = E.
     Examples:
     • (λx|x)M → [M/x]x → M
     • (λx|y)M → [M/x]y → y

2. If E = (BC) for some expressions B and C
     then E′ = (([A/x]B)([A/x]C)).
     Example: (λx|xyx)M → (([M/x](xy))([M/x]x)) → MyM

3. If E = (λy|C) and C some expression

   a. and y=x, then E′ = E.
      Example: (λx|(λx|xN))M → (λx|xN)

   b. and y≠x and y does not occur free in A
      then E′ = (λy|[A/x]C)
      Example: (λx|(λy|xyN))M
              → [M/x](λy|xyN))
              → (λy|[M/x](xyN)))
              → (λyMyN))
              → (λ|MyN)

   c. (RENAMING RULE) otherwise (i.e., y occurs free in A)
      E′ = (λz|[A/x]([z/y]C)), where z is a new symbol never
      used before (or at least not free in A)
      Example: (λx|(λy|xyN))(ay)
              → [(ay)/x](λy|xyN) (note y occurs free in (ay))
              → (λz|[(ay)/x][z/y](xyN))
              → (λz|[(ay)/x](xzN))
              → (λz|(ay)zN)
```

FIGURE 4-6
Substitution rules for lambda calculus.

parameter is the same as *x*; by our previous definition there can be no free occurrences of *x* in the function (they are all bound to **E**'s internal binding variables), and thus nothing to substitute for. The result of the substitution is **E** itself.

Rules 3b and 3c are, however, more complex. Here the formal parameter in the function is different from *x*, so any free occurrences of *x* in the body of **C** are also free in the total function **C**, and thus must be replaced by **A**. The complication, as brought out in Figure 4-5, occurs when the expression replacing the *x*'s (i.e., **A**) has within itself free occurrences of the *y*, the formal parameter of the function. Rule 3b covers the case when this does not occur; the substitution goes through without difficulty. If, however, **A** has free occurrences of *y* in it, then a blind substitution into the body of the function will end up changing those instances of *x* in **A** from free to bound, radically changing the value of the expression.

The solution is to change the formal parameter of the function and all free occurrences of that symbol in the function's body. The symbol we change it to must be one that does not conflict with any symbols already in use. This can be done if we avoid any symbol that has free occurrences in either **C** or **A**. In Figure 4-6 we assume that this symbol is *z*.

Note further that the substitutions must be done in the indicated order. First we replace all *y*'s in **C** by *z* ([*z*/*y*]**C**)—this prevents conflicts with the *y*'s in **A**. Then we replace all free *x*'s in the result by **A**—now the free *y*'s (and just those *y*'s) remain free.

This final rule is called the ***renaming rule*** for obvious reasons.

Figure 4-7 diagrams a proper substitution version of Figure 4-5(*b*). The answer is again 10, as in (*a*). Figure 4-8 diagrams another sample application where if renaming is not applied, the result changes.

4.6 CONVERSION RULES, REDUCTION, AND NORMAL ORDER

(Stoy, 1977, pp. 64–67)

The basic lambda calculus execution mechanism "replaces" one expression by another by substituting an argument into a function. Normally the expression after the substitution is simpler than the original, thus the term ***reduction*** is used to refer to one function application. In any case, however, the semantics of lambda calculus guarantees that the "meaning" of the before and after expressions is the same—they both represent the same object.

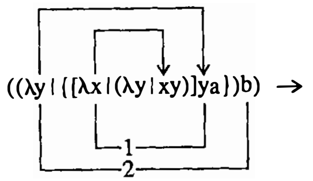
In this context it becomes relevant to discuss under what conditions we know that two separate expressions are in fact the same, and what kinds of standard forms there are that simplify such comparisons.

Not unexpectantly, we say that two expressions **A** and **B** are the same if there is some formal way of converting from one to the other via

```
Reduce: (λp|(λq|(λp|p(p q))(λr|+ p r))(+ p 4))2

Apply (λq|(λp|p(p q))(λr|+ p r)) to (+ p 4) first (Properly)!
    → (λp|(λz|z(z (+ p 4)))(λr|+ p r))2
    → (λz|z(z (+ 2 4)))(λr|+ 2 r)
    → (λz|z(z 6))(λr|+ 2 r)
    → (λr|+ 2 r)((λr|+ 2 r) 6))
    → + 2 ((λr|+ 2 r) 6)
    → + 2 (+ 2 6) → 10
```

FIGURE 4-7
A proper reduction sequence.



- If Application 2 done first,

```
→ [b/y]{[λx | (λy | xy)]ya} = {[λx | (λy | xy)]ba}
→ ([b/x](λy | xy))a → (λy | by)a → ba
```

- If Application 1 done first AND NO RENAMING (an error),

```
→ ((λy | {([y/x](λy | xy))a})b) = ((λy | {(λy | yy)a})b)
                              "name conflict"
... → aa
```

- If Application 1 done first AND renaming used correctly:

```
→ ((λy | {([y/x](λy | xy))a})b) → ((λy | {(λz | [y/x][z/y](xy))a})b)

       Change name to "z"

→ ((λy | {(λz | [y/x](xz))a})b) → ((λy | {(λz | (yz))a})b)
→ ((λy | ya)b) → ba
```

FIGURE 4-8
Sample renaming substitution.

a series of reductions. Notationally, we write "**A** → **B**" if **A** "reduces to" **B** in one or more steps. Figure 4-9 lists formally the three rules governing these valid reductions. Figure 4-10 gives a variety of sample equivalences.

The ***alpha conversion*** rule corresponds to a simple and safe "renaming" of formal identifiers. Two expressions are the same if they differ only in the symbol names they give to their function objects' formal parameters.

```
Alpha conversion (renaming):
  If y not free in E then (λx|E) → (λy|[y/x]E)
  Example: (λx|xzx) → (λy|yzy)

Beta conversion (application):
  (λx|E)A → [A/x]E (with renaming in E)

Eta conversion (optimization):
  If x not free in E then (λx|Ex) → E
```

FIGURE 4-9
Reduction rules for expression equivalence.

```
(λx|+ x 4)3 → [3/x](+ x 4) → (+ 3 4) → 7

(λx|+ x 4)(+ 2 z) → [(+ 2 z)/x](+ x 4) → (+(+ 2 z)4)

(λx|+ x y) 3 → (+ 3 y)

(λx|λy|+ x y) 3 → [3/x](λy|+ x y) → (λy|+ 3 y)

(λx|λx|+ x 3) 4 → [4/x](λx|+ x 3) → (λx|+ x 3)

(λx|λy|+ y 3) 4 → [4/x](λy|+ y 3) → (λy|+ y 3)

(λx|λy|+ y x)(+ z 4) → λy|[(+ z 4)/x](+ y x)
  → (λy|+ y(+ z 4))

(λx|λy|+ y x)(+ y 4) → λz|[(+ y 4)/x][z/y](+ y x)
  → λz|(+z(+ y 4))

(λq|((λp|p(pq))(λr|+ p r))) (+ p 4)
  → [(+ p 4)/q]((λp|p(pq))(λr|+ p r))
  → ([(+ p 4)/q](λp|p(pq))) ([(+ p 4)/q](λr|+ p r))      RULE 2
  → ([(+ p 4)/q](λp|p(pq))) (λr|+ p r)      RULE 3b
  → {λs|[(+ p 4)/q][s/p] (p(pq))}(λr|+ p r)      RULE 3c
  → (λs|[(+ p 4)/q](s(sq))) (λr|+ p r)      AFTER RENAME
  → (λs|s(s(+ p 4)))(λr|+ p r)

(λx|xx)(λx|xx) → [(λx|xx)/x](xx)
  → (λx| xx)(λx| xx)

(λx|xxx)(λx|xxx)
  → (λx|xxx)(λx|xxx)(λx|xxx)
  → (λx|xxx)(λx|xxx)(λx|xxx)(λx|xxx)
  → . . . . an infinitely long concatenation of (λx|xxx)
```

FIGURE 4-10
Sample equivalent expressions.

The ***beta conversion*** rule matches normal lambda calculus function applications. Two expressions are the same if one represents the result of performing some function application found in the other.

Finally, ***eta conversion*** corresponds to a simple optimization of a function application that occurs quite often. It is basically a special case of beta conversion. To show that it is correct, consider any expression of the form

(λ*x*|**E***x*)**A** with no free *x*'s in **E**

With beta conversion (simple application), this expression reduces to

[**A**/*x*](**E***x*) → **EA,** since there are no free *x*'s in **E.**

Using eta conversion first,

$$(\lambda x|\mathbf{E}x)\mathbf{A} \rightarrow \mathbf{EA}$$

which is the same as above for any **E** or **A.**

With these rules there are an infinite number of expressions that might be equivalent to some given expression. In fact, if we used eta conversion in reverse, we could take an arbitrary expression **E** and construct an arbitrarily long equivalent expression of the form

$$\mathbf{E}_\mathrm{n} \rightarrow (\lambda x_\mathrm{n}|(\lambda x_{\mathrm{n}-1}|\ (\ldots(\lambda x_1|\mathbf{E}x_1)\ x_2)\ldots)x_\mathrm{n}) \text{ for any n}$$

where no x_k appears free in **E.** Note that because of the ")" between x_k and $x_{\mathrm{k}+1}$, this is *not* the same as $(\lambda x_1\ldots x_\mathrm{n}|\ (\ldots(\mathbf{E}x_1)x_2)\ldots x_\mathrm{n})\ x_2)\ldots x_\mathrm{n})$.

Given this, it would be convenient to pick one form of an expression as the "simplest" and give rules on how to find it. Given the semantics of lambda calculus, it would seem that this "simplest" form occurs when we can no longer apply beta or eta conversions to it; there are no more applications that can be performed, and no more optimizations. This is called reducing an expression to ***normal order,*** after which the only conversions possible are the essentially infinite number of alpha "renamings" that might go on.

All the examples of Figure 4-10 are shown in their normal order except for the last. That is an example of an expression with no normal-order equivalent; beta reductions can be performed on it forever. Also note that it is possible for intermediate expressions to exceed in size either the original expression or the final normal-order form.

4.7 THE CHURCH-ROSSER THEOREM

(Stoy, 1977, p. 67)

Many expressions have more than one beta or eta conversion that could be performed at any one time, giving rise to several possible different sequences of conversions. Does it matter which sequence one chooses? Is there one or several normal-form equivalents of an expression? Is there any preferred sequence?

The answers to these questions came early in the development of lambda calculus. In 1936 Alonzo Church and J. R. Rosser proved two theorems of historic importance. The first, the ***Church-Rosser Theorem I*** or ***CRT,*** states that if some expression **A,** through two different conversion sequences, can reduce to two different expressions **B** and **C,** then there is always some other expression **D** such that both **B** and **C** can reduce to it (see Figure 4-11). The significance of this is that, given an expression, you can never get two nonequivalent expressions by applying different sequences of conversions to it.

For any expression A:

If:

$$\begin{tikzcd} & B \\ A \arrow[ur] \arrow[dr] & \\ & C \end{tikzcd}$$

Then there is a D such that:

$$\begin{tikzcd} B \arrow[dr] & \\ & D \\ C \arrow[ur] & \end{tikzcd}$$

FIGURE 4-11
The Church-Rosser theorem.

The second theorem, named the ***Church-Rosser Theorem II,*** states that if the expression **A** reduces to **B** by some sequence, and **B** is in normal order (no more beta or eta reductions are possible), then we can get from **A** to **B** by doing the ***leftmost reduction*** at each step. Leftmost in this case refers to the position of the start of the application in the text string representing the expression. Also, reductions in this case include only beta and eta, since any number of alpha conversions could be applied without changing the real structure of the expression.

This theorem is significant because it gives a concrete algorithm to convert an expression to normal form. We simply look for the expression that is in a function position whose text string starts leftmost and that is amenable to a beta or eta reduction, and reduce it.

An important follow-on lemma from the CRT is that within a change of identifier names (i.e., within an alpha conversion) there is only one normal-form equivalent of any expression, if one exists. When combined with the second theorem, this guarantees that not only is there at most one normal form of an expression, but also we have a guaranteed sequence of reductions that will find it, if it exists.

4.8 ORDER OF EVALUATION

There are often several applications possible in an expression at any step in its reduction to normal order (cf. Figure 4-12). The two CRTs guarantee that two different choices cannot give two different final normal-order expressions, and that choosing the leftmost is a guaranteed good choice. However, one could also ask about other sequences, and what other kinds of differences could result.

Following examples use "{ }" to surround leftmost reducible function:

```
({λy|((λx|(λy|xy))ya)} b)
  → ({λx|(λy|xy)}ba)
  → ({λy|by}a)
  → ba
```

FIGURE 4-12
Normal-order reduction.

To answer this we first define two approaches to choosing which application in an expression to reduce at each step. A ***normal-order reduction*** follows the CRT results; namely, it always locates the leftmost function that is involved in an application, and substitutes unchanged copies of the argument expression into the function's body. No reductions are performed on the argument until after this substitution.

In contrast, an ***applicative-order reduction*** reduces the argument (and potentially the body of the function) separately and completely before doing the function application and its required substitution. When the function body is reduced before the application, whether it or the argument is reduced first is immaterial.

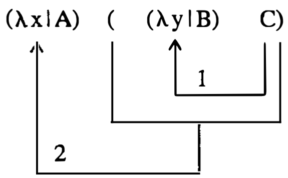
Figure 4-13 gives a generic expression where either applicative- or normal-order evaluation is possible. Normal-order reduction substitutes ((λy|**B**)**C**) for *x* into ((λz|**A**)*x*). Applicative order reduces the argument to [**C**/y]**B** first (and potentially reducing **A**) before replacing *x* in **A.**

Now consider the example in Figure 4-14. The normal-order sequence terminates with a normal-order expression, but it does so only after evaluating the second argument ((λ*z*|*z*)*a*) twice. There are many similar cases where an argument is evaluated not twice but an arbitrary number of times, each time yielding exactly the same result. This is an inefficient way to do computation.

In contrast, look at what happens when we choose an applicative-order reduction. The first argument is reduced before performing the function application. Thus, the work involved is done only once. However, unless we stop doing the leftmost application at some time, the reduction of this argument never ends, we never get a reduced result, and we never even get to the second argument.

Figure 4-14(*c*) diagrams an optimal order of evaluation where the first step is a normal order and the second an applicative order. The infinite loop is prevented, as is the duplicate evaluation of the second.

This example is indicative of the general results about these two evaluation orders. Normal-order evaluation guarantees a reduced expression if one exists, but at the expense of potential duplication of evaluations. Applicative order will not give a different answer. If it terminates, it will yield an expression equivalent to that from the normal-order sequence, and it will do so without duplicate computation. Further, reducing the function body and the argument can be done in parallel, offering the possibility of even more time-efficient evaluation. However, it is possible that the applicative-order reduction will never terminate, but loop forever.



Normal reduction: Do 2 first, then 1

Applicative reduction: Do 1 first

FIGURE 4-13
Multiple applications.

```
Expression: (λx|{λw|(λy|wyw)b}) ((λx|xxx)(λx|xxx)) ((λz|z)a)
  → ([((λx|xxx)(λx|xxx))/x]({λw|(λy|wyw)b})) ((λz|z)a)
  → ({λw|(λy|wyw)b}) ((λz|z)a)
  → (λy|((λz|z)a)y((λz|z)a))b
  → ((λz|z)a)b((λz|z)a))      first redundant evaluation
  → (ab((λz|z)a))      second redundant evaluation
  → aba
```

(*a*) Normal order reduction.

```
  → (λx|[b/w](wyw)) ([(λx|xxx)/x](xxx)) ((λz|z)a)
  → (λx|yyb) ((λx|xxx)(λx|xxx)(λx|xxx)) ((λz|z)a)
  → (λx|yyb) ([(λx|xxx)/x](xxx)(λx|xxx)) ((λz|z)a)
  → (λx|yyb) ((λx|xxx)(λx|xxx)(λx|xxx)(λx|xxx)) ((λz|z)a)
  → . . . . repeat reductions forever. . . .
```

(*b*) Applicative-order reduction.

```
  → ([((λx|xxx)(λx|xxx))/x]({λw|(λy|wyw)b})) ((λz|z)a)      normal
  → ({λw|(λy|wyw)b}) ((λz|z)a)
  → ({λw|(λy|wyw)b}) ([a/z]z)      applicative
  → ({λw|(λy|wyw)b}) a      normal
  → (λy|aya)b      normal
  → aba
```

(*c*) A mixed order of evaluation.

FIGURE 4-14
Different reduction orders.

As demonstrated in Figure 4-14(*c*), if we can somehow know which order to use at each step, it is possible to avoid both problems. Later chapters will introduce such techniques as used in real languages.

4.9 MULTIPLE ARGUMENTS, CURRYING, AND NAMING FUNCTIONS

In conventional programming languages we very frequently describe functions or procedures with multiple arguments. The normal semantics for applying such functions involves a "simultaneous" substitution of all the actual arguments for that particular application into their formal arguments. Although most languages do not support it, ***currying*** a function corresponds to cases where there are not enough actual arguments available to match all the formal ones. The result of such an application should be a function which will accept the rest of the arguments when they are available.

Up to this point our formal definition of lambda calculus has not supported multiple actual arguments in a single application, even though we invented a simplified notation that shows a single λ with multiple formal arguments, as in $(\lambda xyz|\mathbf{E})$. In a sense this notation is "syntatic sugar," since, given multiple real arguments, the formal way to interpret something like $(\lambda xyz|\mathbf{E})\mathbf{ABC}$ is one argument at a time, namely:

$$\begin{aligned}
&(\lambda xyz|\mathbf{E})\mathbf{ABC}\\
&\rightarrow (\lambda x|(\lambda y|(\lambda z|\mathbf{E})))\mathbf{ABC}\\
&\rightarrow ([\mathbf{A}/x](\lambda y|(\lambda z|\mathbf{E}))\mathbf{BC}\\
&\rightarrow \{[\mathbf{B}/y]([\mathbf{A}/x](\lambda z|\mathbf{E})\}\mathbf{C}\\
&\rightarrow [\mathbf{C}/z](\{[\mathbf{B}/y]([\mathbf{A}/x]\mathbf{E})\})
\end{aligned}$$

with care taken in the case where the expression **A** had free occurrences of y or z in it, or **B** had free occurrences of z in it. (In either case a "renaming" of variables within **E** was necessary to prevent confusion.)

Although this handles currying automatically, it is a bit cumbersome for normal usage when it is obvious that an application has all the arguments it needs to satisfy its function object directly. The notational trick we will use to avoid this clumsiness of one argument at a time is a ***multiple simultaneous substitution,*** denoted "$[\mathbf{A}/x, \mathbf{B}/y, \mathbf{C}/z]\mathbf{E}$," where all free occurrences of all the symbols to the right of the "/" are simultaneously replaced by the expressions to the left of the "/." Thus, if we have an expression of the form

$$((\lambda x_1 \ldots x_n|\mathbf{E})\ \mathbf{A}_1 \ldots \mathbf{A}_m)$$

where n ≥ m, we can feel free to express the beta reduction of this either as the classical string of m single-symbol substitutions, or as one substitution of the form

$$[\mathbf{A}_1/x_1, \ldots, \mathbf{A}_m/x_m]\mathbf{E}$$

In either case, if any of the symbols x_{m+1} through x_n appear free in $\mathbf{A}_1$ through $\mathbf{A}_m$, then a renaming of them is needed before the substitution can go forward. However, if the multiple substitution truly is done simultaneously, then there are no renaming problems within the first m x's.

The beauty of this notation is its agreement with conventional usage in the non-lambda calculus world while at the same time permitting an implementation view that is pure lambda calculus if necessary.

A very common example in lambda calculus where this multiple substitution is useful, even when there is only one argument, is in function definitions that employ within them several applications involving the same subfunction. (This will be particularly true for recursive definitions, where this subfunction is the function itself.) The brute-force solution (as pictured in Figure 4-15) requires duplication of the expression defining the subfunction whenever it is needed.

A cleaner approach (also as pictured in Figure 4-15) is to add a new formal argument to the front of the list of formal parameters in the original function and to use that argument wherever a copy of the desired subfunction is needed. Further, a copy of the subfunction is placed only once—immediately to the right of the original function, where it will be absorbed by the new symbol at application time. The normal argument to the original function goes to the right of the subfunction. Application now involves a simultaneous substitution of the arguments and the subfunction, with results the same as the brute-force approach.

The net effect of this is that we have essentially "named" the subfunction with a local name known only to the body of the function, but usable multiple times by reference to the symbol only, and not by copying the whole expression.

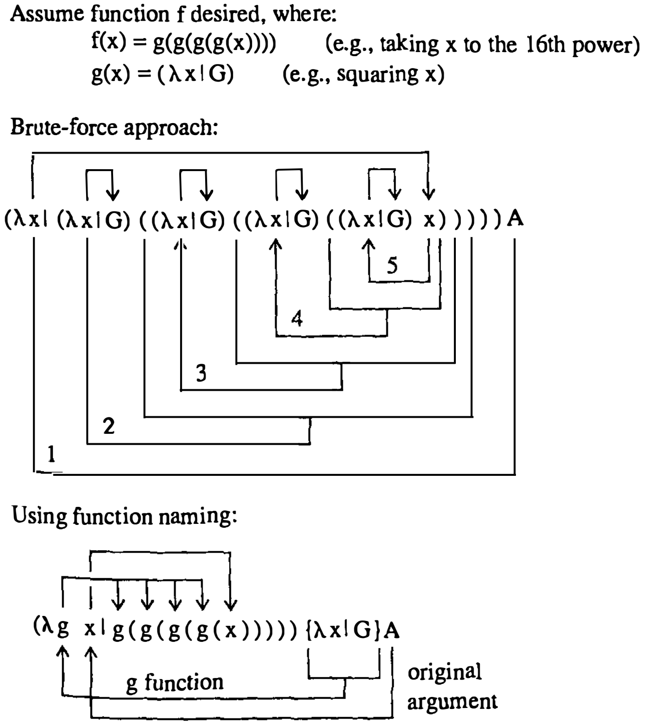

Assume function f desired, where:
f(x) = g(g(g(g(x))))   (e.g., taking x to the 16th power)
g(x) = (λx|G)   (e.g., squaring x)
Brute-force approach:
(λx| (λx|G) ((λx|G) ((λx|G) ((λx|G) x)))))A
5
4
3
2
1
Using function naming:
(λg x|g(g(g(g(x))))) {λx|G}A
g function
original
argument

FIGURE 4-15
Function naming example.

4.10 BASIC ARITHMETIC IN LAMBDA CALCULUS

(Seldin and Hindley, 1980; Stoy, 1977)

The prior sections have described the mechanics of lambda calculus, but have given no real clue how such a simple model can handle nontrivial computations. This section gives a feeling for how this might be done by describing one approach to using lambda calculus for normal integer arithmetic. This will be expanded in the following sections where boolean objects, conditional expressions, and finally recursive functions are described in lambda calculus terms.

As with prior material, this discussion demonstrate the concepts involved without detailed mathematical proofs and backup. The interested reader should see the references, particularly Seldin and Hindley (1980), for the formalities.

The first step to a description for arithmetic is an accurate representation of integers. Mathematically, this can be done recursively by defining what the integer 0 looks like, and a function s (the ***successor function***) which when given an integer k, produces an expression for the integer k+1. From these two objects one can construct any integer one wants by simply applying the successor function to 0 enough times, that is, $s^k(0) = s(s(s\ldots(s(0)\ldots)=k$.

In pure lambda calculus the only kind of expression (object) that one can write down that has no free identifiers is a function. Consequently, it should not be surprising that each integer in lambda calculus is actually a function. In particular, the integer 0 is

$$0 = (\lambda sz|z)$$

As to why this particular expression matches 0 so well, consider the following definition of the successor function:

$$\mathrm{s}(x)=(\lambda x|(\lambda yz|y(xyz)))=(\lambda xyz|y(xyz))$$

Now if we let Z_k represent k applications of the successor function s to 0, we should not be surprised to find that the rest of the integers look like 0, namely:

$Z_0 = 0 = (\lambda sz|z)$ (0 applications of the successor function)
$Z_1 = 1 = (\lambda sz|sz)$ (1 application of successor to 0)
$Z_2 = 2 = (\lambda sz|s(sz))$ (2 applications of successor to 0)
$Z_3 = 3 = (\lambda sz|s(s(sz)))$
...
$Z_k = k = (\lambda sz|s(s(\ldots(sz)\ldots))$ (k applications of s to 0)

Remember that the *s* in these definitions is a binding variable, not the above successor function **s.** It is not until one provides Z_k with an argument does *s* get bound to anything.

As an example, Figure 4-16 diagrams the detailed calculations of the successor of Z_2. As expected, the result is the object we called 3.

These results are very consistent; the integer k, that is, Z_k, is a function of two arguments, with a body that consists of k-nested applications of the first argument to the second. In fact, if we form an application where the function is Z_k and the two arguments are the successor function and 0 itself, the result is still Z_k.

$$(Z_k\ (\lambda xyz|y(xyz)))\ 0) \rightarrow Z_k$$

Further justification of the "rightness" of this notation can be derived by observing the result of applying either of the following two functions to two integer arguments:

$$(\lambda wzyx|wy(zyx))$$
$$(\lambda wzy|w(zy))$$

The first function (***addition***) produces an object which is the integer sum of the two inputs. The second (***multiplication***) produces an object that is equivalent to the product of the two inputs. The reader is encouraged to test his or her understanding of lambda calculus by applying each to two small lambda integers and observing the results.

Other standard integer operations can be defined similarly.

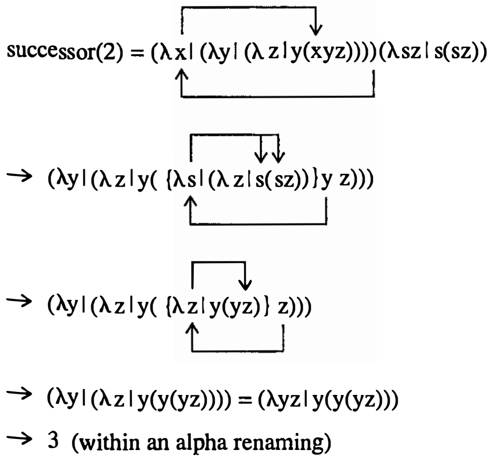

successor(2) = (λx| (λy| (λz|y(xyz))))(λsz| s(sz))
→ (λy|(λz|y({λs|(λz|s(sz))}y z)))
→ (λy|(λz|y({λz|y(yz)} z)))
→ (λy|(λz|y(y(yz)))) = (λyz|y(y(yz)))
→ 3 (within an alpha renaming)

FIGURE 4-16
Sample application of successor function.

4.11 BOOLEAN OPERATIONS IN LAMBDA CALCULUS

Another key part of almost any computational model is the ability to describe ***conditional expressions*** that take one object and see if it has one of two values, true or false. On the basis of this value, one of two other arbitrary expressions is returned as the "value" of the total expression.

Again without any deep mathematical derivations we present two objects which we will call the values true and false (note that the object for false is the same as that for 0):

true = T = (λ*xy*|*x*)
false = F = (λ*xy*|*y*)

As before, these objects are lambda functions with two arguments and no free variables. However, unlike the integers, where the internal body of the function had regularities that matched our concepts of integers, these functions match the truth values because of the way they function when presented with real arguments. Consider the expression **PQR,** where **Q** and **R** are arbitrary expressions and **P** is either T or F:

If **P**=T then **PQR** = T **Q R** = [**Q**/*x*,**R**/*y*]*x* = **Q**
If **P**=F then **PQR** = F **Q R** = [**Q**/*x*,**R**/*y*]*y* = **R**

This is exactly what we would want for a conditional expression. Either **Q** or **R** is returned without any modification. Consequently, we now have a way of building arbitrary conditional expressions of the form "if **P** then **Q** else **R**." We start with a lambda expression for **P** and construct it to return either T or F after reduction to normal form. We then simply concatenate to the left the desired expressions for **Q** and **R** [with "( )" as necessary to avoid confusion].

The lambda expression for **P** is a ***predicate expression*** that performs whatever tests are desired. The simplest such predicates are ***logical expressions*** built out of applying ***logical connectives,*** such as **and, or,** or **not,** to other expressions which themselves reduce to T or F. These connectives are themselves expressible as simple lambda functions as pictured in Figure 4-17.

Next, one might wonder how to construct more complex predicates that perform real "tests" on objects. Figure 4-18 diagrams one such lambda expression for testing an integer for a zero value. Treating this expression as a function with a single integer argument will return T if the integer expression is the 0 defined above, and F if it is any other integer. As with many real functions, there is no guarantee that valid boolean objects will be returned if the actual argument is other than an integer.

```
not = (λw|wFT)
    = (λw|w(λxy|y)(λxy|x))
    Example: not T → TFT → F

and = (λwz|wzF)
    = (λwz|wz(λxy|y))
    Example: and T F → TFF → F

or = (λwz|wTz))
   = (λwz|w(λxy|x)z)
   Example: or F T → F T T → T
```

FIGURE 4-17
Standard logical connectives.

```
zero(x) = (λx|x F not F)
        = (λx|x (λxy|y) (λw|w(λxy|y)(λxy|x)) (λxy|y))

Examples:

  zero(0) = (λx|x F not F) (λnz|z)
          → (λnz|z) F not F
          → ([F/n,not/z]z) F
          → not F
          → T

  zero(1) = (λx|x F not F) (λnz|nz)
          → (λnz|nz) F not F
          → ([F/n,not/z]nz) F
          → (F not) F = F not F
          → F

  zero(Zk) = (λx|x F not F) (λnz|n(..(nz)..)), k>0
           → (λnz|n(..(nz)..)) F not F
           → ([F/n,not/z]{n(n...(nz). .))}) F
           → (F {F ...(F not)..)}) F = F {F ...(F not)..} F
           → F
```

FIGURE 4-18
A lambda zero test predicate.

As a side note, it is interesting to observe that the essentially normal-order reductions used in the examples of Figure 4-18 actually avoid potentially long and involved calculations in the first argument to the first F function (the "{F...(F **not**)...)}" argument) by immediately deleting it. An applicative-order reduction sequence would spend considerable effort, particularly for large integers, in this evaluation.

Finally, to demonstrate the use of conditionals, booleans, and integers, Figure 4-19 diagrams a complete lambda function which, if given an integer, will return 0 if the integer argument was 0, and the integer 1 otherwise. The reader is again invited to work out an example with a real input.

$$\begin{aligned}
&\mathrm{f(a)} = \text{if zero(a) then 0 else } \mathrm{a}+1\\
&= (\lambda \mathrm{a}|\text{zero a 0 (successor a)})\\
&= (\lambda \mathrm{a}|(\lambda x|x\ (\lambda nz|z)\ (\lambda w|w(\lambda xy|y)(\lambda xy|x))\ (\lambda nz|z))\\
&\qquad \mathrm{a}\\
&\qquad (\lambda nz|z)\\
&\qquad (\{\lambda xyz|y(xyz)\}\ \mathrm{a}))
\end{aligned}$$

FIGURE 4-19
A complex lambda function.

4.12 RECURSION IN LAMBDA CALCULUS

A ***recursive function*** is one that invokes itself as a subfunction inside its own definition. Given that lambda calculus has essentially "anonymous" functions, at first glance it would seem difficult for a lambda expression to perform such a self-reference. This section addresses one such approach to overcome this.

Consider the application **RA,** where **R** is some recursively defined function object and **A** is some argument expression. If **A** satisfies the ***basis test*** for **R,** then **RA** reduces to the ***basis case.*** If it does not, then **RA** reduces to some other expression of the form "...(**RB**)...," where **B** is some "simpler" expression. Essentially, making **R** recursive has a lot to do with making it "repeat itself."

To see how to do this self-repetition in general, we first observe the reduction of a particularly peculiar expression:

$$\begin{aligned}
&((\lambda x|xx)(\lambda x|xx))\\
&\to [(\lambda x|xx)/x](xx)\\
&\to ((\lambda x|xx)(\lambda x|xx))
\end{aligned}$$

This expression has the property that it does not change regardless of how many beta conversions are performed. It always generates an exact copy of itself.

Now, using this as a basis, for any lambda expression **R**:

$$\begin{aligned}
&((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx)))\\
&\to [(\lambda x|\mathbf{R}(xx))/x](\mathbf{R}(xx))\\
&\to \mathbf{R}\ ((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx)))\\
&\to \mathbf{R}\ (\mathbf{R}\ ((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx))))\\
&\to \ldots\\
&\to \mathbf{R}\ (\mathbf{R}\ (\mathbf{R}\ (\ldots((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx))))\ldots)
\end{aligned}$$

This allows us to compose an arbitrary function **R** on itself an infinite number of times. The only difficulty is that the function **R** must be embedded in several parts of this other expression. This, however, can be avoided by defining the following function:

$$\mathbf{Y} = (\lambda y|((\lambda x|y(xx))(\lambda x|y(xx))))$$

Now see what happens when we apply **Y** to any other lambda expression **R**:

$$\begin{aligned}
&\mathbf{YR} \to [\mathbf{R}/y]((\lambda x|y(xx))(\lambda x|y(xx)))\\
&\to ((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx)))\\
&\to \mathbf{R(YR)}\\
&\to \ldots\\
&\to \mathbf{R(R(..R(YR)..)}.
\end{aligned}$$

This function **Y** will duplicate any other function, and compose itself on itself any number of times. In Chapter 12 we will show how it is one of a special set of functions called ***combinators*** from which one can build all of lambda calculus, and thus everything described in this chapter. In particular, **Y** is called the ***fixed-point combinator*** function.

Now consider **(YR)A** → **R(YR)A.** If the object **R** is a function of two arguments, it could absorb as its first argument a self-replicating copy of itself (**YR**), and as its second argument the expression **A.** With a normal-order reduction sequence (to prevent **YR** from blowing up), such a function could basis-test **A** and discard the **YR** term (unevaluated) if the test passes. If the basis test fails, this term (**YR**) could be used to generate a recursive copy of **R** for the next call.

With the above ideas in mind, we can now construct a prototypical form for a recursive lambda expression. The expression as a whole would have the form **YR,** where the function-unique expression **R** is of the form ($\lambda fx|$**E**). Within **E** the identifier *f* provides the function part of an application whenever a recursive call to **R** is needed. The identifier *x* is used whenever the actual argument to the function is needed.

Applying this expression to an argument **A** then takes the form **YRA.**

As a detailed example, consider the recursive definition of **factorial** that was used in Chapter 3.

$$\mathbf{fact}(n) = \text{if } \mathbf{zero}(n) \text{ then } 1 \text{ else } n\times\mathbf{fact}(n-1).$$

Assuming that integer arithmetic and conditional expressions are defined as in the prior sections, and that we have available the full lambda equivalents of the functions "**zero,**" "×," and "−," the lambda-calculus equivalent form of this function is

$$\mathbf{fact}(n) = \mathbf{Y}(\lambda fn|\ \mathbf{zero}\ 1\ (\times n\ (f\ (-n\ 1))))$$

Using a more or less normal-order reduction sequence, Figure 4-20 applies this function to the integer 4, and reduces down until normal form is reached; i.e., we get the correct answer, 24.

```
Assume:
   Y = (λy|((λx|y(xx))(λx|y(xx))))
   R = (λr|λn|zero n 1 (×n(r(−n1))) )


Then 4! = Y R 4
   → R (YR) 4
   → (λn|zero n 1 (× n( (YR) (−n1)))) 4
   → zero 4 1 (× 4 ((YR) (− 4 1)))
   → zero 4 1 (× 4 ((YR) 3))
   → F 1 (× 4 ((YR) 3))
   → (× 4 ((YR) 3))
   → (× 4 (R (YR) 3))
   → (× 4 ((λr|λn|zero n 1 (× n(r(−n1)))) (YR) 3))
   → (× 4 ((λn|zero n 1 (× n( (YR) (−n1)))) 3))
   → (× 4 (zero 3 1 (× 3 ( (YR) (−3 1))))
   → (× 4 (zero 3 1 (× 3 ( (YR) 2))))
   → (× 4 (F 1 (× 3 ( (YR) 2))))
   → (× 4 ((× 3 ( (YR) 2))))
   → (× 4 (× 3 ( (YR) 2)))
   → (× 4 (× 3 ((λr|λn|zero n 1 (× n(r(−n1)))) (YR) 2)))
   → (× 4 (× 3 ((λn|zero n 1 (× n((YR)(−n1)))) 2)))
   → (× 4 (× 3 (zero 2 1 (× 2 ((YR)(−2 1)))))))
   → (× 4 (× 3 (zero 2 1 (× 2 ((YR)1))))))
   → (× 4 (× 3 (F 1 (× 2 ((YR)1))))))
   → (× 4 (× 3 ((× 2 ((YR)1))))))
   → (× 4 (× 3 ((× 2 (R(YR)1))))))
   → (× 4 (× 3 ((× 2 ((zero 1 1 (× 1 ((YR)(− 1 1))))))))))
   → (× 4 (× 3 ((× 2 ((F 1 (× 1 ((YR)0))))))))))
   → (× 4 (× 3 ((× 2 ((× 1 ((YR)0))))))))
   → (× 4 (× 3 ((× 2 ((× 1 (R(YR)0))))))))
   → (× 4 (× 3 ((× 2 ((× 1 (zero 0 1 (× 0((YR)(− 0 1)))))))))))
   → (× 4 (× 3 ((× 2 ((× 1 (T 1 (× 0((YR)(− 0 1)))))))))))
   → (× 4 (× 3 ((× 2 ((× 1 1))))))
   → . . . 24
```

FIGURE 4-20
Reduction of 4

4.13 PROBLEMS

1. Compute all the variations in application sequences possible for Figure 4-2 and show that they yield the same answer.
2. In the following lambda expressions, which identifiers are free, and which are bound, and to which lambda.
   a. ((λ*y*|*yxx*)*y x*)
   b. ((λ*x*|(λ*x*|(λ*x*|*x*)*x*)*x*)*x*)
   c. (λ*x*|(*xy*(λ*y*|((*xy*)(λ*x*|*wxyz*))(λ*z*|*wyz*))))
   d. (λ*x*|*y*(λ*y*|*x*(λ*x*|*xz*(λ*y*|*yx*))))

3. Evaluate, showing each reduction: (λ*f*|+(*f* 3)(*f* 4))(λ*x*|×*xx*).
4. Show that the lambda expression **not, and, or** work (show what happens when presented with all combinations of T and F).
5. Define a lambda expression for **xor** (***exclusive or***). Show both in terms on T and F, and when the body of the function has been fully reduced to normal order.
6. Show that (Z_k (λ*xyz*|*y*(*xyz*)) 0) → Z_k. How does this reinforce the notion that Z_k is a reasonable representation of the integer k?
7. Add 3 and 2 using the lambda definitions of add and integers. Show all steps.
8. Multiply 3 and 2 using the lambda definitions of multiply and integers.
9. Assuming that Z_i and Z_j represent lambda expressions for integers as defined in the text, evaluate each for Z_i=2 and Z_j=3. What exactly is the second function?

   (λ*mnf*|*m*(*nf*)) Z_i Z_j
   (λ*mn*|*nm*) Z_i Z_j
10. Write a recursive lambda expression for Ackerman's function (Chapter 3). You may assume the "primitive" functions +, −, **zero,** and the integers. Show where and how **Y** is used.
11. (Hard) Develop a lambda definition for the function **predecessor**(*x*)=*x*−1. Indicate what happens in your definition when *x*=0.
12. Write a recursive lambda expression for computing the n-th Fibonacci number F_n, where $F_n=F_{n-1}+F_{n-2}$, with $F_0=F_1=1$. Evaluate your function when applied to 3.

CHAPTER 5

A FORMAL BASIS FOR ABSTRACT PROGRAMMING

Although lambda calculus is a complete system for describing arbitrary computations, it is a difficult notation for humans to use. The deep nesting of parentheses and the relatively abstruse way in which certain calculations must be represented (such as recursion) makes the reading and writing of lambda expressions very susceptible to mistakes.

In response to this, the notation called ***abstract programming*** (informally introduced earlier) has been developed, which is much simpler to read and yet convertible to completely equivalent pure lambda calculus. In fact, it would be a relatively easy project for a computer science student to produce a compiler to convert a simple abstract program into a pure lambda expression.

This chapter gives a formal definition of abstract programming in terms of both syntax and semantics. The basic approach is to use an ***equational form*** for defining named functions, and to use generous helpings of "syntatic sugar" to express many of the tricks for writing lambda expressions in terms that look like expressions from conventional programming languages. Examples of the latter include conditional if-then-else blocks and local nested definitions of functions and other objects. As before, significant informality will be acceptable when the meaning is obvious.

The following sections give a BNF description for the syntax of an abstract program and a formal set of translation rules for conversion of this syntax into pure lambda calculus. Given that we understand the meaning of a lambda expression, these latter rules thus form a ***semantic model*** for abstract programs. References for other approaches include McCarthy et al. (1965), Burge (1975), and Henderson (1980).

5.1 A BASIC SYNTAX

Figure 5-1 gives a basic BNF description of the syntax of an abstract program. The description here should be treated as basic introduction. Following sections will consider each of the major syntatic units and describe in more detail how to convert them into pure lambda calculus.

The first three syntatic units: ⟨identifier⟩, ⟨function-name⟩, and ⟨constant⟩, are self-explanatory. They are appropriate strings of characters. This is slightly different from our previous lambda calculus syntax, where identifiers were assumed to be only one character long. In abstract programming, multiple-character strings will be one identifier unless separated by spaces or other characters that cannot be part of a name. This agrees with conventional programming notation.

```
<identifier> := <alpha-char>{<alpha-char>|<number>}*

<function-name> := <identifier>

<constant> := <number> | <boolean> | <char-string>

<expression> := <constant>|  <identifier>
               |  (λ<identifier>"|"<expression>)
               |  (<expression>+)
               |<function-name>(<expression>{,<expression>}*)
               |  let <definition> in <body>
               |  letrec <definition> in <body>
               |  <body> where <definition>
               |  <body> whererec <definition>
               |  if <expression> then <expression>
                  else <expression>
               |  ..."standard arithmetic expressions"...

<body>:=<expression>

<definition>:=  <header>=<expression>
               | <definition> {and <definition>}*

<header> :=  <identifier>
           | <function-name>(<identifier>{,<identifier>}*)

<abstract-program>:=<expression>
```

FIGURE 5-1
BNF for abstract programs.

The major syntatic unit, an ⟨abstract program⟩, is equivalent to an ⟨expression⟩. An expression, in turn, can take on quite a few forms, the first four of which mirror lambda calculus almost directly.

Perhaps the most obvious difference comes in the expression of applications. In addition to simply concatenating expressions and treating the leftmost as the function, abstract programming also permits a conventional mathematical notation where we refer to the function by name and list its argument expressions within a single set of "( )" and separated by commas. The only constraint is that this function must be defined in a surrounding let or equivalent.

The let and letrec (and equivalent where and whererec) forms of expression permit statically scoped definitions of functions and identifiers and may be cascaded in several ways to control the values given to symbols in the bodies of the expressions. Although they look like assignment statements, they are not. There is no sense of assigning a reference or storage to a symbol, nor is there ever a change to the value given a symbol. Such expressions are much closer to macro definitions, where the use of a symbol within the scope of the macro is equivalent to replacing the symbol by its equivalent textual definition.

Finally, to highlight cases where boolean objects will be used in conditional expressions, an expression may also separate out into three subexpressions bounded by the keywords if, then, and else, with an obvious interpretation. Note that the else is *not* optional here, as conditional expressions always have a value selected by the boolean test.

Figure 5-2 gives several equivalent abstract program expressions for the evaluation of a factorial.

As before, shortcuts in notation are permissible whenever they make sense, such as in the use of standard infix notation for arithmetic calculations and the elimination of obvious parentheses. Also, as with lambda calculus, when discussing abstract programs we will use single capital letters to represent places where arbitrary expressions could be substituted without changing the meaning of the discussion, as in if **P** then **A** else **B**.

5.2 CONSTANTS

Semantically, a ***constant*** is any object whose name denotes its value directly. In abstract programming we will assume that at a minimum we have syntax for constants of types **boolean, integer,** and **character string,** all of which can be described by their normal written form, and all of which translate directly into equivalent lambda expressions. These lambda expression equivalents represent their semantic meaning.

```
letrec fact  =  (λn|if n=0 then 1 else n×fact (n−1)) in fact(4)

fact(4) whererec fact(n)  =  if n=0 then 1 else n×fact(n−1)

letrec fact(n)  =  (if n=0 then 1 else z)
                    whererec z=n×fact(n−1) ) in (fact 4)
```

FIGURE 5-2
Sample equivalent definitions.

Occurrences of either T or F in an abstract program should thus be taken as the expressions $(\lambda xy|x)$ and $(\lambda xy|y)$, respectively. As described earlier, the meaning of these objects is the way they selectively choose one of the two following expressions.

Any nonnegative integer k translates directly to the form $(\lambda sz|s^kz)$ as described earlier. Negative integers have several possible forms, including functions like $(\lambda n|n-k)$, where k is the positive equivalent.

There are several possible interpretations for character strings. The one we will assume here is as potentially very large integers, where each character is converted to an integer code (as in the ASCII encoding) and then scaled by some number raised to a power equaling its position in the string. Thus, "Hi" might translate to $72\times(256^1)+105\times256^0=18537$. This corresponds closely to standard interpretations in conventional computers.

From these basic data types it is possible to construct notations for arbitrary integers, floating-point numbers, etc. Although we will not describe these conversions here, we will feel free to assume their validity, and will use the more conventional notation whenever possible.

More complex data types, such as lists, can also be expressed as integers, although the conversion process becomes even more remote and unrelated to conventional thought. This is particularly true when discussing, for example, lists where the elements themselves may be recursively defined lists. Chapter 12 will describe one such implementation in terms of functions, called combinators, which can mirror in all important ways the operation of **car, cdr,** and **cons.**

Finally, since in lambda calculus there is no syntatic difference between functions and any of the "constants" described above, we will feel free to assume as another set of constants any of the standard arithmetic, logical, character string, or list-processing operators found in conventional mathematics.

5.3 FUNCTION APPLICATIONS

Besides constants, the simplest form of an expression in abstract programming is the application of a function to one or more arguments. Any pure lambda expression representing an application is acceptable here. This includes the use of normal infix arithmetic notation, since we know precisely how to convert such expressions into the prefix form using pure lambda equivalents of operators and numbers. Thus:

$$x+3 \equiv (\lambda wzyx|wy(zyx)) \times (\lambda nz|n(n(nz)))$$

In addition, however, abstract programming provides a more conventional form of application where we identify the function we want by name, and group all its arguments together inside a set of "( )" and separated by ","—e.g.:

⟨function-name⟩ (⟨expression⟩ {, ⟨expression⟩}*)

The only constraint is that this expression be embedded inside some larger expression that gives a definition to the function via a let or equivalent (discussed later).

As before, we give a meaning to such an application by giving its lambda equivalent, namely, "$\mathbf{f}(\mathbf{A}_1,\ldots,\mathbf{A}_n)$" is equivalent to "$(\mathbf{F}\,\mathbf{B}_1\ldots\mathbf{B}_n)$," where $\mathbf{F}$ is the lambda-expression form of $\mathbf{f}$ [i.e., something like $(\lambda x_1 \ldots x_n|\mathbf{E})$] and $\mathbf{B}_k$ is the lambda equivalent of $\mathbf{A}_k$.

With this translation, the meaning is direct; the argument expressions are substituted into the appropriate places in the function's body by applying the normal rules of lambda substitution.

5.4 CONDITIONAL EXPRESSIONS

The next most useful notation in abstract programming is the ***conditional expression.*** In pure lambda calculus an expression of the form **PQR** operates as a conditional if the expression **P** evaluates to either a true value $(\lambda xy|x)$ or a false value $(\lambda xy|y)$.

Abstract programming notation makes this type of expression easier to read by separating the three subexpressions by the keywords if, then, and else in the form if **P** then **Q** else **R.** The meaning to a human programmer appears obvious: If the expression **P** is T, then return the value of **Q**; otherwise return **R.** The lambda calculus equivalent mirrors this. We simply convert each subexpression to its pure lambda form, concatenate, and surround by "( )." (This guarantees that the **P** expression is treated as a function.)

There are two slight differences between the abstract conditional and the conventional form. First, conventional languages do not define what happens when **P** evaluates to anything other than T or F. The lambda calculus form does—the two expressions are accepted as operands by whatever **P** is. Second, conventional languages often permit the "else **R**" to be optional. This is because the bodies of the conditional are statements that return no value and can either be executed or not executed. In lambda calculus, however, dropping the else term means that there is no second argument for the boolean to absorb. The result is some sort of curried function that causes havoc to further processing. Consequently, to avoid confusion the abstract program form of the condition insists on an else expression.

if-then-else's may be nested as deeply as desired in either the then or else expression. A particularly useful form chains ifs to elses as in something akin to a ***case statement*** in a conventional language such as Pascal:

if $\mathbf{P}_1$ then $\mathbf{E}_1$ else if $\mathbf{P}_2$ then $\mathbf{E}_2$ else...if $\mathbf{P}_n$ then $\mathbf{E}_n$ else $\mathbf{E}_{n+1}$

The meaning of this is that we find the first $\mathbf{P}_k$ that is true, and return as its value $\mathbf{E}_k$. If none of the **P**s are true, return $\mathbf{E}_{n+1}$.

The translation of this to lambda form is direct:

$$(\mathbf{P}_1\ \mathbf{E}_1\ (\mathbf{P}_2\ \mathbf{E}_2\ (\ldots(\mathbf{P}_n\ \mathbf{E}_n\ \mathbf{E}_{n+1})\ldots)$$

For simplicity, the keyword elseif will be used for such combinations.

5.5 LET EXPRESSIONS—LOCAL DEFINITIONS

Many conventional programming languages offer a ***macroexpansion*** capability whereby for a certain region of text in the program (called the ***scope***) all occurrences of a certain identifier are to be replaced by some predefined text string. In other languages, where the passing of arguments to a procedure is by ***call-by-name*** semantics, the occurrence of a formal argument in the body of the procedure actually means replacement of that argument by the result obtained by evaluating the text corresponding to the actual argument.

Both of these mechanisms have two things in common. First, they simplify the writing of expressions by letting some identifier "stand for" some other expression. Second, the use of this identifier is very controlled; throughout the entire scope the identifier's value as an externally defined expression (almost) never changes.

Abstract programming has a notation very akin to a cleaned-up combination of macros and call-by-name. The simplest form of this notation (called a ***let expression***) is

let ⟨identifier⟩=⟨expression⟩ in ⟨body⟩

where ⟨body⟩ is itself an arbitrary expression.

This whole expression is equivalent in value to a copy of the body where every free occurrence of the identifier in the definition part is replaced by the expression in the definition. In terms of its lambda equivalent (from which we get its exact meaning), a let expression of the form "let x=**A** in **E**" is the same as "$(\lambda x|\mathbf{E})\mathbf{A}$," and means that we must perform the substitution $[\mathbf{A}/x]\mathbf{E}$. The let expression is totally equivalent to an application of an anonymous function to an argument, where the formal parameter for the function is the identifier in the definition, the body of the function is the body of the let, and the argument is the right-hand side of the application.

As discussed earlier, this conversion process brings with it a precise definition of which occurrences of x in **E** are targets of the substitution, and what happens if a part of **E** that contains a free x also binds a symbol that is free in **A** (we must "rename" the former binding variable). Figure 5-3 diagrams a sample let demonstrating all these possibilities.

5.5.1 Local Nonrecursive Functions

This definition of let places no constraints on the type of expression of the right-hand side of the "=" in the definition. In particular, it may be an arbitrary lambda function, in which case the identifier in the definition becomes a ***local function*** to the body of the let, as in

let $square = (\lambda x|(x\times x))$ in $square(square(2))$

This is such a handy notation that our definition of abstract programming includes a special form of the let definition that eliminates the λ from the definition's body, and transfers the formal argument from the body's lambda to be inside some parentheses next to the function's name. Thus we have as a form of let:

let ⟨function-name⟩(⟨id⟩ {,⟨id⟩}*) = ⟨expression⟩ in ⟨body⟩

where this is totally equivalent to

let ⟨function-name⟩ = (λ⟨id⟩{⟨id⟩}* |⟨expression⟩) in ⟨body⟩

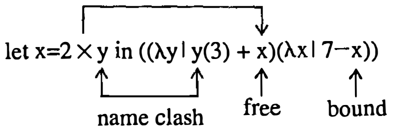
or

(λ⟨function-name⟩|⟨body⟩) (λ⟨id⟩⟨id⟩*|⟨expression⟩)

Thus the above example is equivalent to the more readable

let **square**$(x) = x\times x$ in **square(square(2))**

Either one is equivalent to $(\lambda s|s(s\ 2))\ (\lambda x\,|\,x\times x)$.



is equivalent to

$(\lambda x|\ (\lambda y|y(3) + x)(\lambda x|7-x))(2\times y)$

$\rightarrow ((\lambda z|z(3) + 2\times y)(\lambda x|7-x))$

$\rightarrow 4+2\times y$

FIGURE 5-3
A sample let expression.

Figure 5-4 diagrams another example in detail. In both cases, the new notation is quite easy to read, and agrees closely with notation found in many conventional languages. However, the reader should take care to study these examples closely and verify exactly how each part of the new let form migrates into the multiple functions in the lambda equivalent.

5.5.2 Nested Definitions

Given that a let expression is itself an expression, it is also possible to nest one inside another, particularly inside the other's body. Again the meaning of this is an exact extension of the basic let equivalence. Figure 5-5 diagrams a triply nested let and its lambda equivalent.

As before, it is important to apply the rules for identifying free instances and performing the resulting substitutions properly. The second example in Figure 5-5 diagrams such a case where the middle let has two different x's, the x in the definition expression that is receiving the value 7 from the outermost let, and the x that is equivalent to the value 8 and is being substituted into the third let.

5.5.3 Block Definitions

One important consequence of the nesting rules for let expression is that identifiers that appear free in the expressions for definitions of inner lets are perfectly valid candidates for replacement by definitions in more outer lets. Thus, in "let x=**A** in let y=...x...in...," the free x in the y definition is replaced by **A.**

There are many cases, however, where what is desired is a simultaneous replacement of several definitions into a let body, with no cross-substitutions among the definitions. In our abstract programming language the ***and*** form of definitions permits this simultaneity. An expression of the form

let x=**A** and y=**B** and z=**C** in **E**

let $f = (\lambda xy\,|\,x+3\times y+1)$ in $f(2+x,7)$

or

let $f(x,y) = x+3\times y+1$ in $f(2+x,7)$

are both equivalent to:

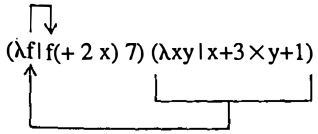
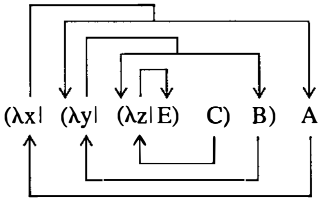


FIGURE 5-4
Ways to define local functions.

```
let x = A in
   let y = B in
      let z = C in E
```



Substitution eqvt: [A/x]([B/y]([C/z]E))

```
Sample: let x=7 in
              let x=x+1 in
                 let f(y)=y+2 × x in x × f(3)

   → let x=8 in let f(y)=y+2 × x in x × f(3)
   → let f(y)=y+16 in 8 × f(3)
   → 8 × (3+16) → 152

Equivalent: (λx | [λx | {λf | x × f(3)} {λy | y+2 × x}][x+1])7
   ≡ [7/x]([x+1/x]([(λy | y+2 × x)/f](x × f(3))))
   → [7/x]([x+1/x](x × (λ y | y+2 × x)(3)))
   → [7/x]([x+1/x](x × (3+2 × x)))
   → [7/x((x+1) × (3+2 × (x+1)))
   → ((7+1) × (3+2 × (7+1)))
   → 152
```

FIGURE 5-5
Nested lets.

means that *x*, *y*, and *z* should be simultaneously replaced in **E** by **A, B,** and **C,** respectively, and that any free *x*, *y*, and *z*'s in **A, B,** or **C** should remain free and should not be replaced by **A, B,** or **C** as happens in the nested form. (Note that an *x* that is free in **A** stays free anyway.)

In terms of pure lambda equivalents, such and forms correspond to creating a function application where the new anonymous function has multiple arguments, namely:

(λ*xyz*|**E**) **A B C**

Evaluating such an expression results in either the triply nested substitution

[**C**/*z*]([**B**/*y*]([**A**/*x*]**E**))

or the equivalent simultaneous substitution [**C**/*z*,**B**/*y*,**A**/*x*]**E.** Note that when the nested substitution is performed inside out and one at a time, the renaming rule will guarantee that any free *z*'s in **A** or **B** and any free variables in the arguments will stay free and unchanged. Figure 5-6 gives an example of this.

5.5.4 Where Expressions

Finally, there are cases where understanding of an expression is enhanced by placing the definitions after the expression they affect. Our version of abstract programming includes the where expression to permit this syntatic reversal; i.e.,

⟨body⟩ where ⟨definition⟩

is totally equivalent to

let ⟨definition⟩ in ⟨body⟩

Further, the where expression may be expanded in an and form, again with exactly the same meaning as the let version.

5.6 RECURSIVE DEFINITIONS

One limitation of the let and where expressions is that they do not permit recursion in a function definition. An expression of the form

let *f*(*n*)=if **zero**(*n*) then 1 else *n*×*f*(*n*−1) in *f*(4)

does not result in the intended effect of having *f* call itself recursively when $n \neq 0$. Instead, the *f* in the definition's expression is replaced by whatever definitions of *f* exist in surrounding expressions.

The brute-force solution to this is to use the ***fixed-point combinator***

```
let x=(y x 1) and y=(zyx 2) and z=(yxz 3) in xyz

≡ (λxyz|xyz) (yx 1) (zyx2) (yxz 3)

→ {[(yx 1)/x](λyz|xyz)}(zyx2) (yxz 3)

→ {(λqz|(yx 1)qz)} (zyx2) (yxz 3)—"y renamed to q"

→ {[(zyx2)/q](λz|(yx 1)qz)} (yxz 3)

→ {(λr|(yx 1)(zyx2)r)} (yxz 3)—"z renamed to r"

→ (yx 1) (zyx2) (yxz 3)
```

FIGURE 5-6
Sample and form of let.

Y, defined earlier, at carefully selected spots in the let expression. Unfortunately, the result is not terribly readable.

The alternative used in abstract programming is the letrec expression, a recursive form of the standard let. Here the major difference is that any free occurrence of the definition's identifier in the definition's expression is replaced by the expression itself. Thus, in

letrec $f(n)$=if **zero**(n) then 1 else $n \times f(n-1)$ in $f(4)$

the free occurrence f in $f(n-1)$ is replaced (recursively) by the whole expression

if **zero**(n) then 1 else $n \times f(n-1)$

All uses of f in the letrec's body now are replaced by this whole recursively defined expression.

The conversion from this notation to a pure lambda equivalent is direct. Given "letrec f=**A** in **E**," we form an application where the function is an anonymous function whose formal parameter is f and whose body is **E**. This is just as with let. The actual argument to this function is, however, different. Instead of just using **A**, we create the object $(\lambda f|\mathbf{A})$ and use that as an argument to the function **Y**. The resulting application is the recursive function we want and is then used as the argument to the outermost application.

In total, the lambda equivalent for

letrec f = **A** in **E**

is

$$(\lambda f|\mathbf{E})\ (\mathbf{Y}\ (\lambda f|\mathbf{A})) = (\lambda f|\mathbf{E})\ (\{\lambda y|(\lambda x|y(xx))(\lambda x|y(xx))\}\ \{\lambda f|\mathbf{A}\})$$

A letrec expression is really useful only for defining local functions (try out your patience on "letrec x=x+1 in x"). Consequently, one would expect the "**A**" in "letrec f=**A** in **E**" to be a lambda function itself (of the form $(\lambda x|\mathbf{B})$). As with let, this is permissible, but a more human-readable form is possible if we permit listing the arguments of f next to f, with only the function body on the right-hand side.

In summary, the conversion process for an expression of the form "letrec $f(x)$=**B** in **E**" involves making a lambda function $(\lambda x|\mathbf{B})$ and creating a nested set of applications from it as defined above. To verify this, consider:

letrec $f(n)$=if **zero**(n) then 1 else $n \times f(n-1)$ in $f(4)$

Its pure lambda equivalent is only one reduction away from that detailed at the end of Chapter 4:

$$(\lambda f|f\ 4)\ (\mathbf{Y}\ (\lambda fn|\mathbf{zero}\ n\ 1\ (\times n\ (f\ (-n\ 1)))))$$

The nesting of letrecs is also permissible, and mirrors nested lets directly. Here the definition of any recursive function can reference any recursive function defined in surrounding letrecs, but cannot use definitions in deeper letrecs.

5.6.1 Mutually Recursive Functions

In contrast to let, however, there are some subtle differences between multiple definitions in the ***and form*** of a letrec versus a let. In the latter there is absolutely no connection between the expressions used in the definitions, even though they may use some of the same identifiers being given definitions. In the letrec form, however, the expression in each anded definition has complete access to every other definition. The result is a set of ***mutually recursive functions*** that are defined together.

This mutual dependency makes conversion of a multiple-definition letrec somewhat more challenging than anything we have done yet. Consider, for example, the expression

letrec $f(x)$=**A** and $g(x,y)$=**B** in **E**

where the expressions **A** and **B** both involve applications using f and g.

As before, this conversion consists of an application of the form $(\lambda fg|\mathbf{E})\mathbf{FG}$, where **F** and **G** are expressions for the functions f and g. As with a single recursion, these **F** and **G** contain objects for f and g that accept as arguments what will be copies of both f and g, along with their natural arguments. They are of the form $(\lambda fgx|\mathbf{A})$ for f and $(\lambda fgxy|\mathbf{B})$ for g.

Define combinators:

$Y_1 = (\lambda fg|RRS)$

$Y_2 = (\lambda fg|SRS)$

where:

$R = (\lambda rs|f(rrs)(srs))$

$S = (\lambda rs|g(rrs)(srs))$

Properties:

$Y_1\ F\ G = F\ (Y_1\ F\ G)\ (Y_2\ F\ G)$

$Y_2\ F\ G = G\ (Y_1\ F\ G)\ (Y_2\ F\ G)$

FIGURE 5-7
Pairwise mutually recursive combinators.

```
letrec f(x) = A and g(x,y) = B in E
```

is equivalent to

$(\lambda fg|E)\ \{Y_1(\lambda fgx|A)(\lambda fgxy|B)\}\ \{Y_2(\lambda fgx|A)(\lambda fgxy|B)\}$

FIGURE 5-8
Conversion of a dual-definition letrec.

The hard part with this is completing the expansion of **F** and **G**. We have to use something like the **Y** expression used above, but **Y** by itself will not work. It involves recursion over only one function, and has no way of introducing dependencies on another. The **Y** equivalents that do work are shown in Figure 5-7. Each one accepts two arguments (the nonrecursive forms of the two functions) and creates an expression consisting of one of the nonrecursive forms applied to two arguments which represent the recursive forms of both *f* and *g*. When this application is evaluated, it gives us a curried function whose remaining arguments are the natural arguments for the function. This is exactly what we want to substitute into **E**.

Figure 5-8 gives a complete conversion. It is left as an exercise to the reader to expand the approach to handle the case where there are three or more mutually recursive functions defined as anded terms in a single letrec.

As with let, the whererec expression is identical to the letrec, but with the definitions given after the body rather than before.

5.7 GLOBAL DEFINITIONS

One final syntatic simplification deals with the common habit of describing different but related functions in different places in a document. Strictly speaking all such definitions should be collected in a single expression, with the end problem to be solved written as an expression involving these definitions as the body of the outermost let or letrec. This should include all the "built-in" functions which we all know can be written but do not want to spend the time or paper describing (e.g., square root or **car, cdr, cons,**...).

The solution to be assumed in this text is that all expressions that are not obviously self-contained are assumed to be lumped as and definitions in a single large letrec, with other and definitions assumed to cover whatever functions are missing. Figure 5-9 gives an example of this.

5.8 HIGHER-ORDER FUNCTIONS

As has been said before, in pure lambda calculus everything is a function. In abstract programming we have simplified much of the notation to hide many common function equivalents (such as integers or booleans), but not deleted the capability. It is still perfectly acceptable to either pass a function as an argument or get one as a result. To distinguish such functions from the more mundane ones, we call them ***higher-order functions***.

Assume the following are defined at different places in a document:

```
f(x) = ...
g(x,y) = ...
h(z) = ...
s = 0
msg = "Hi There"
basevalue = f(s)
```

with g(msg,basevalue) as the desired program output.

This is equivalent to the abstract program:

```
letrec f(x) = ...
and g(x,y) = ...
and h(z) = ...
and s = 3
and msg = "Hi There"
and basevalue =.f(s)
and...
in g(msg,basevalue)
```

FIGURE 5-9
Handling of global definitions.

Figure 5-10 gives some particularly useful higher-order functions, with an example for each. They are so useful in practice that many real functional languages build them in. The first (and most famous) one, ***map,*** takes as its arguments some arbitrary function and some other arbitrary list of objects. The result is a new list of exactly the same length as the list argument, with the k-th element equaling the result of applying the function input to the k-th element of the input list. ***Map2*** is identical, except that it expects two equal length lists, and applies matching elements of both to the function argument.

A slightly different higher-order function is ***reduce***. Here there are three arguments: a function, an arbitrary object, and a list. If the list is not empty, the value returned is the result of applying the function argument to the first element of the list and to the object resulting from recursively applying **reduce** to the rest of the list. When the list finally empties, the value returned is the second argument. As an example, using "+" as the function input results in adding up all elements of the list, plus the original input for *t*.

A function very similar to ***map*** is ***vector***. Instead of applying a single function argument to all elements of a list of objects, this function applies each element of a list of functions to a single object and collects the results in a list.

```
map(f,x) = "apply function f to each element of list x"
         = if null(x) then nil
           else cons(f(car(x)), map(f,cdr(x)))
           e.g., map((λz|× 3 z),(3 5 7)) = (9 15 21)

map2(f,x,y) = "f applied to matching elements of lists x and y
            = if null( x) then nil
              else cons( f{car( x),car( y)}, map2( f,cdr( x),cdr( y)))
              e.g., map2((λxy| x+ y),(1 2 3),(4 5 6)) = (5 7 9)

reduce(f,t,x) = "recursively apply f to each x and prior result"
              = if null(x) then t
                else f(car(x),reduce(f,t,cdr (x)))
                e.g., reduce(+ ,0,(1 2 3)) = 1+(2+(3 +0)) = 6

vector(f,x) = "apply list of functions f to x"
            = if null(f) then nil
              else cons((car(f)) x, vector(cdr(f), x))
              e.g., vector(((λx|2× x) (λz|z−6) (λx|x)),3) = (6 −3 3)

while(p,f,x) = "while loop"
             = if p(x) then while(p,f,f(x)) else x
               e.g., cdr(while({λx|car(x)>0},
                               {λx|cons(car(x)−1, car(x)×cdr(x)},
                               (4.1))) = 4!

compose(f,g) = composition of functions f and g = (λx|f(gx))
  e.g., compose((λx|3×x), (λx|(4+x)) → (λx|3×(4+x))
```

FIGURE 5-10
Some higher-order fimctions.

An entirely different kind of higher-order function is ***while.*** This function takes two function arguments and a third object as an accumulating parameter. It operates much like a ***while loop*** in a conventional programming language. As long as the application of the first function argument (the loop's iteration test) to the accumulating parameter returns T, **while** recursively calls itself with the accumulating parameter modified by applying it to the second function argument (the loop body). When the result is F, the accumulating parameter is returned as the value. The example in the figure computes the factorial of a number.

The final higher-order function is ***compose.*** This function takes as its arguments two other functions, and produces as a result a new function which is the composition of the two.

5.9 AN EXAMPLE—SYMBOLIC DIFFERENTIATION

As an example of many of the above capabilities, we consider here the problem of writing a function that can take the symbolic derivative of an arbitrary mathematical expression. In addition to demonstrating techniques of expression writing, this example also serves as a good example of an expression where the input and output both could conceivably be treated as "code" of some sort.

```
dx/dx = 1

dy/dx = 0 (y ≠ x)

d(A+B+C+..)/dx = dA/dx + dB/dx + dC/dx + ...

d(A−B)/dx = dA/dx − dB/dx

d(A×B)/dx = (dA/dx)×B + A×dB/dx

d(A/B)/dx = ...
```

FIGURE 5-11
Rules for differentiation.

Figure 5-11 gives some of the basic rules for differentiating an expression with reference to some "mathematical" variable. The only difference from standard definitions is in the case of "+," which has been extended naturally to cover multiple operands. This is done deliberately to demonstrate the use of a higher-order function.

Although it is possible to describe a function that takes arbitrary infix notation expressions and produces infix outputs, things get quite messy quite fast (we would need to consider parentheses, precedence, etc.). Instead, we use prefix notation and s-expressions to produce a notation that is still readable but much easier to process. Figure 5-12 gives some syntax rules for this format; the meaning should be obvious.

This prefix s-expression format will show up again in the programming language LISP.

An abstract program for such differentiation is given in Figure 5-13. There are two recursive functions, **map2s** and **dif.** The latter accepts two arguments: the s-expression *e* to be differentiated, and the symbol *x* for

```
<symbolic-expression> := <number> | <variable>
    | ( <binary> <symbolic-expression> <symbolic-expression> )
    | ( + <symbolic-expression>+ )
    | ( <unary> <symbolic-expression> )

  <binary> := −|×|\|...

  <unary> := −|¬|sqrt|...

Example: (−x + sqrt(2×y) + x×y)/22 becomes

(/ (+ (− x) (sqrt(× 2 y)) (× x y)) 22)
```

FIGURE 5-12
BNF for s-expression form of integer expressions.

```
letrec map2s(f,x,y) = list of elements f(z,y) for each element z of list x''
  = if null(x) then nil
    else cons(f(car(x),y), map2s(f, cdr(x), y))

and dif(e,x) =
  if atom(e)
  then if e=x then 1 else 0
  else let op=car(e) in
    if op=``+''
    then cons(``+'', map2s(dif, cdr(e), x))
    else let left=cadr(e) and right=caddr(e) in
      if op=``−''
      then triple(op,dif(left,x),dif(right,x))
      else if op=``*''
        then triple(``+'',triple(``×'',dif(left,x),right),
                          triple(``×'',left,dif(right,x)))
        else ... expressions for other operators
      where triple(x,y,z) = cons(x,cons(y,cons(z,nil)))

in dif(...)

Example:
  Input: (infix notation) 3×x + x×y + (4−x)×z
  Input: (s-expression notation) (+ (× 3 x) (× x y) (× (− 4 x) z))
  Output: (s-expression notation) (+ (+ (× 0 x) (× 3 1)) (+ (× 1 y)
  (× x 0)) (+ (× (− 0 1) z) ((− 4 x) 0))
  Output: (infix notation) (0×x + 3×1) + (1×y + x×0) +
  ((0 − 1)×z + (4−x)×0)
```

FIGURE 5-13
Abstract program for symbolic differentiation.

the variable to drive the differentiation. The main function is recursive and has one local function (**triple**) of its own.

The main body of **dif** is a set of nested if-then-elses that determine what kind of expression the argument is. As usual, the first leg of the tests is the basis case (is *e* an atom?) whose T result is a nested test of whether or not that atom is the same as *x*. The else leg of this test covers all the recursive cases where *e* is not an atom, i.e., an expression with subexpressions that will have to be differentiated in turn. It starts with a let expression that picks off the first element of *e,* which in prefix notation must be the operator name. A series of tests then cover the different operators individually.

The first of these cases is for "+." Here the rule says to add up the derivatives of all the terms of the original sum. The approach chosen uses the recursive higher-order function **map2s,** which operates like map2 except that the second argument for the nested calls is a *scalar* object that does not change from call to call. The actual arguments given to **map2s** include **dif** itself, the list of sum terms to be differentiated, and the derivative symbol. The result is a list to which we add a "+" to recreate the whole result.

After "+," the operators involve exactly two operand expressions. A dual let expressions picks these out of *e* before proceeding. Then, in each case, the actual expression for the derivative uses the simple nonrecursive local function **triple** to create the appropriate results from derivatives of the subexpressions.

5.10 PROBLEMS

1. Write as an abstract program the predicate **free**(*x,e*), which returns true if the identifier *x* occurs free in the lambda expression *e.*
2. Write an abstract definition for a function **subs**(*y,x,m*), where *y* and *m* are any valid s-expressions and *x* is an atom representing a variable name. The result of the function should be an s-expression equivalent to [*y*/*x*]*m.* Assume that you have available a function **newvar**(), which at each call returns a new variable name that has not been used before. Your definition should detect and perform renaming properly.
3. Translate any of the abstract programs for **reverse** into pure lambda notation. Assume only the primitive functions **null, cons, car,** and **cdr.**
4. (Project) Write an abstract program that converts an abstract program back to pure lambda calculus. Use abstract syntax functions as required.
5. (Hard Project) Go the other way—from pure lambda calculus to some readable form of abstract programming.
6. What are the values bound to *x, y,* and *z* at each level of the following:

```
let x=4 in
  let y=x+2
  and z=x−3 in
    let x=y+x
    and y=z+3
    and z=y+6 in x+y+z
```

7. Show where to put the **Y** expression if one wanted to define a single recursive function using just the let expression.
8. Show that Figure 5-7 is correct.
9. In analogy to Figure 5-7 and Figure 5-8, show how to translate to pure lambda a letrec expression of the form "letrec **f**(*x*)=**A** and **g**(*x,y*)=**B** and *h*(*x,y,z*)=**C** in **E.**" Discuss how this generalizes to an arbitrary number of mutually recursive functions.
10. Convert the abstract program in Figure 5-13 to pure lambda calculus form. (You need not translate **car, cdr, cons, atom,** =, or character representations for "+," "-," "*," ....)
11. Write an abstract program to optimize arithmetic expressions constructed according to Figure 5-12. Use optimization rules of the form:

"(× 0 A)" is 0

"(× A 1)" is A

"(+ A 0)" is 0

An expression involving only numbers can be reduced to a number.

You may assume that the predicate **number**(x) returns true if x is a number and false otherwise.

12. Rewrite **reduce** to use an accumulating parameter. You may redefine **reduce** to "parenthesize" the other way if you wish.
13. Define and write in abstract syntax an **until** function modeled after the **while** function described in the text.
14. Complete Figure 5-13 to include division and the unary operators "−" and "**sqrt.**"

CHAPTER 6

SELF-INTERPRETATION

Lambda calculus is powerful enough to express any computable function, and is thus as powerful as any other computing language. Given this, it is interesting to ask if the execution model of lambda calculus is itself expressible as a computable function. If it is, then we have the possibility of writing in lambda calculus an interpreter for itself. Such an interpreter would express the semantics of lambda calculus in itself (and thus for any language that lambda calculus supports—such as abstract programming).

As hinted at several times, and perhaps partially demonstrated by the differentiator evaluator of the last chapter, the answer to this is a resounding yes. We can write a lambda calculus function, usually called ***eval,*** which when given any arbitrary lambda expression **E,** reduces it as far as possible, and returns the result. In particular, since **eval(E)** (before reduction) is itself a valid lambda expression, we are perfectly free to try **eval(eval(E))**, that is, have **eval** determine what happens when **eval** itself is turned loose on an expression. The answer returned from **eval(eval(E))** is the same as that from **eval(E)**, and both are equivalent to the reduced form of **E. Eval** thus forms a completely valid ***interpreter*** for lambda calculus, expressible in lambda calculus.

The first few sections in this chapter describe **eval** using abstract syntax functions for the syntax parsing of generic lambda expressions. Two versions are given, one which performs normal-order evaluations, and one which performs applicative-order evaluations.

After this we will introduce a particular concrete syntax for lambda calculus using s-expressions and prefix notation similar to that for the input to the differentiator example of the last chapter. The abstract syntax functions in **eval** will be rewritten in terms of the appropriate s-expression operators, and **eval** will be modified as necessary.

There are good reasons for this approach. First, it approaches the syntax of the most popular function-based language, LISP. More important, however, the notation is close enough to something that is implementable on conventional computers to consider using it as a ***bootstrap*** process to bring up a real function-based programming system. Having once written a machine-language version of this **eval** function, we can then write a modified **eval** on top of it which has all sorts of extra features (abstract programming syntax, built-in functions, input/output, error handling, etc.), which in turn could support **eval**s for languages with even more powerful programming features.

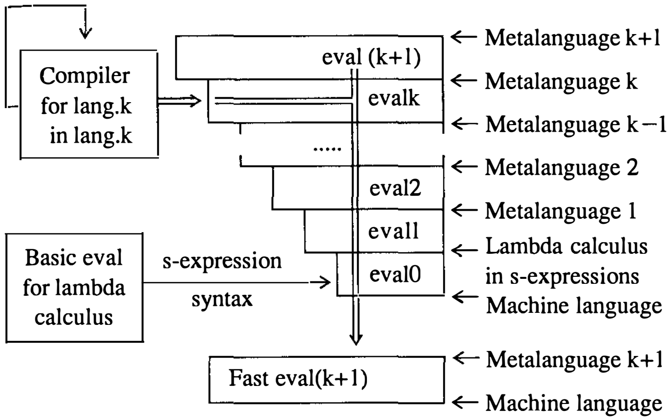
Such languages are often called ***metalanguages,*** and the **eval**s which support them, ***metainterpreters.*** When given a program in the highest such language, the operation of such a system corresponds to the lowest-level interpreter simulating the execution of the first metainterpreter as it simulates the next metainterpreter simulating the one above it, etc., until the final metainterpreter simulating the given program is reached. Although obviously this is potentially a very slow process, this approach is used frequently in the computer science community to quickly investigate the potentials of new programming languages.

We should note that this cascading of interpreters can be sped up quite easily by picking some particular level of metalanguage and writing a compiler for it in itself. As pictured in Figure 6-1, this compiler can then be fed to the metainterpreter for that language, with a copy of itself used as input to the compiler. The result is a compiled version of the compiler, which now can be used to compile the next-higher metainterpreter (or even a compiler for that metalanguage).

Another reason for investing time in an s-expression form of **eval** is that it gives us an excellent vehicle for describing orders of evaluation which are combinations and extensions of both pure applicative- and pure normal-order reduction. Such approaches offer some extremely interesting capabilities, and will be investigated in a later chapter.

A related topic in this chapter discusses a variation to the design of such interpreters that packages the information needed for substitutions into ***association lists,*** and defers actual substitution until the last possible moment. Because of obvious efficiency gains and the match of such structures to objects that can be built into conventional computers, this style of interpreter is used in many real systems for real functional languages, and will be the basis for discussions in later chapters.

Finally, this chapter also discusses how to "package" the process of evaluating an expression in midstream, freeze it, pass it around as an ordinary object, and then restart it at some later time. The package is called a ***closure*** and forms the basis of quite a bit of the advanced language techniques discussed later in this book. Landin (1963) is one of the earliest references to this subject area. Henderson (1980) is another good reference, particularly Chapter 4.



Let "P_n" be a program written in metalanguage n:

$\text{eval}_0(\text{eval}_1(\text{eval}_2(\ldots\ \text{eval}_n(P_n)..) \rightarrow \text{result}$

Assume compiler_k = compiler program for metalanguage k written in metalanguage k.

$\text{eval}_0(\text{eval}_1(\ldots\text{eval}_k(\text{compiler}_k(\text{compiler}_k)..) = {}^*\text{compiler}_k$

Now $^*\text{compiler}_k(\text{eval}_{k+1}) \rightarrow {}^*\text{eval}_{k+1}$

---- A machine-language-speed interpreter for metalanguage k+1

FIGURE 6-1

Metainterpreters, languages, and compilers.

6.1 ABSTRACT INTERPRETERS

The first version of **eval** interprets pure lambda expressions as defined by the original syntax of Chapter 5. This is the syntax without any of the "simplifications" of multiple arguments, reduced "( )," etc. To make this **eval** simple, we will use the ***abstract syntax*** functions listed in Figure 6-2. It should be obvious that such functions could be written in many different programming languages without much difficulty but are largely uninteresting.

For historical reasons, our first version of this interpreter will actually consist of three mutually recursive functions: ***eval, apply,*** and ***subs.*** **Eval**(E) does as advertised: It reduces the arbitrary lambda expression **E** as far as possible. **Apply**(f,a) takes two lambda expressions, f and a, and treats f as a function to which a should be given as its argument. **Subs**(a,x,e) performs the substitution of a for all free instances of the identifier x in the expression e, (i.e., $[a/x]e$).

Figure 6-3 lists these functions. They are all mutually recursive. As

Syntax summary:

```
  <expression> := <identifier> | <function> | <application>
  <function> := (λ<identifier>"|" <expression>)
  <application> := (<expression> <expression>)
```

Abstract predicates: Assume E an arbitrary lambda expression

- is-id(E): true if E an identifier
- is-function(E): true if E a lambda function
- is-application(E): true if E an application

Abstract selectors: Assume E a lambda expression

- get-function(E): get function from application E
- get-argument(E): get argument from application E
- get-id(E): get identifier from function E
- get-body(E): get body expression from function E

Abstract creators: Create an expression

- create-function(x,E): create (λx|E)
- create-application(A,B): create (A B)
- new-id( ): return a guaranteed unique identifier symbol

Examples:

- is-id(((λx|(xy))A)) → F
- is-function(((λx|(xy))A)) → F
- is-application(((λx|(xy))A)) → T
- get-argument(get-body(get-function(((λx|(xy))a))))
  → get-argument(get-body((λx|(xy))))
  → get-argument((xy))
  → y
- let z = new-id( ) in
  create-application(create-function(x,((xy)y)), z)
  → create-application((λx|((xy)y)), z)
  → ((λx|((xy)y))z

FIGURE 6-2
Abstract syntax functions for **eval.**

discussed previously, there is an implied letrec at the beginning, and ands coupling them.

Eval has three cases corresponding to the three forms of a lambda expression. The first handles identifiers by simply leaving them alone. The second handles expressions that are pure functions by recreating the function, but with its body fully reduced. The final case handles applications by selecting out the function and argument subexpressions and passing them to the function **apply.**

Apply handles the reduction of applications. As with **eval,** this function divides into three subcases corresponding to the internal structure of the expression passed as the function. If it is a simple identifier, the only reductions possible are to the argument, and what is returned is the original application put back together again (by the **create-application** function), but with the argument evaluated. If the function part f is itself an application, it is evaluated first before performing the application to the argument a. Finally, if f is a pure lambda function, then the application is performed by substituting the argument a for the binding variable for f into the body of f.

```
eval(e) = "evaluate expression e as far as we can" =
          if is-id(e)
          then e
          else "either a function or application"
               if is-function(e)
               then create-function(get-id(e), eval(get-body(e)))
               else apply(get-function(e), get-argument(e))

apply(f,a) = "normal-order application" =
          if is-id(f)
          then create-application(f,eval(a))
          else "f an application itself or a function"
               if is-application(f)
               then apply(eval(f),a)
               else eval(subs(a, get-id(f),get-body(f)))

apply(f,a) = "applicative-order application" =
          if is-id(f)
          then create-application(f,eval(a))
          else let b=eval(a) in "evaluate argument first"
               if is-application(f)
               then apply(eval(f),b)
               else eval(subs(b, get-id(f), get-body(f)))

subs(a,x,e) = "substitute a for x in e" =
      if is-id(e); see if Rule 1
      then if e=x then a else e
      else if is-application(e); see if Rule 2
           then create-application(subs(a,x, get-function(e)),
                                   subs(a,x,get-argument(e)))
           else let y=get-id(e) and c=get-body(e) in
                if y=x then e; Rule 3a
                else; always Rule 3c—rename binding variable
                     let z=new-id() in
                         create-function(z, subs(a,x,
                         subs(z,y,c)))
```

FIGURE 6-3
Abstract interpreter.

There are two versions given for **apply,** one for ***normal-order reduction*** and one for ***applicative-order reduction.*** The former substitutes the unevaluated argument into the function body; the latter evaluates the argument first. Note also that in the latter case we could, if we wished, reduce the function body before the substitution.

Figure 6-4 gives an example of a normal-order evaluation by this interpreter of the expression ((T *a*) *b*).

There is one subtle problem in the above version of **apply.** If *f* is an application of the form (*xy*), where *x* is a simple identifier, then **apply** will call itself with *f* replaced by **eval**(*xy*), which will end up returning (after another call to **apply**) (*xy*) unmodified. **Apply** will be caught in an infinite loop. It is left up to the reader to describe the relatively simple fixes needed to avoid this.

Finally, the function **subs** performs the substitution process spelled out in the previous chapter, with one exception. When the body *e* is itself a function and the binding identifier *y* of this nested function is different from *x*, the substitution rule calls for a check if *y* occurs free in *a*, and if so, the renaming rule is invoked. For simplicity, the **subs** function given here always renames *y*; the function **new-id** provides a unique new identifier each time it is called. It is left as another exercise to the reader to correct it to rename only when necessary.

```
eval((((λx|(λy|x)) a) b))
→ apply(get-function((((λx|(λy|x)) a) b)),
        get-argument((((λx|(λy|x)) a) b)))
→ apply(((λx|(λy|x)) a), b)
→ apply(eval(((λx|(λy|x)) a)), b)
→ apply(apply((λx|(λy|x)), a), b)
→ apply(eval(subs(a, get-id((λx|(λy|x))), get-body((λx|(λy|x))))), b)
→ apply(eval(subs(a, x, (λy|x))), b)
→ apply(eval((λz|a)), b)
→ apply(create-function(get-id((λz|a)), eval(get-body((λz|a))), b)
→ apply(create-function(z, eval(a)), b)
→ apply(create-function(z, a), b)
→ apply((λz|a), b)
→ eval(subs(b, get-id((λz|a)), get-body((λz|a))))
→ eval(subs(b, z, a))
→ eval(a)
→ a
```

FIGURE 6-4
Example of normal-order **eval.**

6.2 LAMBDA EXPRESSIONS AS S-EXPRESSIONS

The introduction to s-expressions in Chapter 2 emphasized their use for data structures. The symbolic differentiation example earlier in this chapter introduced the idea of using s-expressions in a prefix notation. This section takes the idea one further, and gives lambda calculus a ***concrete syntax*** by showing a conversion between its original syntax and s-expressions. The result is a notation for which we can describe precisely the abstract syntax functions of Figure 6-2 and which begins to approach the actual syntax and interpretative model of LISP.

6.2.1 Pure Lambda Calculus and s-Expression Form

Figure 6-5 gives the conversion between the BNF forms of lambda expressions and equivalent s-expressions. It also includes a diagram of the cell representation for a simple example. The result is a notation where a single atom by itself is an identifier, and an expression in parentheses is either a function or an application. In the former case the **car** of the expression is the keyword lambda; in the latter case it is the expression to be treated as a function, with the **cadr** of the total expression its argu-

Lambda expression <l-expr> ⟹ s-expression <s-expr>:

<identifier> ⟹ <identifier>

(λ<identifier>|<l-expr>) ⟹ (lambda (<identifier>) <s-expr>)

(<l-expr><l-expr>) ⟹ (<s-expr> <s-expr>)

Example: (((λx| (λy| x)) a) b)
⟶ (((lambda (x) (lambda (y) x)) a) b)

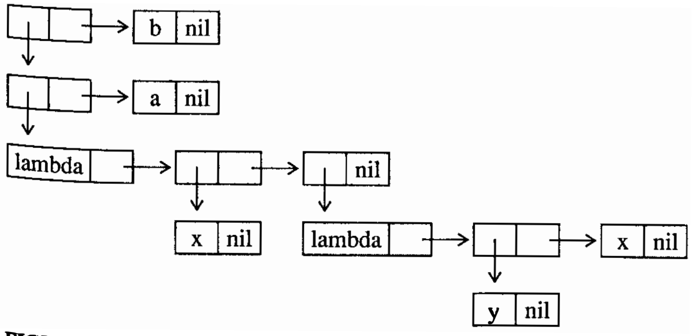


FIGURE 6-5
Lambda expression translation into s-expressions.

ment expression. In either case, looking at the **car** of the expression tells us immediately what kind of expression it is. This looks like a ***prefix notation,*** considerably simplifying a real interpreter specification. In fact, replacing the abstract syntax functions of Figure 6-3 by those listed in Figure 6-6(*a*) gives a complete set of functions for a lambda language in s-expressions. Figure 6-6(*b*) gives the converted form of the **subs** function.

6.2.2 Multiple Arguments, Constants, and Built-ins

The particular conversion process shown above was chosen because it supports cleanly several of the notational simplifications discussed earlier for the original lambda calculus format. First, multiple arguments can be handled both in function definitions and in applications. In the former, the list object following lambda can include as many variable names as there are arguments, as in (λ*xy*|*y*(*xx*)) ⇒ (lambda (*x y*) (*y* (*x x*))). Note that each such variable is a separate element in the embedded list (*x y*).

In conjunction with this, an s-expression application can be extended to have more than two list elements, with the first representing the function as before and the rest of the list representing the available argument values. The order of the argument expressions in the list matches the order of identifiers in the formal argument list of the function. For example:

(λ*xy*|*y*(*xx*)) *w z* ⇒ ((lambda (*x y*) (*y* (*x x*))) *w z*)

Note that a cell representation of this is different from that of Figure 6-5.

```
Abstract function → s-expression equivalent
  is-id(e) → atom(e)
  is-function(e) → eq(car(e),lambda)
  is-application(e) → not(atom(e)) and not(is-function(e))
  get-function(e) → car(e)
  get-argument(e) → cadr(e)
  get-id(e) → caadr(e)
  get-body(e) → caddr(e)
  create-function(id,body) → list(lambda, list(id), body)
  create-application(f,a) → list(f, a)
```

(*a*) s-Expression function equivalents.

```
subs(a,x,e) = "substitute a for x in e—all s-expressions"
  if atom(e); see if Rule 1
  then if eq(e,x) then a else e
  else if not(atom(e)) and not(is-function(e)); see if Rule 2
      then list (subs(a,x, car(e)),
              subs(a,x,cadr(e)))
      else let y=caadr(e) and c=caddr(e) in
          if y=x then e; Rule 3a
          else; always Rule 3c—rename binding variable
              let z=new-id() in
                  list(lambda, list(z), subs(a,x, subs(z,y,c)))
```

(*b*) Substitution of s-expressions.

FIGURE 6-6
Abstract syntax functions for s-expression notation.

Next, numbers, booleans, character strings, etc., can all be used as ***constants*** to convey their standard meaning. In this form, they need only be written in their normal textual form without expansion to the more complex pure lambda expression form.

To go with this, we can expand the allowable types of expressions that can be used in the first element of an application to include common functions such as "+," "×," "**and**," "**or**," "**car**," "**cdr**," "**cons**," etc. Then, either **eval** or **apply** can be expanded to include specific tests for these ***built-in functions,*** and perform the appropriate operations on the argument values. Thus (**cons** (+ 3 4) nil) should evaluate to the list (7).

Note that in such cases we probably want some sort of applicative-order evaluation to reduce the arguments before applying the function. Consider the complexities resulting if we did not, and were asked to evaluate (× (+ 3 4) (− 8 6)) The code for "×" would have to handle unevaluated applications as arguments—a large impact on performance.

6.2.3 Other Special Forms

Just as lambda in the **car** of a list serves as a keyword indicating a lambda function, we can extend our s-expression form of the language to cover many of the other convenient notations from abstract programming, such as let-and-in, if-then-else, etc. When expressed as an s-expression with a special keyword in the **car** position of the list, such notation are called ***special forms.*** Figure 6-7 summarizes this s-expression syntax, and Figure 6-8 gives a sample expression that includes many of the features.

let AND letrec FORMS The let and letrec notations of abstract programming translate to s-expressions by a list of the form:

(let (⟨identifier⟩*) (⟨s-expression⟩*) ⟨s-expression⟩)

Thus let *x*=**A** and *y*=**B** and *z*=**C** in **f**(*x, y, z*) is equivalent to

(let (*x y z*) (**A B C**) (**f** *x y z*))

The second list element (accessed by **cadr** on the whole list) follows the multiple-argument notation introduced earlier. The names of all iden-

```
<id-list> := (<identifier>*)

<expr-list> := (<expr>*)

<builtin> := + | − | × | / | car | cdr | cons | atom | null ...

<expr> := <number> | <nil> | <identifier>
            | <expr-list>
            | (lambda <id-list> <expr>)
            | <special-form>

<special-form> := (<builtin> <expr>*)
                   | (let <id-list> <expr-list> <expr>)
                   | (letrec <id-list> <expr-list> <expr>)
                   | (if <expr> <expr> <expr>)
                   | (cond (<expr> <expr>)*)
                   | (quote <expr>)
```

FIGURE 6-7
s-Expression form of abstract program syntax.

```
letrec fact(n) =
  if n=0
  then 1
  else let m=n−1 in n×fact(m)
and sum(n,a) =
  if n=0
  then a
  else sum(n−1,n+a)
in sum(fact(3),0)
```

(*a*) Abstract form.

```
(letrec (fact sum)
      ( (lambda
         (n)
         (cond ((= n 0) 1)
               (T (let (m) ((− n 1)) (× n (fact m))))))
       (lambda
         (n a)
         (if (zero n )
           a
           (sum (− n 1) (+ n a)))
 (sum (fact 3) 0)))
```

(*b*) s-Expression form.

FIGURE 6-8
Sample conversion to s-expression form.

tifiers being bound to values are included here as elements of a list. Thus, the first element in this sublist is the variable name after the let; the following identifiers follow any coupled ands.

The third list element here (accessed by a **caddr**) is the list of expressions to be substituted for the list of identifiers (**cadr**). The length of the list of expressions and identifiers should match. The expressions themselves are the transformed right-hand sides of the let definitions.

The final element of the original list (accessed by **cadddr**) is the expression into which all these variables are substituted. This corresponds to the body expression following the in in the abstract program form.

To handle this special form, either **eval** or **apply** (usually **eval** in this case) can be modified. It will test the **car** of a list for let and perform the appropriate substitutions. Either applicative- or normal-order evaluations of the arguments are possible.

if-then-else AND cond FORMS Abstract program forms of the if-then-else sort can be included as s-expressions by special forms of the form

(if ⟨s-expression⟩ ⟨s-expression⟩ ⟨s-expression⟩)

where if is a keyword detected by **eval,** and the first expression is evaluated for a true or false value. In the former case, the second expression (**caddr**) is then passed to **eval.** In the latter, the third (**cadddr**) is so handled.

Although either applicative- or normal-order evaluation is possible here, normal order makes more sense. We evaluate the then or else expressions only after evaluating and testing the expression following the if.

A generalization of this form neatly handles nested if expressions of the form if...then...elseif...then...elseif...then...else.... Here the **car** is cond (for ***conditional***), and the rest of the list is two-element lists:

(cond (⟨s-expression⟩ ⟨s-expression⟩)*)

The interpretation of this is that when **eval** finds an expression with cond as its **car,** the evaluation process involves taking the first element of the remaining list and evaluating the **car** of this list. If the result is true, then the **cadr** of this element is evaluated and returned. If the result is false, the next element of the original expression is evaluated.

If none of the pairs has a **car** that evaluates to T, we will assume that nil is returned. To avoid this default, the last pair of expressions should have its first expression equaling T to force evaluation of the second. This corresponds to the last else case in the chain.

An alternative form of the cond that is sometimes used deletes the T in the last expression, leaving something of the form (⟨expression⟩). If none of the pairs in front of it is true, the embedded expression in such a last term is always evaluated and its value returned.

STOPPING INTERPRETATION—QUOTE There are times when a programmer wants to prevent an expression from being evaluated. The expression is to be treated as data, and not as a piece of program text to be reduced. This is particularly common in languages where the syntax for programs looks like s-expressions, with little or no distinction between program and data.

A typical example of this is in the writing of one of the interpreters described in this chapter in real code. In such programs there are many tests for equality between some evaluated expression and the symbolic form of program keywords, such as in if *e*=lambda, or in expressions to test if a function is a "built in," as in **member(***e*, **(cons car cdr**...)). Evaluating the latter with a direct extension of the interpreters to date might reduce (**cons car cdr**...) to (**car cdr**) or worse. This is not at all what is intended.

The popular solution to such problems is to introduce a special function ***quote,*** which when evaluated returns its single argument totally unevaluated (i.e., neither normal nor applicative evaluation). Thus (if (eq *e (quote lambda))...) or* (**member** *x* (quote (cons car cdr...) would work as desired.

6.3 AN EXPANDED INTERPRETER

Figures 6-9 and 6-10 give a version of **eval** that assumes the syntax of Figure 6-7 and thus has all of the following features:

- Interpreted language is in s-expression format.
- Functions may have multiple arguments.
- Support for "built-in" functions such as "+," **car,** etc.
- A mix of normal- and applicative-order reduction.
- Support for special forms such as let, letrec, if, cond, etc.

The result is that this new **eval** supports a function-based language with semantics that is very close to pure lambda calculus, but with the syntatic sugar of abstract programming, a representation that permits easy implementation on conventional computers, and efficiency hooks that speed up certain standard calculations.

Both normal- and applicative-order reduction are used in this interpreter. The typical processing of lambda expressions is via normal order. This greatly simplifies the handling of recursion. Applicative order, however, is used in several places, particularly when dealing with "built-in" functions such as "+," "**car,**"....The arguments to such functions are reduced fully before reaching the function. While this increases performance, it does mean that we cannot curry built-in functions—all arguments to them must reduce to the appropriate type of object.

The s-expression notation used here is the same as that described in the last section, with one limitation. For simplicity, expressions with prefix of letrec are permitted to make exactly one local definition. This is to

```
eval(e) = "reduce s-expression e"
  if atom(e)
  then e "Nonlists are fully reduced"
  else let fcn=car(e) and args=cdr(e) in
      if atom(fcn)
      then "function term is built-in or special form"
          if fcn=quote then car(args)
          elseif member(fcn, builtins)
          then apply-builtin(fcn, args)
          elseif fcn=lambda "look for special forms"
          then list(lambda, car(args), eval(cadr(args)))
          elseif fcn=if then "see which expr to eval"
               if eval(car(args))=T
               then eval(cadr(args))
               else eval(caddr(args))
          elseif fcn=cond then eval-cond(args)
          elseif (fcn=let) or (fcn=letrec)
          then let ids=car(args) "list of identifiers"
              and vals=cadr(args) "matching value list"
              and body=caddr(args) in
              if fcn="let"
              then subs2(vals, ids, body)
              else "a letrec expression"
                apply(list(lambda,ids,body),
                      list(Y, list(lambda, ids, vals))
                where Y=(lambda (y) ((lambda (x) (y(xx)))
                                     (lambda (x) (y(xx))))
          else "must be an identifier"
             apply(fcn, args)
   else apply(fcn ,args) "function is a list"

apply(f,a) = "a normal-order application"
  if atom(f)
  then cons(f,map(eval,a))
  elseif car(f)=lambda
  then "f is a lambda expr—see if curry"
      let formal=length(cadr(f))
      and actual=length(a) in
          if formal=actual
          then eval(subs2(a, cadr(f),caddr(f)))
          else list(lambda,
                    drop(actual, cadr(f)),
                    eval(subs2(a, cadr(f),caddr(f))))
               wherërec drop(count, x) =
               if count=0 then x else drop(count−1,cdr(x))
  else apply(f, a)
```

FIGURE 6-9
An expanded interpreter.

```
apply-builtin(f,a) = "an applicative-order evaluation"
  let args=map(eval, a) in "args = list of evaluated arguments"
      if f="car" then caar(args)
      elseif f="cdr" then cdar(args)
      elseif f="+" then car(args)+cadr(args)
      elseif ...

eval-cond(a) = "special evaluation for cond"
  if null(a) then nil
  else "iterate thru the list of pairs"
        let z=car(a) in
            if eval(car(z))=T
            then eval(cadr(z))
            else eval-cond(cdr(a))

subs2(v,n,e) = "substitute v's for n's in e"
  if atom(e) "if e variable, replace"
  then lookup(e,n,v)
  elseif car(e)=lambda
  then "substitute into a lambda expression—rename"
        let old=cadr(e) in "cadr=id list"
            let new=map(new-id,old) in
            list(lambda,new, subs2(v, n, subs2(new,old,caddr(e))))
  else "e is an application—substitute in all parts"
  map((λz|subs2(v,n,z)), e)

lookup(z,n,v) = "find z in n and replace by value in v"
  if null(n) or null(v)
  then z
  elseif car(n)=z then car(v)
  else lookup(z,cdr(n),cdr(v))
```

FIGURE 6-10
Support functions for expanded interpretations.

avoid for now the complications described earlier for mutually recursive functions.

The major differences between Figure 6-9 and Figure 6-3 are in the handling of multiple arguments and in applications involving ***special forms*** (where "keywords" are prefixes). Syntatically, multiple actual arguments for a standard lambda application are represented as multiple elements in an s-expression list following the prefix element [a function of the form "(lambda (...) (...))"]. They are handled by expanding the **subs** function into **subs2,** which "simultaneously" replaces all the occurrences of identifiers from the lambda function by matching value expressions from the application list. Internally, **subs2** recursively processes the lambda function's body expression one element at a time, using the function ***lookup*** each time an element which is a variable is found.

Currying is detected when the number of actual arguments is less that the number of formal identifiers in the lambda expression. In this case, only those identifiers with matching variables are replaced, and the rest are left as identifiers for a new lambda expression.

Special forms are handled by special if-then-elseif tests in **eval.** "Built-in" functions are handled as described above. All the arguments are evaluated, and then code corresponding to the function is executed.

For the keyword prefix lambda, the evaluation process simply evaluates the body of the function. Note that we are evaluating this body only if a request was made to evaluate the function to begin with. This is in tune with the rules of normal-order reduction.

For if, the evaluation procedure takes the first argument, the test expression, and evaluates it. On the basis of the resulting value, either the second or third argument is evaluated. This is a mix of applicative- and normal-order evaluation.

Evaluation of cond is handled similarly. The arguments in this case are pairs of expressions, the **car** of which are evaluated sequentially until one is found whose value is true. Then its matching **cdr** is evaluated. As with built-ins and if, these **car** tests must be fully evaluatable.

Expressions containing let are translated exactly as the abstract-language version was translated earlier in this chapter. A new function expression is created, with arguments represented by the list in the **car** of the let expression. This function is passed to **apply** along with the second element of the argument list, which itself is a list of argument values for the specified identifiers.

Evaluation of letrec is similar but more complex. For simplicity, the version shown in Figure 6-9 handles properly letrecs with at most one identifier given a recursive definition. It also assumes that this definition is itself a lambda expression (as converted into s-expression notation). As with let, the processing generates a new function expression (again in s-expression notation) and passes it to **apply.** In this case, however, the argument expression for **apply** is taken apart and put back together with an extra identifier. This is the name for the recursive function. It is then made into an application with a prefixed **Y** expression. Again, this is exactly as described for abstract programs.

Note that if we had wanted to, we could have added **Y** as a special form to **eval** which performed the same operations without the substitutions.

Figure 6-11 gives a partial trace of the evaluation of the factorial function. Most of the major parts of **eval** and **apply** are executed here.

6.4 ASSOCIATION LISTS

All the lambda interpreters discussed up to now have employed brute-force substitution to handle applications. The argument expression (either evaluated or unevaluated) is substituted in total for every free occurrence of the binding variable in the function's body. This requires

```
eval(((lambda (x y) (+ x (× 4 x))) (+ 3 5) 6))
  → apply((lambda (x y) (+ x (× y x))), ((+ 3 5) 6))
  → eval(subs2((( + 3 5) 6), (x y), (+ x (× y x))))
  → eval(map((λz|subs2(((+ 3 5) 6), (x y), z), (+ x (× y x))))
  → eval(( + (+ 3 5) (× 6 (+ 3 5))))
  → apply-builtin( +, ((+ 3 5) (× 6 (+ 3 5))))
  → let args = map(eval,((+ 3 5) (× 6 (+ 3 5)))) in ...
  → let args = list(eval(( + 3 5)), eval(( × 6 (+ 3 5)))) in ...
  → ...
  → let args = (8 48) in ...
  → car((8 48)) + cadr((8 48))
  → 8 + 48 → 56
```

(*a*) A simple example.

```
eval( (letrec (f) {lambda (f) (lambda (n) (cond (zero n) (T (× n (f (− n 1))))))})) (f 3)))
  → apply( (lambda (f) (f 3)), (Y (lambda (f n) (cond ...))))
  → eval(subs2((f 3), (f), (Y ...)))
  → eval(((Y ...) 3))
  → ...
  → eval(({lambda (f n) (cond ...)} {Y...} 3))
  → apply(eval({lambda (f n) (cond ...)}), ({Y...} 3))
  → apply({lambda (f n) (cond ...)}, ({Y...} 3))
  → eval(subs2((cond...), (f n), ({Y...} 3)))
  → eval((cond ((zero 3) 1) (T ...)))
  → eval-cond({((zero 3) 1) (T ...)})
  → if apply-builtin(zero 3) then 1 else ...
  → ...
```

(*b*) A version of factorial.

FIGURE 6-11
Sample partial interpretations.

essentially scanning a function's body twice, once to find all the free occurrences and do the substitutions, and once to find the next application to attempt.

While conceptually simple, such an approach suffers from obvious inefficiencies when the function's body contains (as most do) one or more nested if-then-else structures, or when there are multiple arguments. More passes may be needed (up to two per argument), and part, if not most, of the resulting substitutions may be wasted effort (due to then-else cases that are not used).

One alternative to such substitutions is simply to remember at the time of an application which identifiers are to get which values, and then do the substitution on an identifier by identifier basis when the function's body is scanned for the next application. For example, in the application

(λ*wxyz*|**E**) **W X Y Z**

we wish to remember the four-way simultaneous substitution [**W**/*w*,**X**/*x*,**Y**/*y*,**Z**/*z*]. This pairing is often called the ***context, binding,*** or ***environment*** under which the expression **E** is to be evaluated or reduced.

The easiest data structure in which to record such a substitution is an ***association list,*** or ***alist,*** which is created at the time an application is encountered and which, in some way, pairs up identifier names and the values assigned to them via applications. Such a data structure is passed as required between **apply** and **eval** when various parts of the function's body are actually evaluated. Finding an identifier in such an expression should cause **eval** to look up the identifier in the alist and replace it by the identifier's matching value. Thus both **eval** and **apply** must have their definitions extended to include extra arguments that pass these alists.

There are two major forms that an alist for a single application typically takes. First, it can be a single s-expression list whose elements are **cons**ed pairs of names and matching values. Note that the order of each pair is backward from that of the "[/]" notation; the reason is efficiency in many real implementations.

For the previous example, the s-expression form of the alist would be of the form ((*w*.**W**) (*x*.**X**) (*y*.**Y**) (*z*.**Z**)).

Second, the alist can be two lists of the same length, one for the identifier names and one for the matching argument values, such as (*w x y z*) and (**W X Y Z**), respectively. This resembles usage of the *n* and *v* arguments in earlier **apply**s. The former list is often called the ***name list*** and the latter, the ***value list.*** Figure 6-12 diagrams examples of both notations.

Although conceptually simple, the use of either form does require solutions to several problems, including:

1. Multiple formal arguments in a lambda function
2. Nesting of applications
3. Recursion
4. Free variables in either the function body or the arguments
5. Handling of name clashes and renaming
6. Normal-order reduction versus applicative-order reduction

Assume: (let (x y z) (1 2 (/ 9 3)) (× (+ x y) z))

When evaluating (× (+ x y) z):

- alist = ((x.1) (y.2) (z.3))
- name list = (x y z)
- value list = (1 2 3)

FIGURE 6-12
Sample notations for association lists.

Solving the first problem was discussed above. Multiple name-value pairs in the alist match precisely the multiple simultaneous substitutions that correspond to a function applied to multiple arguments.

The second problem, nesting of multiple applications, can also be handled fairly easily by augmenting the alist. Again two approaches are possible. The first simply appends the new pairs onto the front of the prior alist. Searching for a substitution involves scanning down this list until a pair is found where the name matches the identifier in the expression. While the approach is simple, it does lose insight into which application generated which substitution.

The second approach makes the alist a list of lists, each of which in turn corresponds to the substitutions called out by a single application. Each nested application then stacks a new list onto the front of the old one. Locating a value thus involves looking backward through this list of lists, one sublist at a time. The first occurrence of the desired identifier gives the appropriate value.

6.4.1 A Simple Associative List-Based Interpreter

The other problems are somewhat more difficult to understand and are best demonstrated by poking holes in a simple interpreter that uses alists as described above. Figure 6-13 diagrams such an **eval** and **apply.** They assume only the pure lambda calculus (in s-expression format) as inputs; only one argument (no multiple arguments), no built-in functions, constants, or special forms are assumed. Both functions resemble those from Figure 6-3 but do not use the **subs** function or its equivalent. Instead they both have an extra argument representing pairs of names and values for all the applications still in force at the current point in the evaluation process. Thus when **eval** encounters an identifier, it calls a function ***assoc,*** which will look through the alist for the identifier. If the identifier is not the **car** of any pair, this function returns nil. If it is there, it returns the pair of name and value. (This permits us to distinguish between an identifier that is not in an alist and one that is, but bound to nil).

When **eval** receives a match from **assoc,** it takes the value (the **cdr** part of the pair) and reevaluates it. This is to take care of circumstances where there is a chain of substitutions. (Consider, for example, the alist ((*z*.1) (*q*.2) (*y*.(+ *z q*)) (*x*.(× *y z*))) when the value of *x* is desired).

Apply evaluates applications by building a new (name.value) pair, appending it to the front of the current alist, and then passing the body of the function back to **eval** with the augmented list as the new context.

There are some severe problems with this simple-looking set of functions, many of which revolve around situations where the body of a function itself contains a function.

The first example of this is the simple expression

((lambda (*y*) (lambda (*x*) (+ *x y*)) 4)

```
<lambda-expr> := <identifier>
               |  (lambda (<identifier>) <lambda-expr>)
               |  (<lambda-expr> <lambda-expr>)

eval(e,alist) = ''evaluate e with context alist''
  if atom(e)
  then let z=assoc(e,alist) in
         if null(z)
         then e
         else eval(cdr(z), alist)
  elseif car(e)=lambda
  then e
  else apply(car(e),eval(cadr(e),alist),alist)

apply(f,a,alist) = ''apply f to argument a with context alist''
  if atom(f) ''replace a ''variable'' function by its value''
  then let z=assoc(a,alist) in
         if null(z)
         then cons(f, a)
         else apply(cdr(z),a,alist)
  elseif car(f)=lambda
  then eval(caddr(f), ((caadr(f).a).alist)
  else apply(eval(f,alist),a,alist )

assoc(a,alist) = ''find (a.''value'') in alist''
  if null(alist) then nil
  elseif caar(alist)=a then car(alist)
  else assoc(a,cdr(alist))
```

FIGURE 6-13
A simple interpreter using association lists.

The result should be (lambda (*x*) (+ *x* 4)). **Eval,** however, will pass control to **apply,** which in turn will call

eval((lambda (*x*) (+ *x y*)), ((*y*.4)))

This look good, but **eval,** as stated above, will return as its result its first argument unmodified (the argument is a lambda expression). The context ((*y*.4)) is lost.

One solution would be to replace the elseif line of **eval** by

elseif **car**(*e*)=lambda then **list**(lambda, **cadr**(*e*), (**eval**(**caddr**(*e*),*alist*))

This evaluates the body of the function, and would replace the *y* above by the appropriate 4, but would have its own problems. The expression **eval**(((lambda (*y*) (lambda (*y*) (*y y*))) 4), nil) would result in (lambda (*y*) (4 4)). This time we should have avoided the substitution because of the name clash.

Again there is a direct solution to this, namely, rename the inner function's binding variable. This renaming can be done by augmenting the alist used to evaluate the body of a lambda by a new pair that substitutes a unique identifier for the binding variable. Figure 6-14 diagrams such an approach, again for a case where the renaming is always done.

The problem with this approach is that we have essentially done much of the work we were trying to avoid by creating an association list; we end up going through an expression and doing brute-force substitution. A later section will introduce a solution to both this problem and several others. The reader should, however, remember the source of the problem, because it will show up in somewhat different circumstances later as the ***funarg problem.***

6.4.2 Multiple Arguments

Multiple arguments are a feature that we definitely want to handle in real systems. A solution mentioned above involves making an association list into a list of lists, where each list element is actually the substitution list for each function application. Thus it is the list of binding variables and their matching actual arguments. The changes to make this happen in the above interpreter are small. First, the last line of **eval** becomes:

else **apply(car(*e*), map((λ*x*|eval(*x*,*alist*)),cdr(*e*))**

This applies the function **eval** to all the argument expressions in the original s-expression other than the first, with the same context represented by *alist,* and gathered into a new list. This list is then passed on to **apply** as the arguments to the function.

The second change is to the second-to-last line of **apply:**

then **eval(caddr(*f*), cons(map2(cons,cadr(*f*),*a*),*alist*))**

Here the list of formal argument names (**cadr(*f*)**) is paired, via a **map2** function, with the actual argument values and appended to the front of the *alist*. The result is then fed back to **eval** as the new *alist*.

```
eval(e,alist) =
  if atom(e)
  then let z = assoc(e,alist) in
        if null(z) then e else eval(cdr(z), alist)
  elseif car(e) = lambda
  then let z = newid( ) in
        list(lambda,list(z),
              eval(caddr(e),((caadr(e).z) .alist))
  else apply(car(e),eval(cadr(e),alist),alist)
```

FIGURE 6-14
A revised **eval.**

6.4.3 Recursion

Now consider what happens when this interpreter is used on an application with recursive properties. If the function is some sort of "built-in" function where the implementation handles the recursion directly, then the process described above works well and in fact mirrors the frame-building process used in many conventional languages for recursive procedures. The problem, however, comes up when the recursion is explicit, as in **YR,** where **Y** is the ***fixed-point combinator*** discussed earlier and **R** is some recursive function. The applicative-order evaluation shown here blows up, and wanders off to infinity computing **R(YR)** ⇒ **R(R...(YR))** ⇒ **....**

Matters get even worse when a set of mutually recursive functions are desired, as in

letrec *f*=**F** and *g*=**G** and *h*=**H** in **E**

Here the context (association list) for **E** is of the form

(((*f*.**F1**) (*g*.**G1**) (*h*.**H1**))...)

However, **F1** (the value for *f*) (and likewise for **G1** and **H1**) requires knowing the values for *f, g,* and *h* before they are computed. But this is the context for **E.** We need to know the association list for **E** before we can compute the association list for **E.**

The solution for this problem involves new mechanisms to be discussed in the next section.

6.5 CLOSURES

Most of the problems of the last section come from trying to improve the "efficiency" of a basic interpreter through a combination of applicative-order reduction and the use of simple association lists in place of immediate substitutions. These proved not to provide the full efficiency gain hoped for (examples where name clashes might occur), and were relatively inadequate for general recursion.

A better solution to both problems involves "packaging" an expression with its environment into a single unit which can be passed around at will, but still be unpackaged and evaluated when needed. Such a package is called a ***closure*** and is something that not only makes it possible to consider using alists in interpreter descriptions, but also introduces some very novel ideas that permit opportunities for parallelism and for the easy expression of essentially infinite objects. Later sections address the latter opportunities; the rest of this section and the next address the former.

Consider an application $(\lambda x|\mathbf{E})\mathbf{A}$ (or the equivalent, let x=**A** in **E**). Its evaluation involves the substitution $[\mathbf{A}/x]\mathbf{E}$. Assuming for the time being that **A** has no free identifiers, let us suspend this evaluation just before the substitution takes place. What we have is the expression **E** and the ***environment*** x=**A** [or association list ((x.**A**))]. This combination is called a ***closure*** and for this text will be written as

⟨closure⟩ := [⟨expression⟩,⟨environment⟩]

where the ⟨environment⟩ is expressed as an s-expression association list, and the expression is in whatever format seems convenient at the time. The term ***context*** is often used as an alternative for environment.

For example, expressing "let x=3 and y=5 in $2\times x+y$" as a closure results in [($2\times x+y$), ((x.3)(y.5))].

Evaluating a closure consists of restarting the substitution, that is, using the environment part as a source for values for the free variables in the expression part.

Nesting of closures is not only possible, but necessary. Consider, for example, the expression

let x=**A** in let x=x+1 in **E**

This is equivalent to the nested closure [[**E**,((x.x+1))], ((x.**A**))]. In such cases the evaluation of a closure must itself be recursive. To evaluate the outer closure we must first evaluate the inner closure. This evaluation involves replacing all the free occurrences of x in **E** by x+1, where the new x is the one in the outer closure's environment. In general, if an expression inside a closure is itself a closure, the outer one must be suspended while the inner one is worked.

The modifications to **eval** to handle closures are direct. **Eval** has two arguments as before, the expression to be evaluated and the alist. If the expression is an identifier, we look it up in the alist. If it is a closure, we may call **eval** recursively, as in **eval(eval(get-expression(*e*), get-environment(*e*)),*alist*)**. If it is an application, several alternatives are possible depending on the type of reduction sequence desired. In many cases (to be discussed later) we may simply want to form and return a new closure. In other cases we may evaluate it.

Figure 6-15 diagrams a simple case where we will always build a closure upon finding an application, and expand the closure only when its value is needed. In this case a single evaluation of an application returns a closure, and a second evaluation is needed to unpack the closure. To make this fully evaluate an expression we need an outer function which will apply **eval** repeatedly until an expression has no closures in it of value to a user.

```
eval(e,alist) =
   if is-a-identifier(e)
   then let z=assoc(e,alist) in
         if null(z) then e else cdr(z)
   elseif is-a-closure(e) "evaluate closure here ****"
   then let e1=get-expression(e)
         and alist1 = get-environment(e) in
         eval(eval(e1,alist1),alist) {nest evaluations}
   else ... as before...

apply(f,a) = ... as before...
   elseif is-a-function(f)
   then create-closure(get-body(f),
                        create-alist(get-identifier(f),a))
   else ... as before..

Example: let x=3 and y=2 in let x=x+y in x×y
≡ eval([[x×y,((x.x+y))],((x.3)(y.2))], nil)
→ eval(eval([x×y,((x.x+y))],((x.3)(y.2))),nil)
→ eval(eval(eval(x×y,((x. x+y))), ((x.3)(y.2))),nil)
→ eval(eval((x+y)×y, ((x.3)(y.2))),nil)
→ eval((3+2)×2, nil) = 10
```

FIGURE 6-15
Basic closure evaluation.

The key point about this process is that, if formed correctly, a closure is entirely equivalent to the original expression, and it can be evaluated (unsuspended) at any time and still get the equivalent normal-form answer. In a sense it is a closed universe, complete in itself, whose ultimate value never changes. In prior terminology, it is ***referentially transparent,*** since its value is the same whenever it is evaluated. This will permit systems described in later chapters to do the minimal processing necessary to return something to a user but still be capable of recreating the full answer at any time. This will be particularly useful when dealing with very large, even infinite, lists, where the embedded expression describes how to compute the next element.

6.6 RECURSIVE CLOSURES

The above implementation of closures handles nested applications, free variables, and normal- and applicative-order reductions relatively easily. It does not, however, handle recursion. Consider, for example, the differences between "let f=**A** in **E**" and "letrec f=**A** in **E**." In the former, any free instances of f in **A** refer to f's bound in expressions surrounding this one. In the latter, any free instances of f in **A** should refer to the whole value of **A** as is. Even worse, those references to f in the **A** being substituted for *f* in **A** must also be replaced, as must the references to *f* in that replacement. There is nothing to stop this substitution from going on forever.

In terms of the substitution notation we have used to date, what we want is something like:

$$[\mathbf{A}/f]\mathbf{E}=[([\mathbf{A}/f]\mathbf{A})/f]\mathbf{E}=[([([\ldots)/f]\mathbf{A})/f]\mathbf{A})/f]\mathbf{E}$$

The beauty of using a closure to handle this is that it can delay any required substitution, particularly these recursive ones, until they are absolutely needed, and then perform only a minimal amount of substitution necessary to satisfy the immediate evaluation. Keeping all or part of the closure around will permit later recursions as necessary.

The problem with expressing this as a closure is getting an alist entry which has some sort of internal references to itself. As a first cut, what we want is a closure of the form [**E**,((*f*."value for *f*"))], where the "value for *f*" is itself a closure of the form [**A**,((*f*."value for *f*"))]. This "value for *f*" is a nontrivial object since it requires some sort of reference to itself in itself. That is, if we let α stand for the "value for *f*," then

$$\alpha = [\mathbf{A},((f.\text{“value for } f\text{”}))]=[\mathbf{A},((f.\alpha))]$$

The context for the closure α includes the complete closure α itself! Figure 6-16 diagrams this self-reference.

The expansion to and forms of letrec is also worth discussing. Consider the expression:

letrec f_1=$\mathbf{A}_1$ and...and f_n=$\mathbf{A}_n$ in **E**

When converted to pure lambda expressions, the equivalent of this statement gets quite complex. However, if expressed in terms of closures and a self-interpreter, the process is much more understandable. The closure for the whole expression is of the form:

$$[\mathbf{E},\ ((f_1.\text{“closure for } f_1\text{”})\ldots(f_n.\ \text{“closure for } f_n\text{”}))]$$

where the "closure for f_j" is a closure,

$$[\mathbf{A}_j,\ ((f_1.\text{“closure for } f_1\text{”})\ldots(f_n.\ \text{“closure for } f_n\text{”}))]$$

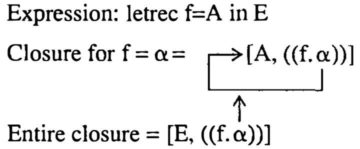

Expression: letrec f=A in E
Closure for f = α = [A, ((f.α))]
Entire closure = [E, ((f.α))]

FIGURE 6-16
Value in a recursive closure.

The contexts for all these internal closures is the same. If we denote this context by β, we get that the closure for the whole expression is

$$[\mathbf{E},\beta] \text{ where } \beta=(\ldots(f_j.\text{“context for } f_j\text{”})\ldots)=(\ldots(f_j.\beta)\ldots)$$

$=((f_1.[A_1,\beta],\quad (f_j.[A_j,\beta], \cdot\ )$

Figure 6-17 diagrams this arrangement.

If we implement such data structures as s-expressions, we find that the **cdr**s of cells containing such contexts point back to the **car**s of that cell or some earlier cell linked to it. This will be discussed in more detail for the SECD Machine.

6.7 CLOSING THE LOOP—READ-EVAL-PRINT

If we were being totally rigorous in our description of these interpreters, they would take the form:

letrec **eval**=...and **apply**=...and...in **eval(E)**

where **E** is some expression that we want interpreted. While fine for one-of-a-kind uses, this is hardly of general-purpose utility. We must rewrite at least part of this overall expression each time we want to interpret a new expression.

A more useful approach is to change the final "in **eval(E)**" into something which repeatedly:

1. Reads in an expression to be evaluated from some input device, such as a terminal
2. Evaluates the expression
3. Writes the expression out, for example, back to the terminal's screen
4. Repeats the process for a new input expression

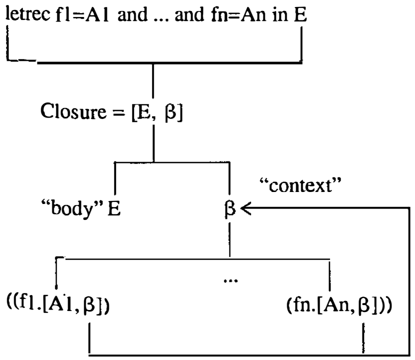


FIGURE 6-17
A fully recursive context.

Most real functional-language interpreters work this way. The code that controls this loop is often termed a ***read-eval-print loop,*** a ***listener,*** or a ***waiter,*** and looks something like:

in **listener**(*e*) whererec **listener**(*e*)=**listener(write(eval(read())))**

where **read** is a function (of no arguments) which returns an s-expression to be evaluated from the input device, and **write** prints out its argument (again an s-expression) on some output device. For convenience, we can assume that **write**(*e*) returns *e* as its functional output.

If our interpreter will not handle functions of no arguments, we can add a dummy argument to **read** as shown above and then simply ignore it.

Note also that **listener** is tail-recursive. This means that an efficient implementation of it could avoid stacking arguments or return information from any kind of stack. This is important if we try to implement it on a machine with finite memory.

Note that the above definition of **listener** never completes—as soon as **write(eval(read()))** completes, **listener** starts up another go-around. The argument for **listener** is simply there to provide a place to put **write(eval(read()))**; the value it retains is never used.

6.8 PROBLEMS

1. Trace out the evaluation of the following expressions using the interpreter of Figure 6-3. Use both normal- and applicative-order **apply.** Show the arguments to all calls (except that you may skip internally recursive calls to **subs** or calls to functions whose values are obvious, like **get-identifier**(λ*x*|...)).
   a. (((λ*x*|(λ*y*|(*yx*)))*a*)*b*)
   b. (λ*x*|(λ*y*|(λ*z*|(*y*((*xy*)*z*)))))(λ*nz*|*z*)
   c. ((λ*x*|(λ*y*|*y*))) ((λ*x*|(*xx*))(λ*x*|(*xx*)))) *w*
2. Convert each of the above to an s-expression form.
3. Repair Figure 6-4 so that it does not get caught in an infinite recursion when called from something like **eval**((*xy*)).
4. Modify **subs** in Figure 6-4 to rename lambda variables only when necessary.
5. Draw out the cell representation for the s-expression (lambda (*x y*) (*y* (*x x*))) *w z*) from Section 6.2.2.
6. Convert the pure lambda expressions for addition and multiplication to the s-expression form of Figure 6-5.
7. Convert the following version of **member** to the s-expression form of Figure 6-5. Assume built-in functions **null, eq, car, cdr.**

   member(*x*,*s*)=if **null**(*s*) then F else if **eq**(*x*,**car**(*s*)) then T else **member**(*x*,**cdr**(*s*))
8. Assume the following syntax for a certain class of s-expressions:

```
⟨logic-expr⟩ := ⟨id⟩ | T | F
              | (AND ⟨logic-expr⟩ ⟨logic-expr⟩)
              | (OR ⟨logic-expr⟩ ⟨logic-expr⟩)
              | (NOT ⟨logic-expr⟩)
              | (LET ⟨id⟩ ⟨logic-expr⟩ ⟨logic-expr⟩)
```

   Define in abstract syntax a function that will evaluate such expressions, assuming that it is given as input an association list of all identifiers and their current values. Show that it works for the expression:

   (LET *x* F (LET *y* T (AND (NOT *x*) (OR *x y*))))

9. Write an abstract program that represents the curried form of each of the following functions. Then show what is returned when this form is applied to the specified argument. Remember that the curried form of a function **f**(*x, y*) is a function **f**′(*z*) such that for any **A** and **B**, (**f**′**(A))(B)=f(A, B)**. (*Hint:* Look at what Figure 6-9 would do.)
   a. Ackerman's function, argument=1.
   b. **Append,** argument=(1).
10. Modify Figure 6-15 to handle an extension to cond such that if the last element of cond's list has only one expression in it, and no prior test passes, the value of this expression is returned as the value of the cond. For example, (cond ((= 1 2) F) ((⟩ 4 5) F) ((+ 1 2))) would return 3.
11. Rewrite Figure 6-9 to employ a special form for the **Y** combinator to handle recursion. This special form would have syntax (Y ⟨lambda-function⟩).
12. Write a function **evaluate** that calls **eval** of Figure 6-15 as often as required to fully reduce an expression so that there are no embedded closures. Show that this interpreter works on (**member** 1 (quote (2 1))).
13. Write s-expression forms of functions **read**() and **write**(*e*) that read and write arbitrary s-expressions. Assume that the only input/output (I/O) built-ins you have are **read-atom**(), which returns the next atom from the input, and **write-atom**(*e*), which writes one to the output. Thus a loop on **read-atom** where the input device has "(cons 1 2)" would return successively "(," cons, "1," "2," and ")." Sending these back to **write-atom** would print the same expression back.
14. Rewrite the applicative order form of Figure 6-3 as a single s-expression in the format of Figure 6-5, including a read-eval-print loop as the expression to be evaluated. Assume that you are provided with functions **read** and **write** to handle complete s-expressions I/O.
15. Write the expanded interpreter of Figure 6-9 and Figure 6-15 in the syntax of Figure 6-5, again with a read-eval-print loop. Is this interpreter capable of interpreting itself? If not, what is missing?

CHAPTER 7

THE SECD ABSTRACT MACHINE

So far we have described two simple functional languages based on lambda calculus: abstract programming and a prefix s-expression equivalent. The operation of these languages has been described in terms of interpreters for such languages (written in themselves). This corresponds to the general concept of ***denotational semantics.***

An alternative approach to describing the semantics of such languages is through ***interpretative semantics,*** where we define a simple ***abstract machine*** and combine this with a description of how a compiler would take programs written in the language of interest and generate matching code for the abstract machine.

This chapter takes such an approach. We will define the ***SECD Machine*** as an abstract machine with properties that are well suited to functional languages, and will give a simple compiler for the prefix s-expression form discussed in the last chapter. The purpose for doing this is twofold. First, it should reinforce the reader's understanding of how functional languages work. Second, it serves as a very clean departure point for discussing those hardware architectural concepts that show up in real machines for real functional languages.

In terms of organization, the first section discusses briefly a very simple form of the memory model used by the SECD Machine, namely, a list-oriented one. The next chapter will expand on it in great detail, but much of that is not needed to understand the rest of the SECD architecture.

This is followed by descriptions of how the SECD Machine uses such memory to implement several important data structures, namely, stacks, association lists, and closures.

Next is a description of the important registers and basic instructions needed by the SECD Machine. The descriptions of individual instructions are via essentially ***axiomatic semantics,*** namely, how the machine state is changed by any of these instructions being executed. Extensions to this ***instruction set architecture (ISA)*** to cover special features are discussed in later chapters.

Following this is development of an abstract program that generates SECD code for simple s-expression programs. Given the equivalence of this s-expression format to abstract programs, the existence of such a compiler means that we could compile into SECD code any of the interpreters discussed earlier, or even the compiler itself.

Finally, the chapter discusses one of the thorniest problems to handle with functional languages, namely, how to handle arguments to functions which are themselves functions, particularly ones that have been only partially evaluated or curried. The is the famous ***funarg problem,*** and our discussion will address not only what it is but how it affects the design of real machines and compilers, and why our simple SECD model avoided it.

The SECD Machine itself was invented by Peter Landin (1963) as a sample target for the first function-based language, LISP. As pictured in Figure 7-1, it has been used since as a basis for semantic descriptions, as an intermediate language for compilers and interpreters for LISP and other functional languages, and as the starting point for many of the current generation of LISP and Artificial Intelligence workstations. The ver-

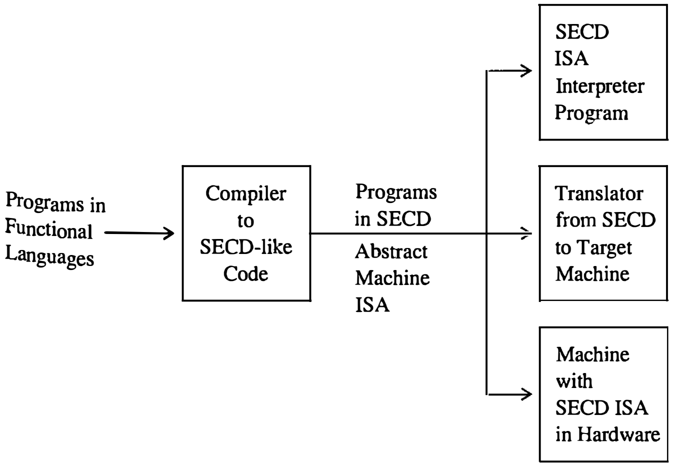


FIGURE 7-1
Uses of the SECD instruction set architecture.

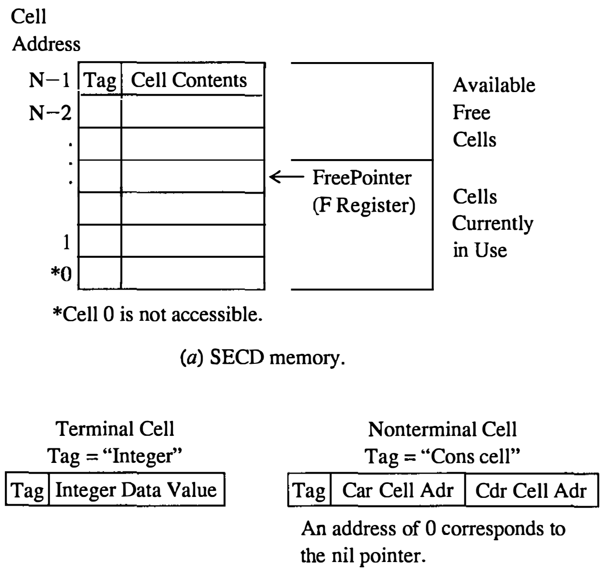
sion of the SECD Machine described here is most closely related to that of Henderson (1981), but it has been modified in several aspects to improve its educational value. Actual Pascal code that describes an interpreter for an SECD Machine can be found in both Henderson (1981) and Henderson et al. (1983). Burge (1975) contains another good description.

7.1 LIST MEMORY

As described earlier, functional languages and s-expressions seem to go well together. Further, the implementation of s-expressions from conventional random-access memory seems to be fairly efficient. Consequently, it should come as no surprise if we assume that our abstract SECD Machine incorporates a memory model that supports s-expressions directly. This section gives an overview of a very simple form of such a model; Chapter 8 is devoted to a detailed discussion of more efficient implementation on real memory structures.

The basic SECD memory model assumes a large collection of identically formatted *cells* in a single memory pool (Figure 7-2). Each cell consists of some fixed number of memory bits and has a unique address through which the contents of the cell may be read or modified. Further, the contents of a cell may have several formats, each of which divides the cell's bits into several fields.

A common *tag field* in each cell describes how the rest of the cell should be interpreted. For this chapter we assume only two basic types: ***integers*** and ***cons cells.*** Later chapters will expand the set of interesting possibilities. The former correspond to ***terminal cells***; the latter to ***nonterminal cells.***



(*a*) SECD memory.

(*b*) Cell formats.

FIGURE 7-2
The SECD memory model.

In integer-type cells the rest of the cell other than the tag field indicates the binary representation of some integer. For cons cells, the rest of the cell is divided into two halves: the ***car field*** and the ***cdr field.*** Both of these fields contain pointers (addresses) to other cells in the memory.

Arbitrary s-expressions are constructed as described in Chapter 3. When some kind of list or dotted pair is needed, it is built out of one or more cons cells interconnected by encoding the addresses of other cells in the **car** or **cdr** fields.

A pointer value of "0" indicates a ***nil pointer.*** Thus cell 0 is not usable by the system.

For now, when an SECD instruction wishes to build a new s-expression, it allocates new cells from memory, in a bottom-up fashion. A register in the machine, the ***freelist pointer*** (or ***F register***), points to the current boundary. All cells at it or below it in memory are in use by the program; all cells above it are free to be allocated as needed. Allocating a new cell causes the F register to be incremented. The cell addressed by this incremented F register is then available for use as the new cell. The appropriate **car** and **cdr** values can then be written into it.

At program start, F is initialized to 0.

At this point we will ignore what happens when F runs over the top of available memory. This, and related questions on how to "reclaim" cells below F that are no longer in use, is a subject of the next chapter.

Finally, to simplify drawings, we will not show tag fields of individual cells unless necessary. If a cell is depicted as a single rectangle with a number in it, it has an integer tag. If it is divided into two subrectangles, it is a cons cell. A nil in either box represents a pointer to cell 0, the nil pointer. Also, as described in Chapter 3, we will very often record the value of a terminal cell in the field of the nonterminal that points to it. The meaning of this should be that the actual value is in fact in a separate cell with a pointer to that cell in the nonterminal.

7.2 BASIC DATA STRUCTURES

There are five key kinds of data structures that the SECD Machine will build, manipulate, and keep in memory:

- Arbitrary s-expressions for computed data
- Lists representing programs to be executed
- Stacks used by the program's instructions
- ***Value lists*** containing the arguments for uncompleted function applications
- Closures to represent unprocessed function applications

Several variations of these data structures will be described in later chapters, particularly when we discuss lazy evaluation.

It is clear how the first of the above five categories, namely, s-expressions, can be built in the memory from the last section. Given that the SECD Machine contains instructions that perform the equivalent of **cons,** building a new s-expression involves allocating a new cell from memory (bumping the F register by 1), setting its tag to **cons,** and storing the appropriate pointers into its **car** and **cdr** fields.

The other data structures are addressed individually in the following sections.

7.2.1 Programs as Lists

Programs for the SECD Machine look like garden-variety lists. Each element of the list corresponds to an individual instruction, and execution proceeds (for the most part) one element at a time from the front of the list to the rear. A call to a function involves saving where one is in the current list and starting execution at the beginning of the list associated with the called function.

While it may seem wasteful to link together strings of often sequential instructions as a list (rather than as sequential words in an "array"), there are several significant advantages. First, we do not need to invent a new data structure just for program storage. Second, and most important, programs now look just like data, making it extraordinarily easy to write program that read, process, or even generate other programs. This makes writing compilers and interpreters for the SECD Machine in SECD code a relative snap.

The individual elements of a program list come in two types: simple integers or arbitrary lists. The former, simple integers, are equivalent to an ***opcode*** specifying some basic instruction in the SECD Machine's architecture. The latter, embedded sublists, usually represent alternative branches of program flow that will be decided dynamically when the program is run. Instructions which choose between these alternative flows of control correspond somewhat to the branch and call instructions found in conventional architectures. For example, to do an if-then-else, one element of a program's list will be a basic instruction (called out by an integer), which when executed will decide which of the two following elements of the program list should be executed. These following elements will be lists themselves. Each represents a then or else snippet of code. Instructions at the ends of the snippet returns control to the main thread.

Figure 7-3 gives a simple example.

7.2.2 Stacks as Lists

The SECD Machine uses several stacks during execution. One serves as an evaluation stack in a way reminiscent of ***reverse Polish*** execution. Other stacks contain points in the program to pick up from when the current functions are completed.

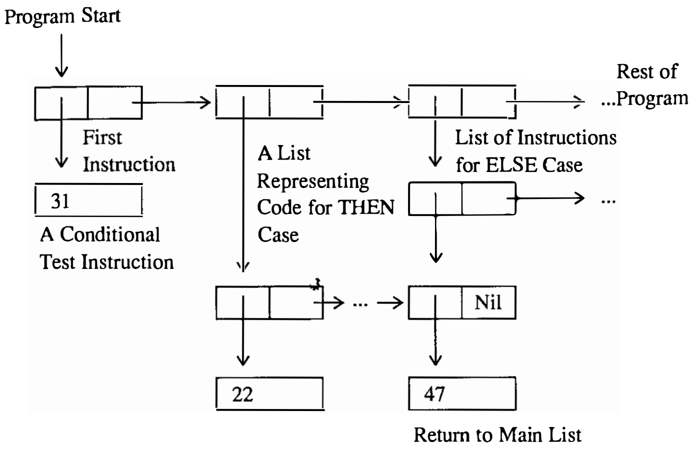

Program Start
First Instruction
31
A Conditional Test Instruction
A List Representing Code for THEN Case
List of Instructions for ELSE Case
Rest of ...Program
...
Nil
22
47
Return to Main List

FIGURE 7-3
Programs as lists.

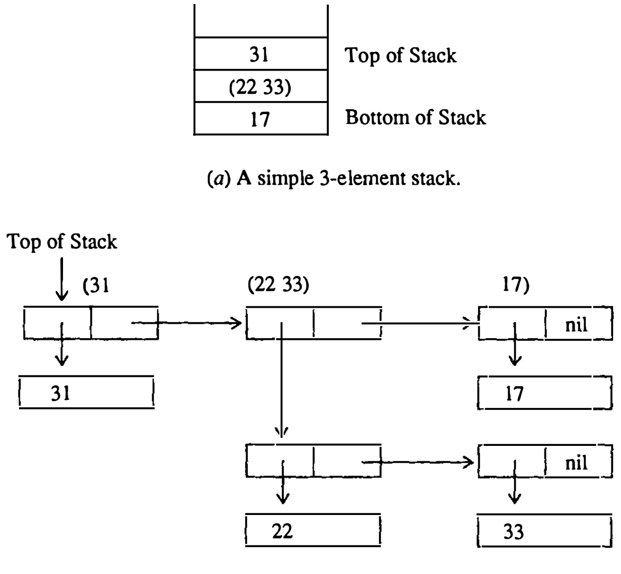
In all cases the stack operates just as a conventional stack would. We can ***push*** values onto the stack and ***pop*** them off again, all in a last-in, first-out manner.

The major difference from a conventional stack implementation is that, like programs, stacks in the SECD Machine are made from lists of memory cells. Again, this may seem wasteful of storage bits, but it does have the advantage of using the machine's natural data structure. Also, it permits us to grow different stacks to arbitrary sizes in arbitrary orders, until memory is exhausted. This is unlike conventional machines, which grow stacks in consecutive locations of memory until some preallocated boundary is reached. In such cases the maximum size of the stack is limited to the amount of storage allocated to it by the system. Overgrowing one stack's area is not permitted, even if storage exists in other stack or data areas.

By analogy, the top of the stack is equivalent to the leftmost element of its list equivalent. Thus, pushing an object to a stack is implemented by **cons**ing it onto the matching list. Popping an element requires a **caar** to get the element's value, and a **cdr** to return a pointer to the rest of the list. An empty stack is the nil list.

Figure 7-4 diagrams a simple example.

A subtle but important difference between these stacks and conventional stacks built out of sequential memory is that with lists a "pop" followed by a "push" does not overwrite the storage allocated to the element "popped" off. The new element is created in a separate memory cell, with the cell's cdr pointing to the cell holding the next element, wherever it is. The cell holding the original value "popped" off still exists, with its cdr pointing to the same next cell.



(*a*) A simple 3-element stack.

(*b*) The list equivalent.

FIGURE 7-4
Stacks as lists.

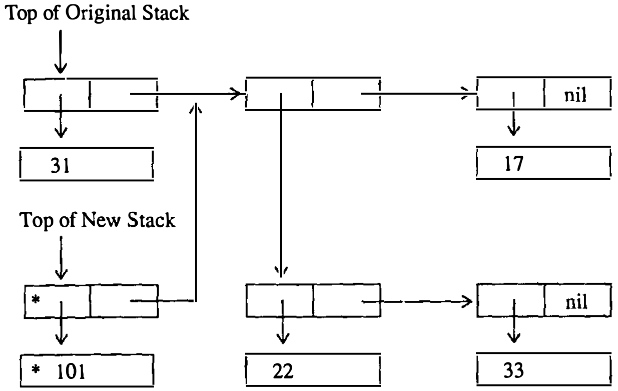
For example, if Figure 7-4(*a*) is implemented conventionally in sequential memory locations, popping 31 off and then pushing, say, 101, back on will cause the memory location holding the 31 to now contain 101. The 31 is lost.

In comparison, doing the same thing to Figure 7-4(*b*) erases neither the cell containing the 31 nor the cell containing the pointer to it. The cdr of that cell still points to the second list cell, as does the cdr of the new list cell whose car points to 101. See Figure 7-5.

There are cases where keeping this old stack available without modification is a valuable feature. There are many other cases, however, where this represents the generation of ***garbage***, that is, memory cells that are no longer used but are not available for reuse. Recovering these old cells if in fact no one else needs them is a function of a ***garbage collection*** system, which the SECD Machine defined to this point does not have, but which most real machines with such memories do have. Again, the next chapter will address such systems.

7.2.3 Value Lists

The previous chapter demonstrated the utility of association lists in implementing lambda calculus-based languages. They permit simple combinations of normal- and associative-order evaluations. Equally important for future topics, they provide, through closures, a natural mechanism for deferring an application.



Operation performed: Push 101 onto stack resulting from popping top off of stack (31 (22 33) 17).

* = new cells allocated during operation.

FIGURE 7-5
Popping and pushing a list-managed stack.

Given that the SECD Machine supports arbitrary s-expressions, it naturally supports association lists. In particular, the instruction set of the next section supports directly a form of association list that includes only the value half of each pair. The SECD form is then a list of sublists, where each sublist contains the argument values (and not the names) for a particular function call that has been made but as yet is not complete. Figures 7-6 and 7-7 diagram some examples of this.

A simple compiler can eliminate the need for the identifier name part by building an analog at compile time, measuring exactly where in

let x=1 and y=2 and z=3 in (x+y)+z
Equivalent to: (λxyz | (x+y)+z) 1 2 3

Association list for code = (((x.1) (y.2) (z.3)))
Name list = ((x y z))
Value list = ((1 2 3))

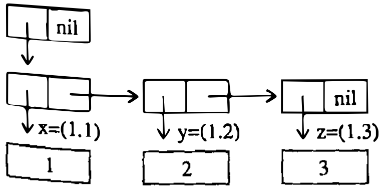


FIGURE 7-6
A simple value list.

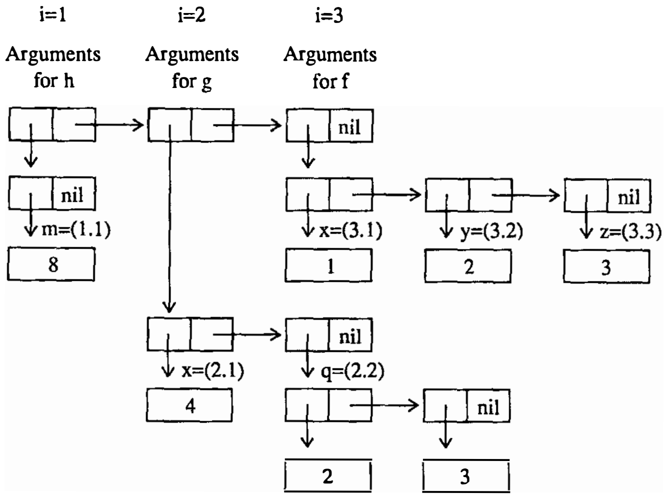
```
let h(m) = (3+m) ×9 in
    let g(x,q) = h(car(q) ×x) in
        let f(x,y,z) = g(x+3, list(y,z)) in f(1,2,3)
```

Note: f(1,2,3) → g(4, (2 3)) → h(2 ×4) → (3+8) ×9 → 99

(*a*) A sample program.

When Executing:	Alist:	Value List:
code for f	((x.1) (y.2) (z.3)))	((1 2 3))
code for g	(((x.4) (q.(2 3))) ((x.1) (y.2) (z.3)))	((4 (2 3)) (1 2 3))
code for h	(((m.8) ((x.4) (q.(2 3))) ((x.1) (y.2) (z.3)))	((8) (4 (2 3)) (1 2 3))

(*b*) Matching association lists.



(*c*) The value list for h.

FIGURE 7-7
More complex value lists.

the association list the value will be at run time, and encoding that index into the appropriate SECD Machine instructions. This index will be of the form (*i.j*), where both *i* and *j* are integers. The *i* value determines which sublist of the alist is desired, and then *j* determines which element of that sublist is the actual argument. Thus *i* corresponds to how far back in the stack of pending function calls the identifier is found as an argument, and *j* determines which of that call's arguments is the desired one.

For reference, Figure 7-8 defines a function ***locate*** that, when given a paired index and a value list, returns the appropriate element. Note the interesting double use of the recursive subfunction **loc.**

```
locate(ij, vlist) = loc(cdr(x), loc(car(x), vlist))
  whererec loc(y, z) = if y=1 then car(z) else loc(y-1, cdr(z))

= j'th element of i'th sublist of vlist where ij=(i.j)
```

FIGURE 7-8
The **locate** function.

Closure for h just before application to argument list (8)

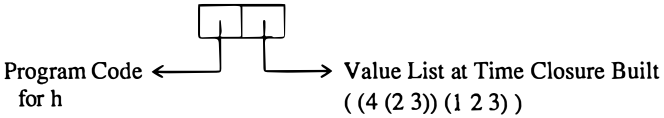


FIGURE 7-9
Closures as lists.

7.2.4 Closures

To the SECD Machine a ***closure*** is the combination of the code for some function and a value list. This combination is such that the actual function can be unpacked and executed at any time after the closure is built, and will return an answer that is absolutely identical to what would have been returned if the function had been executed at the the time the closure was built. As pictured in Figure 7-9, such a closure consists of **cons**ing two pointers into a single memory cell—a pointer to the function's code and a pointer to the appropriate value list.

7.2.5 Recursive Closures

The last chapter closed with a discussion of association lists that support recursively defined functions. The final solution was to package the expression being evaluated in a closure where the association list included as values separate closures for each of the recursive functions. The association lists for each of these embedded closures was simply a pointer back to the association list for the expression being evaluated. Thus when a function required a call to itself, its association list would lead back to the closure defining it.

Given the SECD memory model, this translates directly into a data structure. Figure 7-10 diagrams such a configuration just before the overall expression is evaluated. The value list for E's closure is a list of cells where the first argument is the list of values for **fl** through **fn.** Each of these values is a closure where the value list is a pointer back to the whole value list for **E.** The only difference between this and any of the

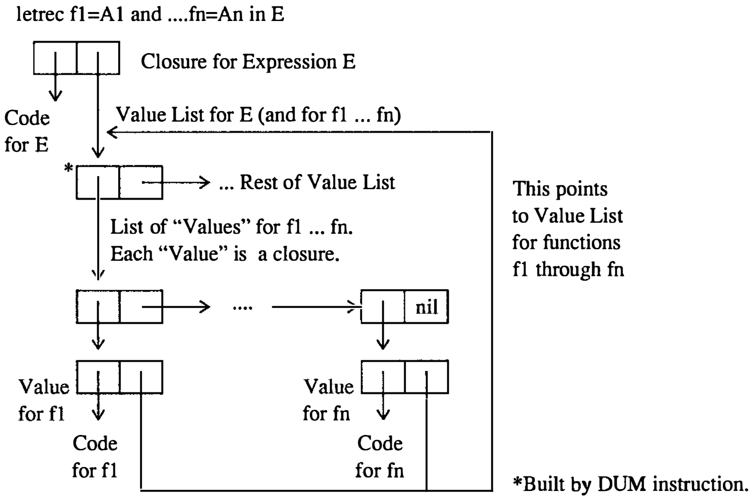

letrec f1=A1 and ....fn=An in E
Closure for Expression E
Code for E
Value List for E (and for f1 ... fn)
*
... Rest of Value List
List of "Values" for f1 ... fn.
Each "Value" is a closure.
This points to Value List for functions f1 through fn
....
nil
Value for f1
Code for f1
Value for fn
Code for fn
*Built by DUM instruction.

FIGURE 7-10
Value lists and closures for recursive expressions.

s-expressions we have encountered so far is that there is a circular loop in the pointer trail. Construction of such a loop cannot be done with the **cons** operator; a special SECD instruction to be described later is needed.

7.3 THE SECD MACHINE REGISTERS

The basic instruction set architecture of the SECD Machine consists of instructions that manipulate four main data structures found in memory. The structures consist of an evaluation stack (called simply the ***stack***) for basic computations, a value list or ***environment*** for storing argument values, a ***control list*** for the current program, and a ***dump*** where copies of the other three structures can be stored when one function application is suspended and another executed. Four machine registers, the S, E, C, and D registers, control each of these structures. As described earlier, a fifth register, the F register, indicates the next available memory cell for any of these structures.

The instruction set is divided into roughly three parts. One set of instructions manages the evaluation of "built-in" functions that the machine is capable of executing directly. Another set of instructions deals with ***special forms*** such as if-then-else. A final set deals with application of program-defined functions to program-specified expressions for arguments. There is considerable distinction made between nonrecursive and recursive functions.

For the most part, the stack, environment, and dump act like conventional stacks; items are "pushed" on the top and "popped" off in a last-in, first-out fashion. Further, except for the fact that everything is built out of linked cells rather than sequential memory locations, all these structures, registers, and associated instructions are close analogs to other abstract machine ISAs that support conventional programming languages [such as the p-code architecture for Pascal and the Forth threaded code architecture (Kogge, 1983).]

The following subsections describe these registers in more detail. Later sections describe the instructions and give example programs.

7.3.1 The S Register

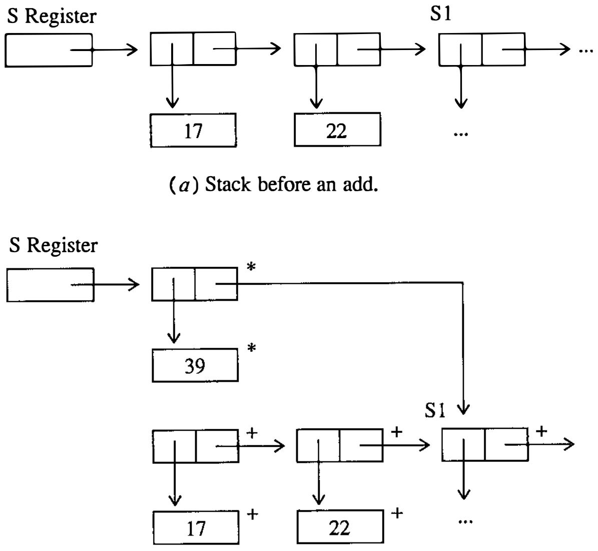
The ***S register*** points to a list in memory that is treated as a conventional evaluation stack for ***built-in*** functions, such as the standard arithmetic (+, −, ×, /) and list operations (**car, cdr, cons**). Objects to be processed by these functions are "pushed" on by doing a **cons** of a new cell onto the top of the current stack. The car of this cell points to a copy of the object's value. The S register after such a push points to the new cell. New cells are obtained by incrementing the F register and using the cell location specified by the result.

When an instruction specifying a built-in function application is executed, the appropriate objects for its arguments are obtained from the cars of the cells at the front of the list. The result will be placed in a new cell, and S set to point to yet another new cell whose car points to the new value and whose cdr points to the stack list remaining after the arguments. For example, an add (Figure 7-11) will take the values pointed to by the cars of the top two cells in the S list, add them, and set S to point to a cell whose car contains a pointer to the sum and whose cdr points to the stack after the original top of stack cells.

A key point is that, unlike a conventional stack built out of sequential memory locations, this new result does not overwrite the memory locations containing the original inputs. New cells are allocated both for the value and for the pointer cell in the list. The reason for this is that in various circumstances these original inputs (in fact, the entire stack before the function) are needed at other points in the overall computation. The drawback, of course, is that more storage is taken up, particularly if no one else needs the inputs. As defined so far, this storage for the original inputs is simply lost to future use. The next chapter will describe a common technique used by most real systems to identify when such "garbage" is generated, and to "collect" it for reuse.

7.3.2 The E Register

The ***environment register*** (or ***E register***) points to the current value list of function arguments (see Figures 7-6 and 7-7). This list is referenced by the machine when a value for an argument is needed, augmented (via a **cons**) when a new environment for a function application is constructed, and modified when a previously created closure is unpacked. The pointer from the closure's cdr replaces the contents of the E register.



(*a*) Stack before an add.

*New cells allocated by add instruction.
+Not changed but no longer referenced after add complete.

(*b*) Stack after an add.

FIGURE 7-11
The S register and the stack.

As with the stack above, the prior value list designated by the E register is not overwritten by any change to E, and is still intact in memory. Other mechanisms, including the dump, often retrieve the old list when the current function completes.

7.3.3 The C Register

The ***control pointer*** or ***C register*** functions just like the ***program counter*** or ***instruction counter*** in a conventional computer. It points to the memory cell that designates through its car the current instruction to execute. In the SECD Machine these instructions are simple integers which specify the desired operation. Unlike many conventional computers, there are no specialized subfields for registers, addressing designations, etc. When additional information is needed for an instruction, such as which argument to access from the E register's list, the information comes from the cells chained to the instruction cell's cdr.

Conventional computers normally increment their program counter after completing most instructions. The analog in the SECD Machine is the replacement of the C register by the contents of the cdr field of the last memory cell used by the current instruction (C←**cdr**(C)). Again as with conventional machines, there are exceptions to this, particularly for the equivalent of conditional branches, new function calls, and returns from completed application. Here the C register is replaced by a pointer provided by some other part of the machine.

7.3.4 The D Register

The ***dump register*** or ***D register*** is the last of the SECD Machine's registers. As with the S register, the D register points to a list in memory, this time called the ***dump***. The purpose of this data structure is to remember the state of a function application when a new application in that function's body is to be started. This is done by appending onto the dump three new cells which record in their cars the values of the S, E, and C registers before the application. When the application completes, popping the top of the dump restores those registers to their original values so that the original application can continue. This is very similar to the ***call-return*** sequence found in conventional machines for procedure or subprogram activation and return.

7.4 THE BASIC INSTRUCTION SET

Figure 7-12 diagrams the basic instruction set for the SECD Machine. For each instruction there is a brief description of how the machine's four registers change after its execution. The notation used consists of four s-expressions before and after a "→." The four s-expressions before the "→" represent the assumed lists in memory pointed to by the S, E, C, and D registers just as the machine starts to execute the instruction. The s-expressions after the "→" represent the same four registers after the instruction has been executed.

As described earlier, the notation "(**x y.z**)" stands for an s-expression whose first two elements are **x** and **y**, respectively, with the rest of the expression **z**.

With this convention, all the original s-expressions for the C register consist of a list with the first element designated by the name of the instruction being executed. In real life each such instruction would be a cell containing a specific integer. Thus any cell containing a "2" in a program list might represent an LDC, while a "15" might represent an ADD. For readability here we will use a mnemonic form.

These instructions break down into six separate groups, namely, those that:

1. Push object values onto the S stack
2. Perform built-in function applications on the S stack and return the results to that stack
3. Handle the if-then-else special form
4. Build, apply, and return from closures representing nonrecursive function applications

Instruction	Operation
	—Access Objects and Push Value to Stack—
NIL	s e (NIL.c) d → (nil.s) e c d
LDC	s e (LDC x.c) d → (x.s) e c d
LD	s e (LD (i.j).c) d → (locate((i.j),e).s) e c d
	—Support For Builtin Functions—
ATOM, CAR, CDR...	(a.s) e (OP.c) d → ((OP a).s) e c d
CONS, ADD, SUB,...	(a b.s) e (OP.c) d → ((a OP b).s) e c d
	—If-Then-Else Special Form—
SEL	(x.s) e (SEL ct cf.c) d → s e c? (c.d) where c? = ct if x≠0(T), and cf if x=0(F)
JOIN	s e (JOIN.c) (cr.d) → s e cr d
	—Nonrecursive Functions—
LDF	s e (LDF f.c) d → ((f.e).s) e c d
AP	((f.e′) v.s) e (AP.c) d → NIL (v.e′) f (s e c.d)
RTN	(x.z) e' (RTN.q) (s e c.d) → (x.s) e c d
	—Recursive Functions—
DUM	s e (DUM.c) d → s (nil.e) c d
RAP	((f.(nil.e)) v.s) (nil.e) (RAP.c) d → nil (rplaca((nil.e),v).e) f (s e c.d)
	—Auxiliary Instructions—
STOP	s e (STOP.c) d → s e (STOP.c) d -stop the machine
READC	s e (READC.c) d → (x.s) e c d where x is character read in from input device
WRITEC	(x.s) e (WRITEC.c) d → s e c d where x is printed on the output device

where rplaca(x,y) "replace car of x by y, and return pointer to x"
and locate(x,e) = loc(cdr(x), loc(car(x),e))
    whererec loc(k,e) = if k=1 then car(e) else loc(k−1, cdr(e))

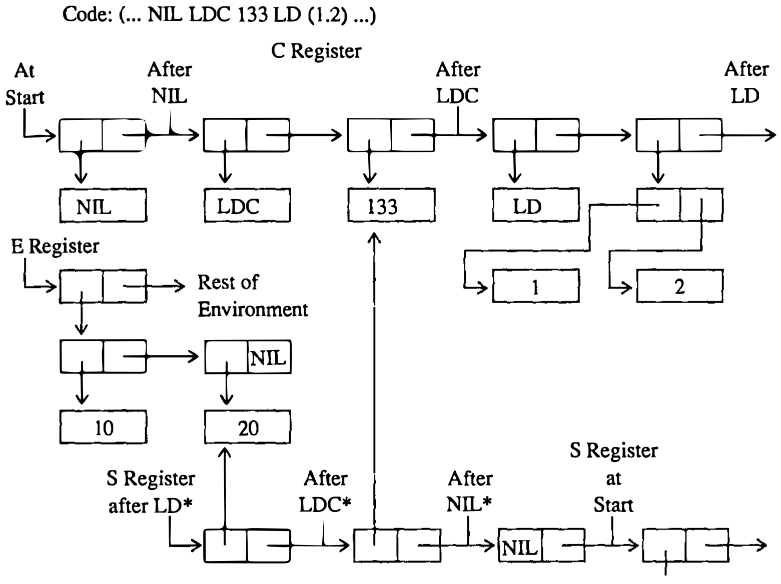
FIGURE 7-12
Basic instruction set for SECD Machine.

5. Extend the above to handle recursive functions
6. Handle input/output (I/O) and machine control

The following sections describe each of these groups.

7.4.1 Accessing Object Values

The first group pushes values of objects onto the S stack (see Figure 7-13 for an example). The first of these, ***NIL,*** pushes a nil pointer. Again this means that the S register after the instruction has been executed points to a newly allocated cell whose car contains a nil pointer (an address of location 0) and whose cdr points to the list designated by the S register before the instruction was executed.



*New cells allocated by instructions.

FIGURE 7-13
A sequence of data access instructions.

The second of these, ***LDC,*** loads a "constant" onto the stack. The constant value to be pushed is found as the next element on the C list after that for the instruction (the **cadr** of the C list at the time the instruction is started). Again a new cell is allocated, and chained onto the top of the S list. The car of this new cell is loaded with a pointer to this constant. Note that the value is not duplicated—only a pointer to it is. This means that this "constant" could in fact be an arbitrary s-expression, and the effect would be the same. The top of the stack would be that expression.

For this SECD Machine, the representation for the boolean constant F (false) is 0, and T (true) is taken as any integer other than 0, usually 1.

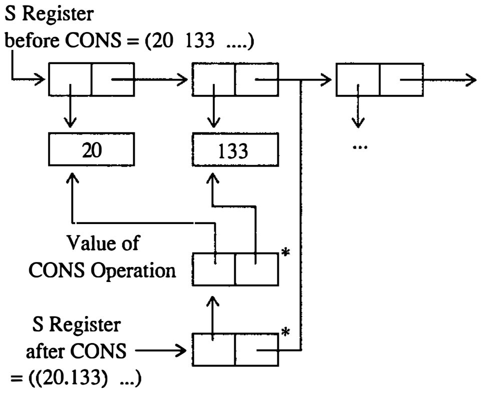
The third instruction of this group, ***LD,*** "loads" an element of the current environment. The **cadr** of the C list is a cell of the form (i.j), where the car i is the sublist element of the E list desired, and the cdr j is the element of that sublist. The function **locate** (see also Figure 7-8) describes how this access works. Again a pointer to that object is stored in the car of a new cell whose cdr points to the old S list.

7.4.2 Built-in Function Applications

The next group of instructions handle built-in functions such as **CAR, CDR, CONS, ATOM, ADD, SUBtract,**... Here the proper number of objects are accessed from the top of the S stack and the result placed in a new cell which is pointed to by the car of a second new cell. The cdr of this latter cell points to the rest of S's original list after the initial arguments. S is reset to point to this latter cell. Figure 7-11 diagrams a typical case for an **ADD.** A somewhat more complex example is the **CONS** instruction. This instruction allocates not one but two cons cells, one to hold the **cons** of the operands, and one to **cons** this result onto the S list. Figure 7-14 diagrams this, with Figure 7-15 diagramming the same situation in terms of possible memory location values.

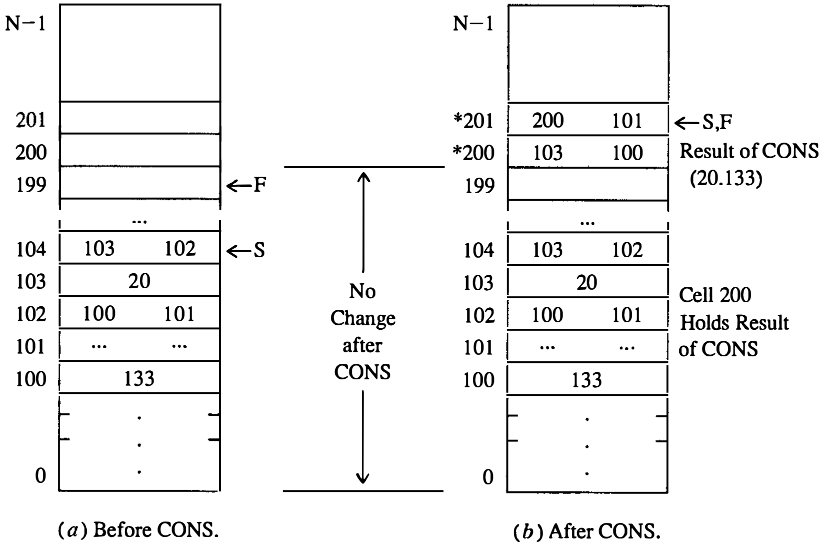
7.4.3 Instructions for if-then-else

The group of instructions used to implement if-then-else special forms are used in a specific order. The ***SEL*** instruction ("select") assumes that the top of the S list is an integer either zero or nonzero (encoded as described above to represent a boolean). Following the SEL in the control list are two elements (the **cadr** and **caddr** of the C list), both of which are themselves lists of instructions. The last instruction in each list is a ***JOIN.*** When executed, the SEL will push onto the dump a pointer to the C list just beyond the second sublist (i.e., a pointer to the **cdddr** of the original C list). The machine will then pop the top element off the S stack, test it, and replace C with its **cadr** if the value was nonzero, and with its **caddr** if the value was zero. Thus, these sublists correspond to the code for then and else expressions, respectively. The last instruction of each of these sublists, the JOIN, then resumes the original program by popping the top off the dump and resetting C to point to it. Figure 7-16 diagrams an example.



*New cells allocated during CONS execution.

FIGURE 7-14
Execution of the **cons** instruction.



(*a*) Before CONS.

(*b*) After CONS.

*New cells allocated by CONS.

FIGURE 7-15
Memory before and after a **cons** instruction.

Although the normal use of a SEL is after some test instruction which leaves a boolean on S, its definition also permits its use as a zero test after any arbitrary instruction sequence, such as a SUB. Thus we do not really need a ZERO, NULL, or even an EQUAL instruction in the SECD ISA.

7.4.4 Nonrecursive Program-Defined Functions

The nonrecursive function application instructions also work together in a very specific way. The ***LDF*** ("load function") instruction is followed in the C list by an element pointing to a sublist containing the code representing some program-defined function. The last instruction in this subprogram list is a ***RTN,*** which will function similarly to a JOIN.

When LDF is executed, it builds in a new cell a closure consisting of a pointer to the new function's code and a copy of the current E register. The latter represents the value list for all the identifiers (other than the function's immediate arguments) that have been bound to specific values by previous code. The closure is pushed onto the top of the S list.

Note that the function is not executed at the current time, merely packaged in a closure on the stack. At some arbitrarily later point in time, an AP ("apply") instruction will find on the top of the S stack a copy of this closure, and underneath it a list representing the argument

Code fragment: if null (x) then 10 else −10

Program list (... NULL SEL (LDC 10 JOIN) (LDC −10 JOIN) ...)

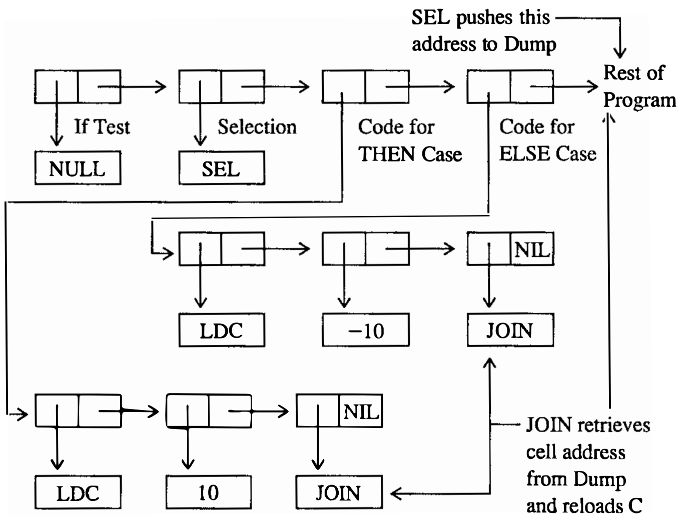


FIGURE 7-16
Structure of if-then-else code.

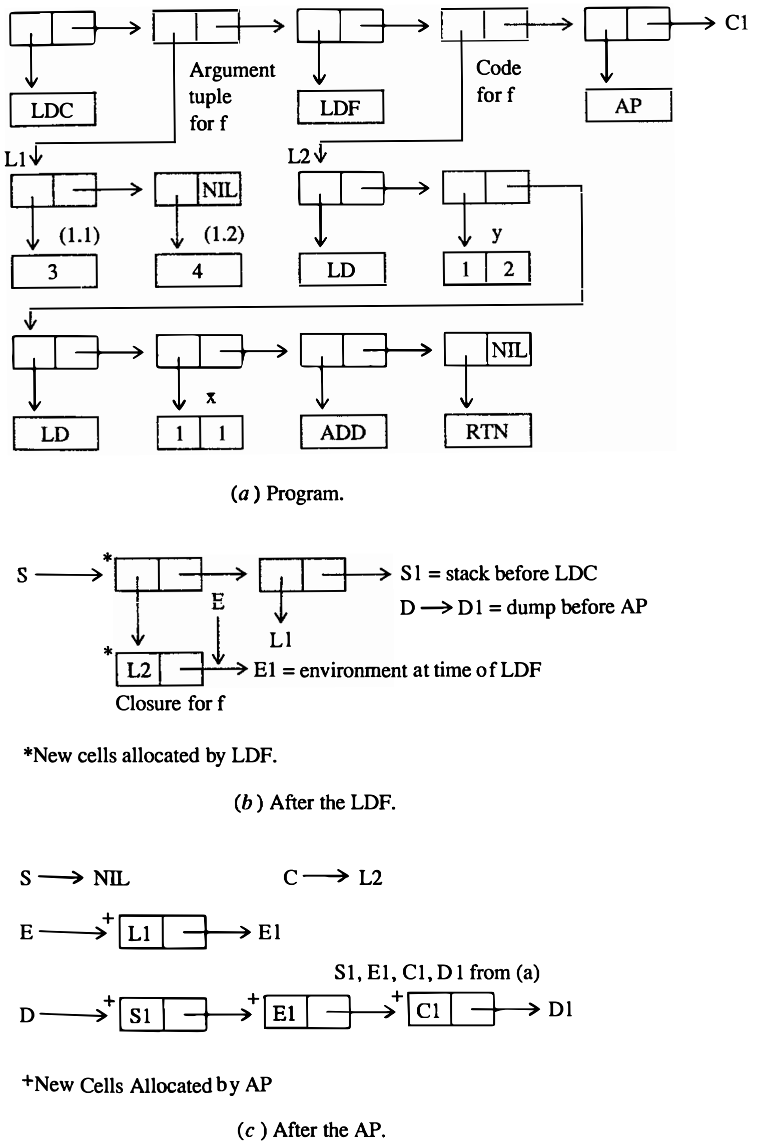
values to be applied to the function. Note that this argument list is a single element on the S list (namely, its **cadr**), and that the rest of the S list is arbitrary.

Executing the AP causes the **cddr** of S, the E, and the cdr of C to be pushed onto the dump. This represents the state the machine is to return to when the function application is complete. After this, the S register is reset to nil, making it an empty stack, the C register is set to the beginning of the code specified by the closure (the car of the closure cell), and the E register is set to the **cons** of the second element on the original S list and the cdr of the closure cell. This latter operation establishes a value list for the function application which consists of the function's arguments in the first sublist and the environment needed by the function's internal definition as the rest. Appropriate (i.j) indexing constants inside the function's code will give it access to these values.

The last instruction of a function's code list should be a RTN. This instruction takes the top element off the S stack as the value to return from the application. This value is **cons**ed to the old S value previously pushed on top of the dump, with S reset to point to this list. The E and C registers are restored directly from the dump, and the calling function is restarted. The only difference from the point at which it called the function is that the top of the S stack now contains the desired result.

Figure 7-17 diagrams a simple sequence of code that uses this set of instructions. Note that the actual program list shown was chosen for its

Assume: let f(x,y) = x+y in f(3,4)
Possible SECD code: (...LDC (3 4) LDF (LD (1.2) LD (1.1) ADD RTN) AP ...)



(*a*) Program.

*New cells allocated by LDF.

(*b*) After the LDF.

+New Cells Allocated by AP

(*c*) After the AP.

FIGURE 7-17
Basic nonrecursive application instructions.

simplicity in explaining LDF and AP and thus is somewhat different from that which would be produced by the basic compiler described later in this chapter.

In Figure 7-17 the argument list for the function application is a list of constant values. This is not always the case. Normally the arguments for the call must be computed by some set of function applications themselves. Given the way the AP instruction works, though this means that however the arguments are computed, they must be gathered into a list before we can invoke the AP. In the SECD architecture the easiest way to do this is to evaluate the arguments before invoking the function, but cons them together first. This involves using an initial NIL instruction before any argument is evaluated, and then evaluating the arguments one at a time, from right to left, with a CONS instruction after each evaluation sequence to put the argument value onto the growing argument list. The leftmost argument is thus evaluated last and placed on the front of the list. Figure 7-18 diagrams an example of this. The compiler discussed later implements the process of generating the proper SECD code.

7.4.5 Recursive Program-Defined Functions

The group of instructions handling recursive function calls are extensions of the previous set, and are perhaps the most difficult of the SECD instructions to understand. The reader should simply try to get the general idea here, and then observe later examples to see how everything fits together.

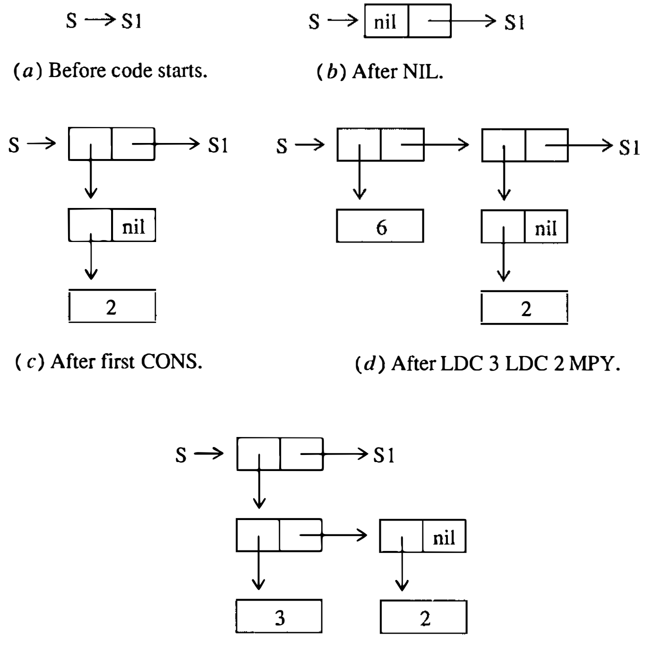
The ***DUM*** instruction **conses** onto the front of the environment list a new cell whose car is nil. This corresponds to a new argument list that is initially empty (a "dummy" list). This list will eventually be a pointer to the self-looping value list as shown in Figure 7-10.

A DUM instruction is used in a program just before the LDF(s) to build the closures for the recursively defined functions. This will make the environments stored in those closures point to a value list where the first element is this current "dummy" sublist.

The ***RAP*** instruction ("recursive apply") assumes that the top of the S stack looks like that for an AP, namely, a closure representing a function to be executed, and an argument list (i.e., the arguments for the expression at the end of the letrec ... and..in). In this case the function in the closure is the expression that calls the recursively defined functions (see Figure 7-19). The closure's environment is a pointer to the same list indicated by the current E register. This, in turn, should be a list where the first element was that built by the DUM. Furthermore, the list of argument values should be a list of closures, one per recursively defined function, where the environments of these closures are also pointers to the same dummy cell.

Execution of the RAP is identical to the AP except that the car of the cell pointed to by E (i.e., the dummy cell) is reset to point to the second argument of the S stack (the list of closures). Given that the closures in that argument also point to the dummy cell, the result is exactly the loop of pointers desired.

```
Program: let f(x,y)=x+y in f(2×3,6−4)
Code: (NIL LDC 6 LD 4 SUB CONS
        LDC 3 LDC 2 MPY CONS
        LDF (LD (1.2) LD (1.1) ADD RTN)
        AP ...)
```



(*a*) Before code starts.

(*b*) After NIL.

(*c*) After first CONS.

(*d*) After LDC 3 LDC 2 MPY.

(*e*) After final CONS.

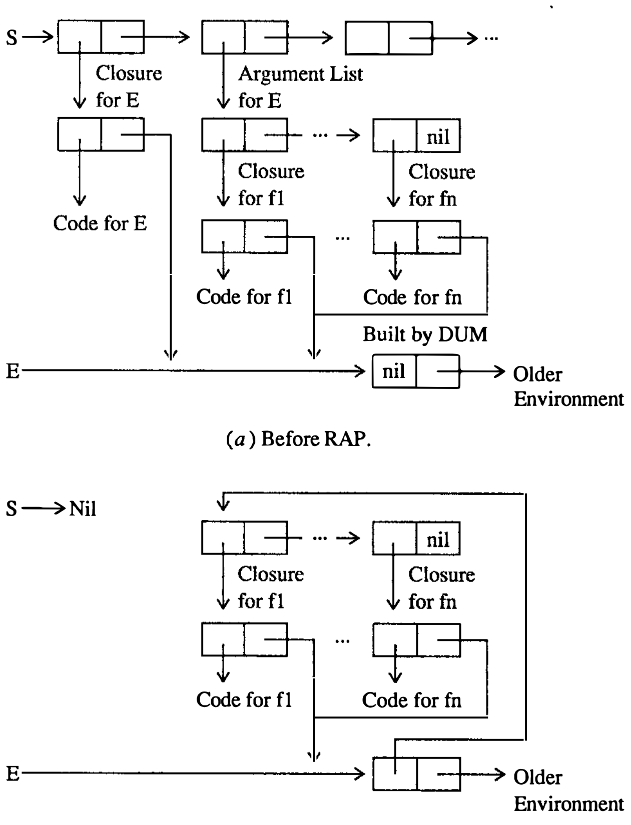
FIGURE 7-18
Construction of argument lists.

Within the code called by the RAP, any required calls to the i-th recursively defined function **fi** are initiated by a LD(1,i) followed by an AP. The LD fetches the closure for the function from the environment, and AP unpacks it as before. The environment established by the closure is the same environment it came from, with the exception that the AP adds a list to the front representing the arguments to the function. Thus, a LD(2.i) from inside the code for the function **fi** will retrieve an identical copy of its closure, and another AP will thus start a recursive call to **fi** properly.

Again, a RTN at the end of the expression unstacks the original S, E, and C values stacked by the RAP function(s).

```
Assume: letrec f1 =A1 and ... and fn = An in E
        = (λf1 ... fn|E) A1 ... An

Code = (DUM NIL LDF (..code for An... RTN) CONS
                LDF (..code for A1..RTN) CONS
                LDF (..code for E..RTN) RAP )
```



(*a*) Before RAP.

(*b*) After RAP.

FIGURE 7-19
Executing a RAP instruction.

7.4.6 Machine Control Instructions

The final set of SECD instructions control other aspects of a real machine's operation. The first of these, ***STOP,*** stops the machine in its tracks. No more instructions are executed, and the registers are left unchanged. This should be the last instruction in any program.

The ***READC*** and ***WRITEC*** instructions perform simple input/output operations. READC takes a character from some input device, constructs the integer equivalent in a new cell, and pushes a pointer to it onto the S stack.

WRITEC takes the integer on the top of the S stack and prints a character equivalent of it out to the standard output device.

These instructions are included here simply for minimal completeness. In real life much more complex I/O would be found, and would be handled much as in conventional machine architectures.

7.5 COMPILING SIMPLE S-EXPRESSIONS

Writing an elementary ***compiler*** to generate SECD code from an s-expression program is a relatively straightforward process. We recursively take the s-expression, look at its car element, and generate a standardized set of code based on its value and structure. Subexpressions embedded in the s-expression are converted into SECD code lists and then appended into the overall code. Figure 7-20 lists these SECD code sequences for each major special form.

For simplicity of notation here we assume that *⟨**expr**⟩ stands for the SECD code compiled from the s-expression ⟨expr⟩. Also, **AB** stands for the result of appending list **A** to list **B,** as in (1 2)(3 4)=(1 2 3 4).

Figure 7-21 outlines an abstract program which implements these transforms; Figure 7-22 lists some functions used in this program to handle special forms. The main function **compile** has three arguments, the expression *e* being compiled, a namelist *n,* and an accumulating parameter *c*. The namelist represents the variables that would be available in the environment when the s-expression *e* is executed. The accumulating parameter represents already-generated code to which the new SECD code for *e* should be appended on the front. Thus the initial call to compile an expression *e* would look like

compile(*e,* nil, (STOP))

With the exception of the if-then-else form, most of the s-expression forms are compiled into SECD code which emulates an applicative-order evaluation—the arguments to a function are evaluated first.

There is no error-checking built into this compiler, either syntatically or semantically.

7.5.1 Data Accessing Forms

The data accessing forms translate directly into sequences of SECD instructions that push the appropriate value onto the top of the S stack. In the case of an identifier, this involves identifying what entry in the value list will have the appropriate value at run time. The compiler of Figure 7-21 computes this by looking through argument *n* for the first entry that has the same symbolic name and keeping track of where the match occurred (the function **index** in Figure 7-22).

```
Syntax: <number>
Code: (LDC <number>)

Syntax: nil
Code: (NIL)

Syntax: <identifier>
Code: (LD (i.j)) where (i.j) is index into E

Syntax: (<builtin> <expr>_1 ... <expr>_n)
Code: *<expr>_n|| ... ||*<expr>_1|| (<builtin>)
Example: (MPY (ADD x l) 256)
        → (LDC 256 LDC 1 LD (1.1) ADD MPY)

Syntax:(IF <expr>_1 <expr>_2 <expr>_3)
Code: *<expr>_1 || (SEL) || (*<expr>_2 || (JOIN)) || (*<expr>_3 || (JOIN))*<expr>^2
Example: (IF (null x) 1 (car x))
        → (LD (1.1) NULL SEL (LDC 1 JOIN) (LD (1.1) CAR JOIN))

Syntax: (LAMBDA (<id>_1 ... <id>_n) <expr>)
Code: (LDF)||(*<expr> || (RTN))
Example: (LAMBDA (x y) (ADD x y))
        → (LDF (LD (1.2) LD (1.1) ADD RTN)

Syntax: (LET (<id>_1 ... <id>_n) (<expr>_1 ... <expr>_n) <expr>)
Code: (NIL)||*<expr>_n || (CONS) || ... *<expr>_1 ||
        (CONS LDF) || (*<expr> || (RTN)) || (AP)
Example: (LET (x y) (1 2) (+ x y))
        → (NIL LDC 2 CONS LDC 1 CONS
           LDF (LD (1.2) LD (1.1) ADD RTN) AP)

Syntax: (LETREC (<id>_1 ... <id>_n) (<expr>_1 ... <expr>_n) <expr>)
Code: (DUM NIL) || *<expr>_n || (CONS) || ... *<expr>_1 ||
        (CONS LDF) || ( (*<expr> || (RTN)) RAP)
Example: (LETREC (f) ((LAMBDA (x m)
            (IF (null x) m (f (CDR x) (+ m 1)))) (f (1 2 3) 0))
            → (DUM NIL LDF (LD(1.1) NULL SEL
                                 (LD (1.2) JOIN)
                                 (NIL LDC 1 LD (1.2) ADD CONS
                                  LD (1.1) CDR CONS LD (2.1) AP JOIN)
                                  RTN)
               CONS
               LDF (NIL LDL 0 CONS LDC (1 2 3) CONS (1.1) AP RTN)
               RAP)

Syntax: (<expr> <expr>_1 ... <expr>_n)
Code: (NIL)||*<expr>_n||(CONS) || ... || *<expr>_1||(CONS)||*<expr>||(AP)
Example: ((LAMBDA (x y) (MPY x y)) 1 (PLUS 2 3))
          → (NIL LDC 3 LDC 2 ADD CONS LDC 1 CONS
             LDF (LD (1.2) LD (1.1) MPY RTN) AP)


Note: A||B = append(A,B). Thus (NIL)||(CONS) = (NIL CONS).
Also *<expr> is compiled form of <expr>.
```

FIGURE 7-20
Code sequences for s-expressions.

```
compile(e,n,c) = "compiler for expression e"
  if atom(e)
  then "a nil, number, or identifier"
       if null(e)
       then cons(NIL,c)
       else let ij = index(e,n) in
            if null(ij)
            then cons(LDC, cons(e, c))
            else cons(LD, cons(ij, c))
  else let fcn = car(e) and args = cdr(e) in
       if atom(fcn)
       then "a builtin, lambda, or special form"
            if member(fcn, builtins)
            then compile-builtin(args, n, cons(fcn, c))
            elseif fcn = LAMBDA
            then compile-lambda(cadr(args), cons(car(args), n), c)
            elseif fcn = IF
            then compile-if(car(args), cadr(args),
                            caddr(args), n, c)
            elseif fcn = LET or fcn = LETREC
            then let newn = cons(car(args), n)
                 and values = cadr(args)
                 and body = caddr(args) in
                    if fcn = LET
                    then cons(NIL, compile-app(values, n,
                            compile-lambda(body, newn, cons(AP,C))))
                    else " a letrec"
                    append((DUM NIL),
                           compile-app(values, newn,
                           compile-lambda(body, newn, cons(RAP, c))))
            else "fcn must be a variable"
              cons(NIL, compile-app(args, n, cons(LD, cons(index(fcn, n), cons(AP, c)))))
       else "an application with nested function"
       cons(NIL, compile-app(args, n, compile(fcn, n, cons(AP, c)))
```

FIGURE 7-21
A compiler from s-expressions to SECD code.

7.5.2 Built-in Function Applications

The code generated for the application of ***built-in functions*** consists of the sequences of SECD code needed for each argument appended to the SECD instruction that performs the function. By convention, the rightmost argument in the s-expression form is computed first, and then execution proceeds from right to left. The net effect of this is that at execution time the topmost value on the stack is the leftmost argument, with later values further down the stack.

7.5.3 Conditional Forms

The compilation of a ***conditional form*** is an SECD code list consisting of the sequence of instructions that evaluates the test expression, followed

```
compile-builtin(args, n, c) =
   if null(args)
   then c
   else compile-builtin(cdr(args), n, compile(car(args), n, c))

compile-if(test, then, else, n, c) =
   compile(test, n,
            cons(SEL, cons(compile(then, n, cons(JOIN,nil)),
            cons(compile(else, n, cons(JOIN,nil)), c))))

compile-lambda(body, n, c) =
   cons(LDF, cons(compile(body, n, cons(RTN, nil)), c))

compile-app(args, n, c) =
   if null(args)
   then c
   else compile-app(cdr(args), n,
                     compile(car(args), n, cons(CONS, c)))

index(e,n) = indx(e, n, 1)

indx(e, n, i) =
   if null(n) then nil
   else letrec indx2(e, n, j) =
        if null(n) then nil
        elseif car(n)=e then j
        else indx2(e, cdr(n), j+1) in
   let j = indx2(e, car(n), 1) in
      if null(j).
      then index(e, cdr(n), i+1)
      else cons(i, j)
```

FIGURE 7-22
Auxiliary functions to compile special forms.

by a SEL instruction, followed by two lists which correspond to the then and else expressions. respectively. Each of these sublists is computed by recursively calling **compile** with the accumulating parameter initialized to (JOIN). This places the JOIN at the end of each sublist. The auxiliary function **compile-if** in Figure 7-22 performs this process.

7.5.4 Lambda Function Definitions

Programmed functions are any functions that are defined either explicitly or implicitly by a lambda expression. This includes ***lambda expressions, let expressions,*** and ***letrec expressions.*** The former simply defines a function; the last two include both function definition and their application.

The code compiled for a lambda expression consists of a two-element list, the first of which is a LDF instruction and the second of which is a list consisting of the compiled form of the lambda expression's body terminated by a RTN. As with the conditional forms, this RTN is

inserted by recursively calling **compile** with its third argument set to (RTN) (see **compile-lambda** in Figure 7-22).

Executing such a code sequence will push onto the S stack a closure whose code pointer is pointing to the sublist following the LDF and whose embedded environment is the cell pointed to by E when the LDF is executed.

When **compile** is called recursively for the lambda's body, the namelist argument has the list of argument identifiers for that lambda appended to the namelist passed into **compile** when the lambda was discovered. This reflects the fact that once inside the body of a lambda, the top of the value list on E must correspond to a list of the lambda's arguments, with the rest of the environment list consisting of the environment that was present before the function was applied. Further, the order of the identifier names in this sublist should be the same as the order in which the values will be when the code is executed. This permits a search of the namelist to return the proper (i,j) value to encode into LD code.

7.5.5 Combined Function Definitions and Applications

Let and letrec expressions correspond to evaluating a series of expressions, associating them with identifiers, and then evaluating a new expression that references these identifiers. The code compiled for these forms must compute each of these expressions and then **cons** the results together into a list that can be passed as an argument to the lambda function implied by the in expression.

Compile does this by pushing an initial nil onto the stack, and then generating code for the rightmost let expression that will leave the result of that expression above the nil. A CONS instruction combines these two into a single element list.

The process of compiling code for each of the remaining let expressions is similar. The process goes from right to left, compiling in a sequence of code to evaluate each subexpression, and following it by a CONS to append it to the previous expression list.

When such code is executed, it leaves on the top of the stack a single element which is itself a list. The order of the elements on this list corresponds directly to the order of let expressions, from left to right.

After this, the compiled code sequence consists of a LDF followed by a code list representing the in expression and terminated by a RTN. When executed, this pushes a closure representing the in expression on top of the value list.

For a let expression the final instruction compiled into the sequence is an AP. When executed, this instruction unpacks the closure for the in expression, and starts the body, with the computed values established on the top of E.

The only difference for a letrec expression is that a RAP is used in place of an AP, and there is an extra instruction to begin the sequence, namely, a DUM. As discussed earlier, this helps build a dummy environment that RAP modifies to form the environment loop.

7.5.6 Nested Function Expressions

The final special form handled by the compiler is when the expression in the function position of an s-expression is something other than a keyword. The compiler function **compile-app** handles this case. As with let, the arguments are evaluated one at a time from right to left and **cons**ed into a list so that the leftmost one is first. Then the code for the function subexpression is compiled and appended to the right of the argument evaluation code, again just as for let. This subexpression could be anything, as long as when the code is executed it places a closure on the top of the stack. In most cases this subexpression will be a (lambda...) expression or an identifier which has been bound earlier to a (lambda...). Again there is no error checking to see if this is in fact the case.

The final instruction compiled for this type of an expression is an AP. When executed, it will take the list of argument values and the closure and evaluate the combination.

7.6 SAMPLE COMPILATION

This section diagrams the code that would be compiled for yet another variation of our familiar factorial function. This definition includes an extra let at the beginning with an assignment of "1" to the identifier *one* just to show the differences between that and letrec. Not all calls to compiler functions are shown; the calls chosen are those with significance, such as major changes in the namelists used to determine where variables are in the environment.

The original abstract program to compile is:

```
let x=3 and one=1 in
    letrec fact(n, m)=
        if (eq n 0) then one
        else fact(n− one, n× m)
    in fact(x,one)
```

The s-expression equivalent of this is:

```
(let (x one) (3 1)
   (letrec (fact)
      ((lambda (n,m) (if (= n 0) one (fact (− n one) (× n m)))))
      (fact x one)))
```

The compilation proceeds as follows:

```
compile((let...),nil,(STOP))
 ⇒ (NIL.compile-app((3 1), nil,
     compile-lambda((letrec...), ((x one)), (AP STOP))))
 ⇒ (NIL.compile-app((3 1), nil,
     (LDF.(compile((letrec...), ((x one)), (RTN))(AP STOP)))))
 ⇒ (NIL.compile-app((3 1), nil,
     (LDF (DUM NIL.compile-app(((lambda...)), ((fact)(x one)),
        compile-lambda((fact x one), ((fact)(x one)),
        (RAP RTN))(AP STOP))))))

 ⇒ (NIL.compile-app((3 1), nil,
     (LDF (DUM NIL.compile-app(((lambda...)), ((fact)(x one)),
        (LDF
           (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
           RAP RTN)
      AP STOP)))))

 ⇒ (NIL.compile-app((3 1), nil,
     (LDF (DUM NIL.compile-app((), ((fact)(x one)),
        compile((lambda...), ((fact)(x one)),
           (CONS LDF
              (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
              RAP RTN)
              AP STOP))))))

 ⇒ (NIL.compile-app((3 1), nil,
     (LDF (DUM NIL.compile-app((), ((fact)(x one)),
        compile-lambda((if...), ((n m) (fact) (x one)),
           (CONS LDF
              (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
              RAP RTN)
              AP STOP))))))

 ⇒ (NIL.compile-app((3 1), nil,
      (LDF (DUM NIL.compile-app((), ((fact)(x one)),
        (LDF.(compile((if...), ((n m) (fact) (x one)), (RTN))
        (CONS LDF
              (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
              RAP RTN)
              AP STOP)))))))
```

```
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app((), ((fact)(x one)),
      (LDF
         (LDC 0 LD (1.1) EQ SEL
            (LDC 1 JOIN)
            (NIL LD (1.2) LD (1.1) MPY CONS
                LD (3.2) LD (1.1) SUB CONS
                LD (2.1) AP JOIN)
            RTN)
      CONS LDF
         (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
         RAP RTN)
   AP STOP))))))

⇒ (NIL.compile-app((3 1), nil,
   (LDF (DUM NIL LDF
      (LDC 0 LD (1.1) EQ SEL
          (LDC 1 JOIN)
          (NIL LD (1.2) LD (1.1) MPY CONS
              LD (3.2) LD (1.1) SUB CONS
              LD (2.1) AP JOIN)
          RTN)
  CONS LDF
      (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
      RAP RTN)
  AP STOP)))))

⇒ (NIL LDC 1 CONS LDC 3 CONS LDF
   (DUM NIL LDF
      (LDC 0 LD (1.1) EQ SEL
          (LDC 1 JOIN)
          (NIL LD (1.2) LD (1.1) MPY CONS
              LD (3.2) LD (1.1) SUB CONS LD (2.1) AP JOIN)
          RTN)
  CONS LDF
      (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
      RAP RTN)
AP STOP)
```

This final code is shown as a linked list of cells in Figure 7-23. For compactness, this figure takes any car field that contains a pointer to a terminal cell (tag=integer) and writes the value of that terminal where the pointer would go. In reality, the other terminal cell is still there.

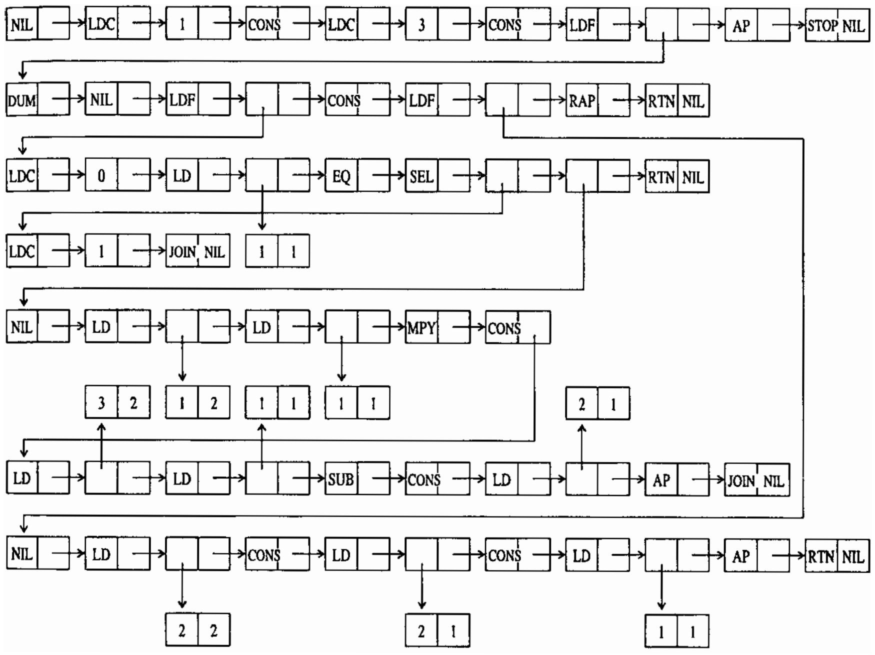


FIGURE 7-23
Shorthand cell equivalent of sample code.

7.7 THE FUNARG PROBLEM

(Bobrow and Wegbreit, 1973; Georgeff, 1982)

The SECD Machine uses stacks for almost everything. This leads to easy-to-understand compilers and relatively simple hardware implementations. It is also very similar to the abstract machines used for many conventional languages such as Pascal.

For someone schooled in conventional architectures, however, a direct implementation of the SECD Machine seems to have some significant inefficiencies. Each time an object is pushed onto the S stack, for example, the F register is incremented and two writes to the cell are performed (one for the data in the car and one for the cdr pointer). This work is doubled if the object being pushed is a new one, for example, the result of an ADD instruction.

In contrast, a conventional implementation of a stack involves simply incrementing a stack pointer and writing the data to memory. This is far cheaper in terms of machine cycles than the above procedure.

Even worse is the list of lists found on the environment. Accessing an argument via a LD instruction requires a doubly nested count down through these lists. A more normal implementation, such as for Pascal, would use a stack of contiguous locations for arguments and often a ***display*** of pointers into this stack to locate the beginning of an argument list for each function call (see Figure 7-24). With such an implementation, accessing an argument takes only two indexed memory accesses (even fewer if the display is implemented as an array of registers inside the CPU).

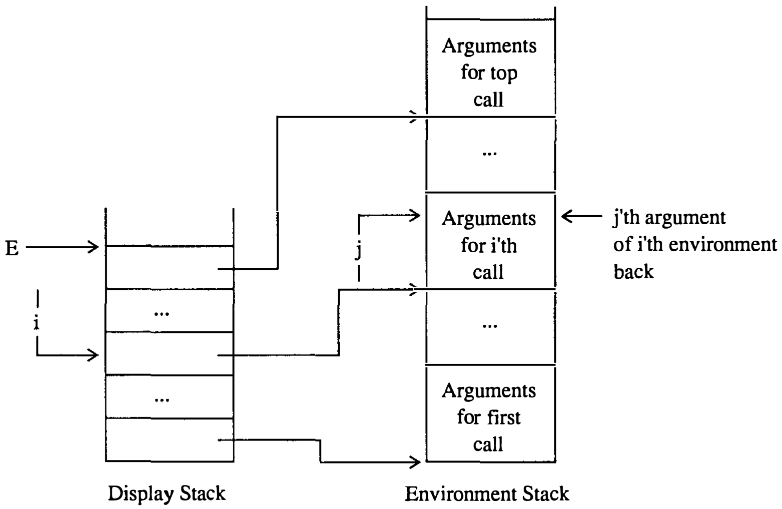
Given this, why would anyone use the SECD method of building stacks from linked lists? A good part of the answer stems from consideration of what happens when closures are generated by one function and passed as arguments to others, and particularly when identifiers within the closure function code have been bound values by functions which called the original closure-generating function. The handling of the stack and the environment becomes crucial, and if care is not taken, all kinds of strange results can occur. This has been studied extensively, and goes by the name of the ***funarg problem.***

Consider, for example, the abstract program:

let f=(let y=4 in ($\lambda z \mid y\times z$)) in f(3)

The innermost let builds a valuelist ((4)). The result it passes to the outer let is a closure with a function code equivalent to ($\lambda z \mid y\times z$) and an environment ((y.4)).

Consider what would happen if we built stacks, particularly the environment stack, as is done in conventional machines. Pushing an item simply causes an increment of the stack pointer and a store into the designated memory cell. Whatever is there from before is overwritten. Popping an object simply decrements the pointer, permitting the next push to overwrite the prior value. In such an implementation, at the exit from the code for the innermost let, the equivalent of the E register would be decremented. Now the code for the outermost let builds a new environment (a closure for *f*) *in exactly the same* memory cells as ((4)) took. The value "4" is lost, and the references to *y* inside the code for the inner let will pick up the closure and not the proper value for *y*. The value 4 is "lost."



Note: Both stacks implemented in sequential memory cells.

FIGURE 7-24
An alternative to a list environment.

This is the funarg problem, and it can occur whenever a function which produces a function result of some sort contains ***free variables*** which are given values by calling functions.

Note what happens in this example with our linked-list implementation for stacks. The valuelist ((4)) is not overwritten by the outer code. Instead the machine allocates new storage for the new valuelist for *f*. All pointers in this closure to the old environment thus remain valid, meaning that executing the code in the closure will retrieve the proper value of 4 for the variable *y*.

Keeping linked lists that are not overwritten is not the only solution to the funarg problem. Other solutions include separate heaps for storing environments when potential problems might occur, implementing duplicate environment stacks (see Harrison, 1982), ***spaghetti stacks*** or ***cactus stacks*** instead of conventional stacks (see Bobrow and Wegbreit, 1973), or even doing explicit code copying and substituting argument values as was done in our very early lambda calculus interpreters. A later chapter on real machines for functional languages will amplify on some of these.

7.8 OPTIMIZATIONS

There are quite a few optimizations that can be made to this basic SECD architecture and compiler that would permit faster execution of programs on it. Some of these, such as recovering memory cells that have been allocated but are no longer needed, will be discussed in the next chapter. Others, however, deserve at least a brief mention here. The interested reader is encouraged to consider how these optimizations might be reflected in either the SECD architecture, the compiler, or both.

7.8.1 Improved Constant Handling

As currently defined, lists that are made up of constants are painstakingly built up one at a time through a series of LDC and CONSs every time they are needed in the program. This could be considerably simplified by having the compiler check each s-expression before it compiles code for it. If the s-expression is made up of nothing but constants, the compiler could generate a single LDC followed by the unevaluated list (with numbers converted into internal representation, of course). Executing this would result in a pointer to the appropriate list being pushed to the stack without excessive make-work.

This would also be the technique used to implement the ***quote*** function found in many functional languages.

7.8.2 Simplifying Conditionals

When compiled by the previous compiler, most functions (particularly recursive ones) have an outermost structure of the form:

((**basis test**) SEL ((**basis case**) JOIN) ((**recursion**) JOIN) RTN)

The SEL pushes the cell address of the RTN onto the dump. Whichever case is selected will then execute, with a final JOIN to pop the appropriate return address from the stack. In the case of code sequences like the one above, this return takes the program to a RTN which then pops the dump again.

In many circumstances this double pop can be avoided by replacing the JOINs by RTNs, and replacing SEL by an instruction which does a similar selection function but does not push anything to the dump. Such an instruction might be called a ***TEST*** and operate something like:

TEST: (x.s) e (TEST ct.c) d→s e c? d where c?=ct if $x \neq 0$ and c?=c if x=0

Note that the false code need not be a separate sublist, but simply the continuation of the code after the TEST.

With this instruction the above typical code structure can be expressed as:

((**basis test**) TEST ((**basis case**) RTN) (**recursion**) RTN)

7.8.3 Simplifying Apply Sequences

Tail recursion occurs when the last thing a function does is call some other function with a new set of arguments. A very common piece of code generated by the above compiler for such circumstances looks like (...AP RTN). When executed, the AP pushes values for S, E, and C onto the dump. The return from the function called by the closure invoked by AP will pop these values off the dump and reestablish them in the proper registers, only to have the same values overwritten by the RTN following the AP. Three extra pushes and pops to the dump have been performed, with no useful effect.

An interesting extension to the SECD architecture that avoids this double dump pop might take the form of a ***DAP*** (for ***Direct APply***) instruction. This instruction would have a register transition that performs only the environment modifications from the AP and leaves the dump alone:

DAP:((f.e′) v.s) e (DAP c) d→NIL (v.e′) f d

Thus the code sequence (...AP RTN) would be replaced by the much more efficient (...DAP). No extra items are pushed to the dump, and we rely on the RTN instruction at the end of the function called by DAP to return control to the function which called the code containing the DAP, with no intermediate stops. In a sense we have short-circuited the intermediate call/return.

7.8.4 Avoiding Extra Closure Construction

While DAP avoids the double dump push/pop, that is not the end of the line for optimizations. Consider the very common sequence:

(...LDF (...**function code**...RTN) AP RTN)

DAP optimizes this to

(...LDF (...**function code**...RTN) DAP)

The LDF builds a closure that is immediately taken apart by DAP. The environment modified by the DAP is the same one that is already in effect.

Consider instead what would happen if we invent a new instruction ***AA*** (***Add Arguments***) with the following register transitions:

AA: (v.s) e (AA.c) d→s (v.e) c d

Now the above code sequence could be replaced by:

(...AA...**function code**...RTN)

We have avoided the extraneous closure-building of the prior sequence. Further, on machines where branches are expensive (as is the case for most high-performance machines), we have also eliminated the change in the C register still present in the DAP form.

7.8.5 Full Tail-Recursion Optimization

An astute reader might detect that there is one further optimization that could be performed for tail recursion. In all the above cases we are still consing a new argument list onto the current environment, the top of which is a sublist of arguments for the current function call. In most cases these latter arguments will never be referenced again, so the storage associated with them becomes wasted space. If we could avoid leaving these unused arguments on the environment, we could open up the possibility of recovering the storage.

An expansion to AP (or AA) that avoids this growth would "replace'' the topmost sublist of E by the new argument list from S. We call this instruction TRAP, for ***Tail-Recursive APply.***

TRAP: ((f.(e′.e″))v.s) e (TRAP c) d→NIL (v.e″) f d

or even better:

TRAP: ((f.e') v.s) e (TRAP c) d→NIL **rplaca**(e',v) f d

As before, the **rplaca** function replaces the car subfield of the top cell pointed to by the E register by a pointer to the v list.

This TRAP would replace the DAP defined above in all places where the called function does not need reference to the current arguments. In turn, this has two effects on the compiler. First, the compiler must verify that any function called either directly or indirectly by TRAP does not in fact use any of these arguments. While tricky to determine in general, there is one extremely common case where such determination is relatively easy. This is ***self-recursion,*** where the function being called is the same as the calling function. In such cases there is no way to refer to the old arguments, so they can be deleted in complete freedom.

The second compiler modification involves the namelist passed recursively to ***compile*** to compile the function being called. This namelist must reflect the modified valuelist at the time the code will be invoked. In reference to Figure 7-21, this involves something like **cons**(**car**(*args*), **cdr**(*n*)).

A further optimization along the lines of the AA would lead to an instruction like ***MA*** (***Modify Arguments***), with register transitions of the form:

MA: (v.s) e (MA.c) d→s **rplaca**(e,v) c d

This instruction replaces the first element of the current environment by the argument list currently the top element of the stack.

There are several ways of using this instruction. If, as with the AA instruction above, we do want to return to the current function's code, we could simply follow the MA with the new function's code, terminated with a RTN.

If the function is self-recursive (the new function called is the same as the current one), a more interesting possibility emerges. We can simply let the **cdr** of the memory cell whose **car** holds the MA instruction code point back to the beginning of the function's code sequence. Then after executing the MA, the machine branches right back to the beginning of the program for another recursive iteration but without adding anything to the environment or the dump. For very highly recursive programs, this converts the program's execution to essentially a ***do loop,*** and is something that is used in most real functional-language compilers.

The only problem with this is some increase in compiler complexity. The compiler must now recognize when the function being invoked is one for which code already exists, and retrieve the starting address of that code sequence. This requires that the namelist argument be expanded to include starting code addresses, much like a conventional compiler does with a ***symbol table.***

7.8.6 Example

Figure 7-25 diagrams an optimized version of the SECD code for the factorial program of the prior section. Virtually all the techniques described above are used.

7.9 PROBLEMS

1. Generate the SECD code for the following. (Note that problems b and c must be converted into s-expression form first.)
   - **a.** (let (x y) (1 2) ($\times$ x y))
   - **b.** let f=(let y=4 in (λx $y\times x$)) in f(3)
   - **c.** letrec f=(λn | if n then 1 else $f(n-1)+f(n-2)$) in f(3)
   - **d.** (Long!) the towers of Hanoi function of Chapter 3.
2. Verify that each of the examples of Figure 7-20 actually execute as expected.
3. This problem will give insight into the process of optimizing an architecture. Mentally execute the factorial program of Section 7.6 and then answer the following questions.
   - **a.** How many of each type of SECD instruction are executed? (*Hint:* Count number each time through a full recursion.)
   - **b.** How many memory cells are allocated during execution? (*Hint:* For each instruction type used in (a), how many cells are allocated?)
   - **c.** What is the average number of memory cells allocated by the average instruction in this program?
   - **d.** How many memory references (reads and writes) are made (include instruction reads)?
   - **e.** How big a factorial could be computed if we had, say, only 1 million words of memory left at the start of execution of the program?

```
(NIL LDC (3 1)
  AA
  DUM NIL LDF
    (LDC 0 LD (1.1) EQ TEST (LDC 1 RTN)
    NIL LD (1.2) LD (1.1) SUB CONS
    LD (3.2) LD (1.1) SUB CONS
    MA.*) ''where * is the starting address of (LDC 0...''
  CONS LDF
  (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) DAP)
   RAP STOP)
```

FIGURE 7-25
Optimized SECD code for factorial.

   f. Assuming a machine that can execute 1 million SECD instructions per second, how long will it take to exhaust this 1-million-word memory?

4. If one were to implement the SECD memory model out of a conventional memory with 32-bit words, how many bits could be allocated to integers, and how big could the memory be (in cells) if:
   - One memory word was one cell.
   - Two memory words made up one cell.
5. Draw what storage would look like if you "popped" the stack of Figure 7-4(*b*) twice, pushed 103 on, and then pushed back on the **cdr** of the second element that was popped off. Show all memory cells, indicating which ones may become garbage after the operation.
6. Consider the expression:

   let **f**(x)=x+1 in let **g**(x,y)=**f**(y)×x in **g**(3,4)

   a. Show what the value list would look like (both as an s-expression and as linked cells in memory) when starting the evaluation of **g**(3,4).

   b. Convert the abstract program into s-expression form and then into a program list for the SECD Machine.
7. Write an SECD program for:

```
append(x,y)=if atom(x)
              then y
              else car(x).append(cdr(x),y)
```

   a. How many new memory cells are used up during the computation of **append**((1 2), (3 4)). [Don't count the cells for the program or for the initial cells for the lists (1 2) and (3 4).]

   b. If you had 100,000 cells available, how big a list could you append to (3 4)?
8. There is a dichotomy between functions called by AP and built-ins. The latter take their n arguments from n consecutive elements of S. The former require their n arguments be packaged in an n-element list on the front of S. Devise a mechanism (and necessary changes to current instructions) that permits a programmed function to take its arguments directly off the stack, lump them into a list, and augment its environment, all internally to itself. This saves each call to the program from having to do this. (*Hint:* Defer pushing S to the dump at an AP, and add an instruction ARGS to handle the argument building.)
9. Develop optimized SECD code for the expressions of Problem 1.
10. Modify the compiler of Figure 7-21 to include the instruction DAP and its optimization techniques.
11. Modify the compiler of Figure 7-21 to include the instruction TRAP and its optimization techniques.
12. (Small Project) For your favorite microprocessor:

    a. Decide how to simulate the basic SECD memory cell.

    b. Decide how to allocate S, E, C, D, and F to real registers.

    c. For each SECD instruction, show an equivalent sequence of real instructions.

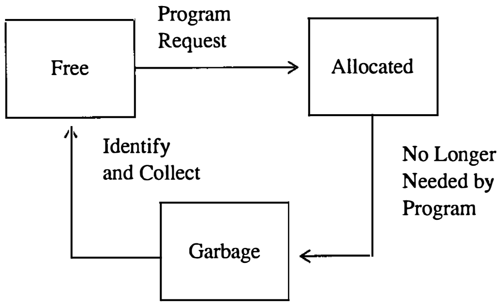
CHAPTER 8

MEMORY MANAGEMENT FOR S-EXPRESSIONS

The SECD architecture as stated so far is fairly simple to understand and implement, but suffers from a variety of problems that make it unsuitable for any real applications. Most serious is the lack of ***garbage collection***; namely, once a cell is used, it is never reused, even if it is clearly free for such use. Once the F register is incremented beyond a cell, that cell becomes permanently allocated to the computation that first requested it, even if the computation is long gone.

This is particularly true of the very short-term results of arithmetic and logical operations. A value is computed (allocated to a cell), used (pushed on the stack), and no longer needed (popped off), all in the space of perhaps a half-dozen instructions. Given any realistically sized memory, any non-toy SECD program would run out of storage in literally seconds.

This chapter investigates more efficient methods of managing a pool of memory cells so that a program can continue to request cells as needed, never explicitly release them, and yet still reuse storage that is no longer linked into the current SECD registers in any way. The basic approach is to either remember or compute the state of each memory cell in the memory. As pictured in Figure 8-1 there are three such states: ***free, allocated,*** and ***garbage.*** The transition from free to allocated occurs when a cell is needed (F register incremented). From allocated to garbage oc-



For the basic SECD ISA, cell x is:

- Free if x>F
- Allocated if x≤F
- Garbage if x≤F and not in lists from S, E, C, or D

FIGURE 8-1
States of a memory cell.

curs when a cell can no longer be reached by any pointer chained to the machine registers. Both of these occurred in the basic SECD ISA of the last chapter, although there was no attempt to identify the latter. What we will address here is the third transition, namely, mechanisms to convert garbage back into free cells.

In particular, we will address three major topics: alternative ways of allocating cells, methods for locating and freeing garbage cells, and finally some discussion on alternative representations for lists that either reduce the amount of memory usage or permit other special features to be supported. Three major techniques will be covered for the process of locating garbage: the ***mark-sweep*** method, whereby all of memory is scanned; the ***reference count*** method, whereby each cell keeps track of the number of other cells that point to it; and the ***compaction*** method, whereby cells still in use are copied and compacted into a contiguous block of storage. All three are used in real systems, with variations of the compaction approach (those based on ***Baker's algorithm***) being of most importance. The interested reader is referred to Cohen (1981) for one of the first detailed survey of such techniques in the literature.

8.1 ALLOCATION

The most visible part of memory management is the ***allocation*** of a particular memory cell to receive some value. In our basic SECD Machine we assumed that the ***F register*** provided the basic hook to support this. When an instruction's execution requires a new cell, the F register provides an index into the next available one. Typically this occurs when there is a **cons** operation of some sort implied by the instruction, either explicitly as in a CONS or implicitly as in some other built-in operation such as ADD, which pushes a value on a stack.

The actual allocation process consists of several stages as pictured generically in Figure 8-2. First is a test to see if there is any easily allocatable memory. If there is not, either an error condition is flagged (as in our simple SECD Machine) or a ***garbage collection*** process (as discussed in the next sections) is triggered to locate the garbage and make it allocatable. Finally, if storage is available, some particular cell must be allocated and the F register updated.

```
allocate(f) = "return address of a free cell"
   if no-more-free-space(f)
   then allocate(garbage-collect(f))
   else get-next-free-cell(f)
   where
      no-more-free-space(f) = T if no free cells available
      and garbage-collect(f) collects all garbage cells
      and get-next-free-cell(f) gets next available free cell

Example: cons(a, d) = let f=allocate(F) in
        "following is sequential storage modifying code:"
        car(f) <— a;
        cdr(f) <— d;
        tag(f) <— "allocated, of type cons";
        Return f ("and update F register");
```

FIGURE 8-2
Generic free cell allocation function.

There are several methods of implementing each of these allocation tests or actions. Figure 8-3 diagrams three of the more common ones. Each is discussed in the following subsections. They differ in the time required to allocate a cell (both in total amount and in predictability) and in demands placed on the garbage collector to recover storage.

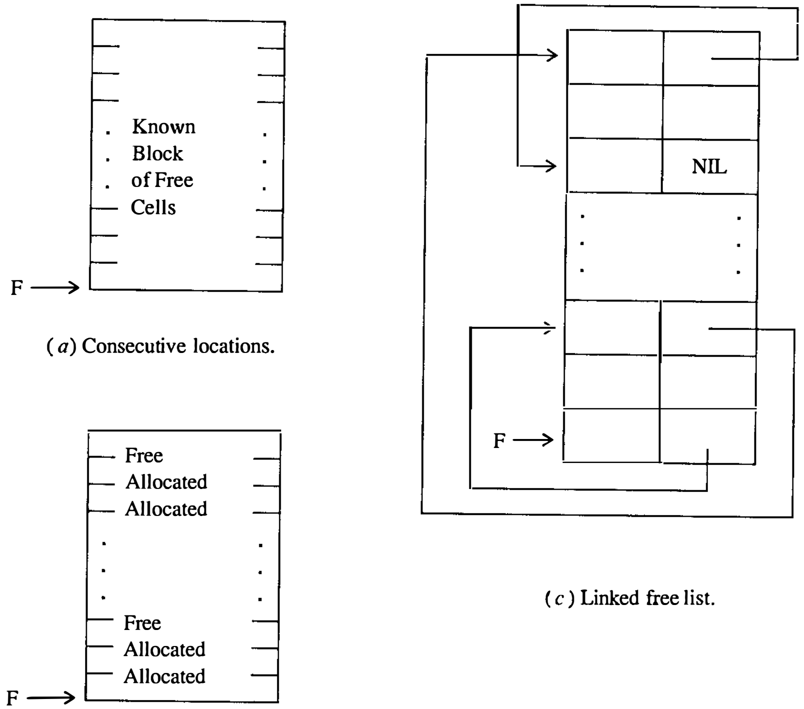
8.1.1 Free Cells in Consecutive Locations

The simplest method of allocation is what was discussed for the simple SECD Machine. All memory with addresses higher than F [see Figure 8-3(*a*)] is assumed free; all memory at or below F is allocated. In terms of Figure 8-2, the function **no-more-free-space** is simply a comparison of F to the highest available address in memory. If F equals that point, then there is no directly available memory.

If F is less than that point, the function **get-next-free-cell** is a simple incrementation of F.

The primary advantage of this approach is speed. Both of these operations can be done in parallel (or even integrated together) so that at most a single machine cycle is needed for them. No references to memory are needed to perform the tests.

The disadvantage is, of course, the inability to reuse memory below the F register. This means that any associated garbage collector must find blocks of contiguous cells that are all free. Often this in turn requires ***scavenging*** of all useful data out of a large block of memory and copying it into a compacted area, above which there is all free cells. A later section will discuss such a technique.



(*a*) Consecutive locations.

(*c*) Linked free list.

(*b*) Scan to next free cell.

FIGURE 8-3
Three free space allocation strategies.

8.1.2 Free Cells Individually Marked

A variation of this simple sequential approach permits reuse of memory cells if other mechanisms are in place to identify when they are free. Basically, each cell contains in its tag field an extra bit, the ***mark bit,*** which has two values: "free" and "allocated." Initially, all cells are marked free. As cells are allocated, this bit is changed to allocated. When some garbage collection mechanism determines that an allocated cell is actually garbage, its mark bit is also changed to free.

This means that in general there is no guarantee that a block of storage above some address is all free (see Figure 8-3). Instead, the **allocate** function must scan through memory, checking the mark bits.

In terms of the functions of Figure 8-2, the function **no-more-free-space** becomes a relatively complex loop that repetitively increments F, testing the mark bit of the associated cell, until either one is found that is marked free, or all of memory has been scanned.

The **get-next-free-cell** function changes the mark bit of the selected cell to allocated and returns its address.

The advantage of this approach is that it permits reuse of cells anywhere in memory without special treatment. The glaring disadvantage is, of course, unpredictable and potentially excessive execution time. It is impossible to predict exactly where in memory the next cell with a free mark bit could reside. This is compounded by the need to read, test, and write to memory inside the **allocate** process (unlike the prior method, which did not need to access memory at all).

8.1.3 Linked Free List

A third way of implementing Figure 8-2 represents a trade-off between the other two [see Figure 8-3(*c*)]. Here some other mechanism in the machine, such as the garbage collector, has taken all the free cells in memory and linked them together in a conventional list. The order of cells on the list is immaterial. Such a list is often called the ***free list.*** The cdr field of one cell points to the next available cell. The car fields are not used. The F register points to the first cell in this list.

The allocation process is now almost as fast as the first method. The **no-more-free-space** function involves simply testing the F register. If it contains a nil pointer, then there is no more easily accessible free space. If F is not nil, then the cell referenced by F is free. The **get-next-free-cell** function then involves returning the current value of F as the free cell, and replacing F by **cdr**(F). This sets up F to point to the next available cell.

Time of a typical allocate is thus approximately one memory read, and is constant unless the free list is empty. While this method is somewhat slower than the first, it is nowhere as bad as the scanning approach. The major disadvantage is that the garbage collection process must link all newly recovered memory cells into this list, a process which involves a memory write per cell.

8.2 MARK-SWEEP COLLECTION

What happens after any of the allocation methods of the last section have run out of obviously free cells? Either the machine can error stop, or it can invoke a mechanism that attempts to identify allocated storage that is no longer in use (***garbage***) and convert it back to allocatable free cells (***garbage collection***). This section describes variations of the simplest such procedure, called the ***mark-sweep method.*** It is compatible with either of the last two allocation methods of the previous section.

Implementing mark-sweep relies on dedicating at least one bit in each cell. As described in the preceding section, this bit, the ***mark bit,*** records whether or not the cell is free. The basic approach is first to unmark all of memory (resetting all mark bits to 0) and then trace through all cells that are still of use, ***marking*** them by setting the mark bit. All storage remaining unmarked is thus free.

After this point one can either return to the suspended program and let the allocation mechanism scan through memory looking for unmarked cells, or first ***sweep*** all the unmarked cells into a ***free list.***

The following subsections describe variations of this basic procedure. It should be noted that many of the descriptions of the actual algorithms are decidedly nonreferentially transparent. The functions are executed for their side effects of marking and sweeping specific memory cells. This makes their implementation on real machines a nontrivial exercise, particularly when such complications as interrupts, multiprogramming, and other time-sensitive operations are considered.

8.2.1 Marking

The purpose of the marking algorithm is to identify all cells that are still of use to the program. Such cells are those that are somehow linked to one of the main machine registers S, E, C, or D. Any cell that is not in a list linked to one of these registers can never be reached by an SECD program, and is thus of no possible use to it, regardless of what is stored in the cell or how it got there.

The key requirement to support this marking process is at least one ***mark bit*** per cell. As described earlier, this will represent the state of the cell, either "free" or "allocated."

For a variety of historical reasons that will make sense later, these values are often called ***colors,*** making the marking process equivalent to one that "colors" each cell. We will assume initially that white is to be the color of a free cell (a 0), and black the color of an allocated cell (a 1).

Figure 8-4 lists some functions used in the basic marking process.

```
color(i, c) = "set mark bit of cell i to c"
  begin
  mark-bit(i)←c;
  return i;
  end;

color-all-cells(i, c) = "color all cells above location i to c"
  if i>"top-of-memory"
  then 0
  else color-all-cells(color(i, c) + 1, c)

mark(i, c) = "color all cells accessible from i as c"
  if mark-bit(i) = c "stop on marked cell"
  then c "return color for convenience"
  else let ix = color(i,c) in
       if tag(i) = terminal
       then c
       else mark(cdr(ix), mark(car(ix),c))
```

FIGURE 8-4
Basic marking functions.

The first step after discovering that all known free space has been exhausted is to call **color-all-cells**(0,0) to set the mark bits of all memory cells to white. The philosophy is to assume that all cells are free until proven otherwise.

Next the function **mark** is called four times, once for each of the four SECD registers. Each call has the name of a different register and a color (black in this case) as its arguments. The register value is used as a starting pointer into a memory list, all of whose cells should be colored black (definitely allocated).

The function **mark** runs down the list pointed to by the register, coloring all cells it encounters. It has several tricky aspects to consider. First, to prevent endless loops (as when tracing recursive environments), it must first test a cell to see if that cell has already been marked to the appropriate color. If so, then we will not attempt to mark it or its descendants again. For convenience, the value returned from **mark** is the color being marked.

Next, the tag of the cell directly influences how marking proceeds. If the cell is a terminal cell, there are no embedded pointers, and no further cells need be traced. As before, we return the color being marked as the function's value in this case. If the cell is a nonterminal, however, there are two pointers embedded in it, both pointing to potential sublists, both of which must also be traced. This requires a doubly recursive call as shown, once for the car field and once for the cdr field. Note that returning the color being marked is convenient for handling the double recursion.

Finally, in both cases we must color the cell just reached. For the case of a nonterminal cons cell this must be done *before* tracing the car or cdr field to prevent potential infinite loops. The "let *ix*=**color**(*i*, *c*) in" statement specifies this side effect, but in order for it to work at the proper time, it must be evaluated before the nested **marks**. This requires an applicative-order evaluation.

After a return from calling **mark** with one of the SECD registers, we are guaranteed that all cells that can be reached from that register have been colored black, and are thus known to be still in use. Repeating for all four registers thus marks all useful cells, leaving all other cells colored white.

8.2.2 Basic Sweeping

If we allocate free cells by scanning, the above marking process is all that is needed. Allocation can resume by starting F at 0 and incrementing it until either a cell with a white color is found or memory is scanned once more. In the latter case there truly is no free memory, and the program must stop.

If we allocate free cells off of a free list, another step, called the ***sweep*** step, is needed before the suspended allocation process can be resumed. This step must collect all free cells into a list pointed to by F. Figure 8-5 diagrams this procedure. It is also a simple loop through all of

```
F-register = sweep(0,0,nil)
  whererec sweep(i, c, link) = "list of all cells colored c above i in memory"
    if i>top-of-memory
    then link
    elseif mark-bit(i)≠c
    then sweep(1 + i,c,link)
    else "its the right color—collect it"
      "Again some bit writing to memory"
      sweep(1 + i, c, rplacd(i, link))
```

FIGURE 8-5
Basic sweeping function.

memory, testing each cell's color. When a cell with the appropriate free color is found, a side effect-based operation replaces its cdr field by a pointer to the rest of the list. At completion, the address of the final, linked in cell is returned as the new F register value. Note the use of ***rplacd*** here to write directly into a cell's cdr field. A **cons** operation is not proper.

Finally, looking at this sweep step and the previous mark step, we see that memory is being scanned up to three times: first to color all cells, then to mark the cells in use, and then to sweep up the remaining ones. For large memories this is obviously a very time-consuming operation.

One optimization to remove at least one of these scans is to specify that no operation other than the garbage collector uses the mark bits. Then, in the process of performing the final sweep, each cell that is encountered that has a black (allocated) tag can have the tag rewritten to white. Thus, the next time the free cells are exhausted, we are guaranteed that all cells are already colored white and the initial bulk **color-all-cells** pass in the mark process can be skipped.

While this eliminates a scan through memory, it does require a memory write into each and every cell of memory; black cells are colored white, and white cells have their cdr fields set. At least the former writes can be avoided by noticing that the binary values assigned to white and black are totally arbitrary. Thus, during the process of writing to a free cell's cdr field, we could simultaneously flip its mark bit to the opposite binary value. At the end of this sweep all cell mark bits throughout memory thus have the same binary value, namely, that which we associated with black (allocated) at the beginning of the garbage collection. If at the end of the sweep we flip our definition of which binary values correspond to black and white, respectively, then at the next pass we are again guaranteed that all cells are white, and thus can immediately start tracing the registers.

8.2.3 Marking Complexity Problems

There are some hidden but important complexity issues with implementing the double-recursion branch of the **mark** function:

```
    else mark(cdr(ix), mark(car(ix, c))
```

The innermost recursion is not ***tail recursive***; no amount of optimization or rewriting can convert the recursive call into an inexpensive looping operation. Regardless of how it is implemented, we must save the current state of the computation (at least the current value of *ix*) on a stack or equivalent structure before recursively calling to handle the car field. Further, this inner call may itself encounter a nonterminal cell, requiring yet another set of values to be placed on this stack. This can go on indefinitely, with the stack growing to an unpredictable depth. The worst case occurs when a register's list is of the form ((...(1.nil).nil)...).nil). Each call to **mark** causes a recursive call to handle the car. Only after the 1 is reached does the stack start shrinking.

The basic problem with this is that if we are out of memory to begin with, where is this extra memory for the stack coming from? If the list happens to use up all of memory, the number of recursions that need to be stacked can approximate the number of cells in the memory!

There are several ways of attacking this problem. First, one can test the cells at the car and cdr of a nonterminal first, and if they are terminal, they can be marked immediately, without requiring a deliberate recursion to come back to them later. While this is an important optimization technique, it does not solve the problem; we can always construct a list which defeats such tests and maximizes the double calls.

Another technique uses a finite-length wraparound area of memory to hold the inner recursion information. If it overflows, new entries overwrite the earliest ones. At the end of processing, if the stack has overflowed, a second scan of memory begins, looking for marked cells whose descendants are unmarked. When such a cell is found, a new marking cycle begins on it, using the stack as before to hold untraced branches. Only when a scan of all memory reveals that all marked cells have marked successors does the marking process stop. While this fixes the maximum amount of memory needed for the marking stack to a finite amount, it does open up the possibility of scanning memory a huge number of times.

One obvious way to help minimize this repetitive scan is to keep the smallest address that is "forgotten" as a result of the original marking process. No addresses smaller than it need be rescanned.

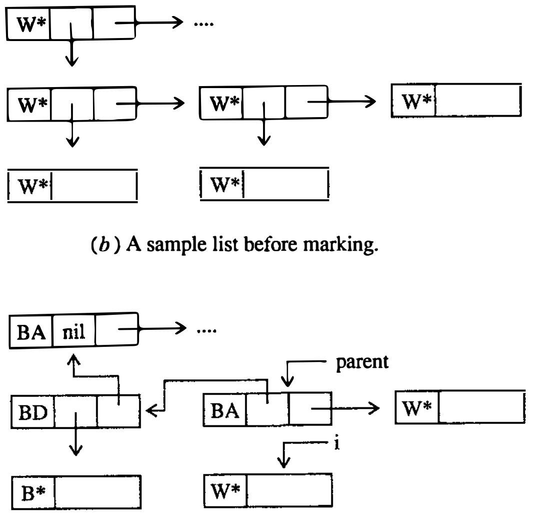
An entirely different technique, called the ***Deutsch-Schorr-Waite algorithm*** (Schorr and Waite, 1967), can avoid a stack altogether by dynamically reversing the pointers in the lists being traced. When a terminal node is reached, the marking process can use these reversed links to move backward up the list to the most recent cell with an unmarked child, unreversing the links as it goes.

Figure 8-6 diagrams a simple form of this procedure. While there are two mutually recursive functions, a careful review of the code reveals that both are tail recursive and could be implemented without stacking anything. The only additional information needed here is yet another tag bit in each cell (the ***direction tag***) and an extra argument in the function call to hold the parent of the current cell being marked. This new tag in-

```
mark(i, parent, c) = "color all cells accessible from i as c"
   if mark-bit(i)=c
   then backup(i, parent, c)
   else let ix=color(i,c) in
         if tag(i)=terminal
         then backup(i, parent, c)
         else let child=car(i) in
               mark(child, rplacdtag((rplaca(i,parent), A), c)

backup(child, parent, c) = "back up from child"
   if null(parent) then c
   else dtag=direction-tag(parent)
   and pcar=car(parent)
   and pcdr=cdr(parent) in
        if dtag=A "see which parent field was reversed"
        then "reset parent's car, and then reverse parents cdr"
           mark(pcdr, rplactag(rplacd(rplaca(parent,child),pcar),D),c)
        else "reset parent's cdr and backup"
              backup(rplacd(parent, child), pcdr, c)
```

(*a*) A link reversing marker.



(*b*) A sample list before marking.

(*c*) An intermediate point in the marking.

Tag = Color, Direction Where
Color = White/Black (W/B)
Direction: Ancestor/Descendant (A/D)
*Don't care (either one)

FIGURE 8-6
A link reversing marker.

dicates which link, the car or the cdr, has been reversed and now holds a pointer to the parent of this cell. The function **backup** uses this information to reset a link when a terminal node is reached, and a back up of the tree is desired. The function ***rplactag*** functions like **rplaca** or **rplacd**; it returns the address of its first argument, but it has the side effect of changing the tag field of the specified cell.

The marking process is now started by an expression of the form **mark** (register, nil, color), and terminates when **backup** finds a nil parent pointer.

8.2.4 Multicolor Marking

Both of the above marking procedures, finite queue and link reversal, have relatively complex data structures and are relatively sensitive to being interrupted in midstream. A variation can both remove the complex data structure requirement and solve the need for large stacks by using two bits for color rather than just one. These two mark bits will contain three possible values: white, black, and gray. The marking algorithm (Figure 8-7) proceeds much as with the original marker, except that instead of stacking the information necessary to make the outer recursive call, it colors the cell containing the unprocessed field gray. This makes **mark** a highly tail-recursive procedure, and one that can be written as a tight loop of conventional assembly code.

At completion of a marking run, either because a terminal node is reached or an already-colored cell is encountered, the function **next-gray** scans through memory looking for the first gray cell, if any. If none is

```
mark(i, c) = "color all cells accessible from i as c"
             if mark-bits(i) = c "stop on marked cell"
             then next-gray(0,c) "and start scan from 0"
             elseif mark-bits(i) = gray
             then mark(cdr(color(i, c)), c)
             else let ix = color(i,c) in
                  if tag(i) = terminal
                  then next-gray(0,c)
                  else let iy = color(car(i), gray) in
                       mark(cdr(ix), c)

where next-gray(i, c) =
  if i>top-of-memory
  then c
  elseif mark-bits(i) = gray
  then mark(i, c)
  else next-gray(i + 1, c)
```

FIGURE 8-7
Multicolor marking.

found the marking process is done; if one is found, the marking process is restarted on its untraced field (the cdr in this case).

Most of the optimization tricks discussed above could be used to shorten total execution time.

Although of only marginal interest as a technique by itself, this idea of multiple colors will become important when used in conjunction with other memory management functions.

8.2.5 Parallel Mark-Sweep

(Dijkstra et al., 1976; Ben-ari, 1984)

All of the above mark-sweep variants suffer from a significant problem from the user's viewpoint. When storage is exhausted, the machine stops doing useful work for up to seconds while memory is scanned repeatedly. Although some of the techniques to be discussed later can spread this time out, it is still a significant fraction of a machine's computational resources. From a computer architect's viewpoint this represents a golden opportunity to introduce specialized hardware that buys back these lost resources, particularly hardware that runs in parallel with the computer performing the computations.

The most famous such algorithms assume a ***parallel garbage collector*** (called the ***collector***), which runs in tandem with the main computer (called the ***mutator***), trying to sweep no-longer-useful cells into a ***free list*** without interfering with the mutator's computation. The mutator independently removes cells from this free list as it needs them. These algorithms are also called ***on-the-fly collectors.***

The general approach assume three colors as in the prior section:

- White represents free or potentially free cells.
- Gray represents cells touched by the collector or mutator, but whose car or cdr has not been marked yet.
- Black represents cells that have been fully traced by the collector.

Note that black does not guarantee that the cell is in use, only that it has been traced.

Operations starts with all cells colored white, all known free cells on the free list, and the root cells (indicated by the machine registers) colored gray. As the mutator allocates new storage from the free list, it colors them gray also.

The collector, working in parallel, traces cells as described above. A gray cell is colored black, its children are colored gray, and the process is repeated. If the collector is fast enough, it eventually catches up with the mutator, and all cells in memory are either white or black. At that point the collector begins a sweep phase, linking white cells into the free list and recoloring the black cells white. After the sweep is complete, the root registers of the mutator are colored gray, and the marking phase of the collector is restarted.

The mutator continues during this sweep and recolor phase, again recoloring any allocated cells from white to gray. The collector will not sweep up gray cells.

If the mutator ever reaches the end of the free list, it stops until the collector catches up and sweeps at least the first new free cell into the free list.

Figure 8-8 diagrams a complete cycle of this activity.

Note that this process protects for one cycle cells that are allocated by the mutator after the collector samples the root registers but that become unused before the collector completes its mark. They are still colored gray initially and then traced and converted into black. Although protected from collection on this sweep, the next time through they will not be found connected to anything and will be collected. Because they represent garbage which does not get collected right away, they are often called ***slow garbage.*** Cells which are freed by the mutator while the collector is sweeping are identified fairly soon, and are thus ***quick garbage.***

Hickey and Cohen (1984) have identified three kinds of cycles, formulas which estimate when they might occur, and how much idle time the mutator might be forced to wait. These cycles include:

Stable cycles, where the mutator never has to wait

Alternating cycles, where the mutator must idle every other cycle

Critical cycles, where the mutator must wait every cycle

For a reasonable set of assumptions the cycles tend to be stable if the percent of average memory used by the mutator to the total available memory is less than about 50 percent. Above a 70 percent level the cycles become critical all the time. In between these levels the cycles tend to be alternating.

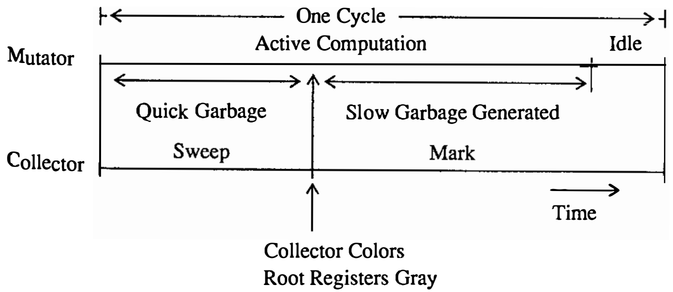


FIGURE 8-8
Parallel mark-sweep cycle.

8.3 REFERENCE COUNTS

All of the previous garbage collection methods require one or more scans of memory to find cells that are still in use. Even when done in parallel, they often result in the main computation being forced to the sidelines for potentially very long periods of time.

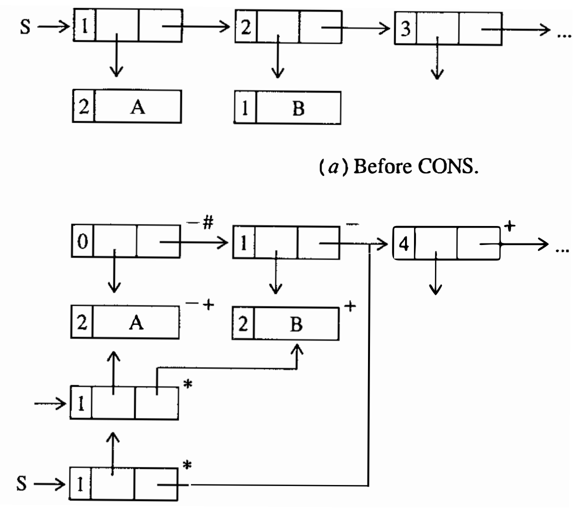
A garbage collection method that reduces this need for long dead periods would be of obvious benefit for many applications. This section describes a method that for the most part breaks the cost of garbage collection into small pieces, and spreads them out a little at a time across many operations. The basic approach is to augment the tag field of each cell with a multibit ***reference count*** field. This field maintains a count of the number of other cons cells which have pointers to this cell. It gives no information as to where they are, only how many there are. If this field is properly maintained, then any cell with a reference count of0 is a free cell.

There are two parts to maintaining this count: incrementing it when new cells are created, and decrementing it when cells are abandoned. Figure 8-9 diagrams the effects of this on the SECD instruction ***CONS***. Allocating the two new cells for the result requires setting their reference counts to 1. The cells containing the new car and cdr values have their reference counts incremented, as do the **cddr** cell off the S list (the new result will point to it). Finally, the first cell on the S list has its count decremented because the S register will not point to it after the operation. Figure 8-10 diagrams a sample case indicating which cells have their counts incremented or decremented.

The function ***rplacrc*** is assumed to operate like **rplacd** and the like; it returns a copy of its first argument, but has the side effect of setting the reference count field of that cell to the second argument.

```
CONS: "in side effect code"
        let s1 = rplacrc(allocate(F), 1)
        and x = rplacrc(allocate(F), 1)
        and a = inc-rc(car(S))
        and d = inc-rc(cadr(S))
        and s4 = inc-rc(cddr(S))
        and s2 = dec-rc(S) in
             let result = rplacd(rplaca(x,a),d) in
                 S←rplacd(rplaca(s1,result),s4)
        whererec inc-rc(x) = rplacrc(x,rc-tag(x) + 1)
        and dec-rc(x) =
             let count = rc-tag(x) – 1 in
                 if (count = 0) and (tag(x) = cons)
                 then let a = dec-rc(car(x))
                        and d = dec-rc(cdr(x)) in rplacrd(x,count)
                 else rplacrd(x,count)
```

FIGURE 8-9
Reference count maintenance in the SECD Machine.



(*a*) Before CONS.

*new cell, count set to 1
−count is decremented
+count is incremented
#cell becomes garbage

(*b*) After CONS.

FIGURE 8-10
Sample reference count modification.

8.3.1 Decrementing the Count

The interesting part of Figure 8-9 is the definition of the function to decrement the reference count of a cell, **dec-rc.** After reduction by one, if the resulting count is 0, then this cell is garbage and any pointers it has to other cells are no longer relevant. This means that if this cell is a nonterminal cons type, then both cells referenced by its car and cdr fields should also have their reference counts decremented. Recursively, if those counts go to 0, the process should be repeated on their children, and so on. If all the cells in a list have reference counts of 1, decrementing the count of the first cell will cause the reference counts of all of them to be decremented.

In Figure 8-10, the car cell **A** in the original S list is incremented because it is referenced by the newly created cons cell. It is also decremented because the car of the original S list is decremented, and its count went to 0. Likewise, the second cell has its count decremented because of the ripple out of the first cell's 0 count.

Finally, because of this ripple effect, it is very important that all the increments in Figure 8-9 be done *before* the decrements. If not, it is possible that a cell that went to 0 could cause an avalanche of decrements down the line, even though the very next step will be to increment its count back up. Without care, this could cause improper labeling of cells as garbage, or at least a waste of computation time to handle recursively decrementation and reincrementation of counts.

8.3.2 Sweeping

Given reference counts, free storage can be recovered in one of two ways. First, a sweep of memory can be made as before, except that we collect cells that have a 0 count rather that a certain color markbit. Again, this means that the typical program execution comes to a halt when the sweep is in progress.

The second approach integrates the sweep into the **dec-rc** function. Whenever a cell's count reaches 0, it is immediately known to be garbage, and it can be collected as soon as any internal pointers are read. To be useful, this means linking it into the free list.

This latter approach has several performance advantages. First, a separate sweep through memory is eliminated and replaced by small amounts of very local and incremental collection. This is particularly valuable to modern computer systems that have large virtual address spaces and caches to speed up individual accesses. All the accesses to a particular cell are done within the space of a few instruction times, minimizing cache misses and page faults. Additionally, the collection occurs in small segments rather than large chunks, making it more acceptable to the typical user. Finally, the total time spent sweeping tends to be less than for marking approaches, because the only cells that are touched are those that are real candidates for collection. The vast majority of cells involved with relatively long-lived data structures are never touched.

8.3.3 Implementation Problems

Reference counts are not without their problems, some of which are serious. These include:

1. The inability to guarantee a constant, or even bounded, amount of time for each operation that modifies a reference count
2. The potentially large number of bits required to hold a reference count
3. The possibility of loops causing instabilities in the algorithm

The first problem comes about because of the frequent need to decrement reference counts for even the simplest of machine operations, as was demonstrated for the **CONS** instruction. Each such decrement can potentially cascade all through memory in an unpredictable manner.

The number of bits needed to maintain a full reference count can also be a problem. Many modern ISAs can support up to 32 bits of address space, which if all chained together in a pathological chain can result in roughly 2**32 references to a single cell. This requires a reference count field in each cell of 32 bits.

Finally, anytime an operation such as **rplaca** or **rplacd** is used, it opens up the possibility of a circular set of pointers in a list. For constructions such as the environments for recursive functions, this is in fact what is intended. The problem is with the reference count for the first cell that is fully in the loop. It has a reference count of at least 2, one for the cells in front of it in the list and one for the pointer that completes the loop. A **dec-rc** on the front end of the list may reduce those counts to 0, freeing them, but the loop-completing reference will prevent the first cell in the loop, and thus the rest of the cells in the loop, from ever going to zero. Without care we will quickly end up with unused loops of cells that the reference counting will never recover. This is "lost storage."

While there are no "good" solutions to the first problem, there are some partial remedies. For example, one could limit the depth of a recursive decrement to some fixed number, such as five. Then, if there are still more cells left to be decremented, their addresses could be stacked or handled as in the marking process. Then if other calls to **dec-rc** result in fewer than five cell count decrements, the rest could be picked up from this stack of uncompleted ones. While this results in relatively constant time per call, it obviously requires very careful analysis to guarantee proper operation under all conditions.

The problem with size of reference count fields is typically handled by limiting the field size to, say, two bits. These two bits would encode four possible reference count values, 0, 1, 2, or ≥2. This reflects the fact that in real programs the vast majority of reference counts are 1 or 2. Data from Clark and Green (1977) indicates that as little as 2.5 percent of all cells have a count of more than 1.

The changes in the incrementing and decrementing functions are direct. As long as the count is 2 or less, the two bits encode things perfectly. However, as soon as the count goes over 2, the encoded value "sticks" to ≥2, and will never decrease below it, regardless of how many calls to decrement it are made.

While this guarantees that useful data will never be lost, it also opens up more cases like the circular lists where true garbage cannot be recovered.

The typical answer to these two cases of unrecoverable garbage is to ignore it until the program really does run out of available storage, including that which can be retrieved by the reference count method. When this occurs (at intervals much father apart than the prior approaches), the action taken should be to default to a more classical mark-sweep algorithm which will identify and recover all garbage in memory, regardless of its apparent state.

8.4 GARBAGE COMPACTION

(Baker, 1978; Hickey and Cohen, 1984; Lieberman and Hewitt, 1983)

An interesting aspect of linked lists in a conventional memory structure is that the actual locations that the cells occupy, and thus the pointer values used to link them, are immaterial as long as the overall structure is maintained. Despite this, however, all the memory recovery techniques discussed so far have gone to great pains to leave useful data exactly where it was found.

Many modern systems employing dynamic memory management use a totally different approach. Instead of skipping around cells that are still in use, these techniques literally move still useful cells to another area of memory. When all cells still in use in an area have been moved out, all cells in the original area are free, without the need to scan or sweep them. Usually the area of memory to which cells are moved is a contiguous block.

Such algorithms are often called ***compacting garbage collectors*** because they compact the useful information out of a large area of memory into a smaller, more densely occupied one. The original memory is then ready to be allocated as free space, often using the simplistic sequential allocation scheme with which we began this chapter.

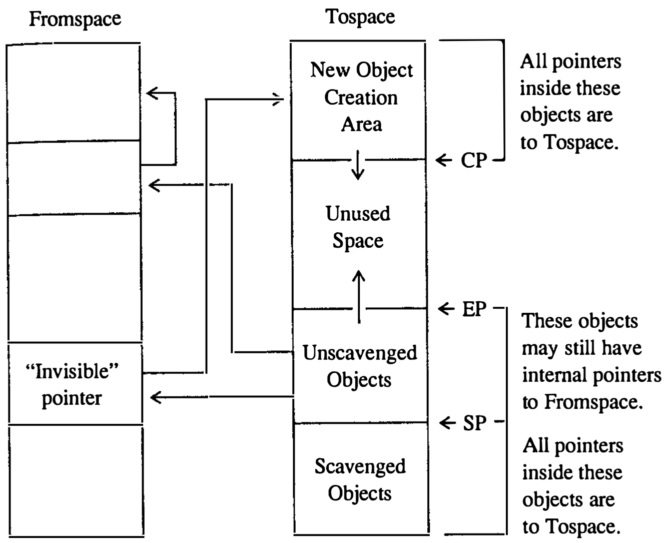
The advantages of such an approach are significant. All that is left is a modification of the marking process. Memory in its entirety need never be scanned. The allocation process becomes trivial. Further, the work involved with what is left of the marker can be divided into small pieces that can be spread in a uniform manner over many operations. Next, variations of the general approach are excellent matches to paged and virtual memory systems. Finally, they are ideal candidates for implementation as either on-the-fly or parallel systems as defined in the last section. The following subsections address the most famous of these algorithms, Baker's algorithm.

8.4.1 Basic Concepts

The basic approach in ***Baker's algorithm,*** and its modern descendants, is to divide memory up into roughly-equal sized pieces. For the present we will assume just two such pieces, both equal, and each called ***hemispaces,*** as pictured in Figure 8-11. One of these halves, called the ***fromspace***, represents the storage from which all useful data should be removed. The other half, the ***tospace,*** represents the area of memory that the program is currently using as a source of new cells, and which should receive all the useful data from fromspace.

The memory in fromspace is said to be ***condemned,*** and all objects still of use to the program are to be ***evacuated*** out. The process that looks through fromspace for useful data is called the ***scavenger,*** and the act of actually verifying that a cell is fully moved is called ***scavenging*** it.

When all useful data has been evacuated out of fromspace, that entire half of memory is now free and can be reused. This involves ***flipping*** the areas called fromspace and tospace, condemning the old tospace (the new fromspace), and starting over the process of evacuating all useful data from it. Needless to say, this does constrain a program never to use more than half of the total available memory, or flipping will not be possible.


Fromspace
Tospace
New Object Creation Area
Unused Space
Unscavenged Objects
Scavenged Objects
"Invisible" pointer
← CP
← EP
← SP
All pointers inside these objects are to Tospace.
These objects may still have internal pointers to Fromspace.
All pointers inside these objects are to Tospace.

CP: Creation pointer (= F Register in SECD machine).
EP: Evacuation pointer.
SP: Scavenger pointer.

FIGURE 8-11
Major memory allocations in Baker's algorithm.

Tospace is where all new cells are allocated and all cells evacuated from fromspace are placed. Within it are two main subregions. Growing from one area of memory is space for all new cells created to support the executing program. A pointer called the ***CP*** (for ***creation pointer***) indicates the next available cell in this region, and it is incremented each time a cell is allocated by the program. This is exactly equivalent to the F register in the original SECD Machine.

Growing from the opposite end of memory is the ***evacuation area.*** The ***EP*** (for ***evacuation pointer***) indicates the next available cell in it. Whenever a cell is evacuated from fromspace, its contents are copied unmodified to the cell indicated by EP, and EP is bumped.

Evacuating a cell means moving an exact copy of it from fromspace. This does no good if all other cells that point to it (through pointers in their car or cdr) do not know where it has been moved to. This is solved by leaving behind in the cell whose contents are copied a "forwarding address," which consists of a special tag and a value equaling the address in tospace. This tag, often called an ***invisible pointer*** or ***reference pointer,*** is equivalent to an ***indirect address*** on a conventional machine. Any reference to such a cell in fromspace is invisibly forwarded to the indicated cell in tospace.

Finally, just copying a cell from fromspace to tospace does not guarantee that it will be fully compatible with tospace. If it is a nonterminal, its car or cdr field might still point to a location in fromspace.

The scavenger function will look at the pointers, evacuate the indicated cells if necessary, and update the fields in the original cell. In Figure 8-11 the *SP* (for *scavenger pointer*) represents the point in the evacuated area of tospace below which all cells have had their pointers so adjusted. Above this point, the cells have been evacuated from fromspace but have not had their fields checked.

8.4.2 Basic Support Procedures

The process of managing the evacuation process involves three major procedures: **evacuate, scavenge,** and **flip,** as pictured in Figure 8-12. Because of the need to manipulate specific memory cells, none of these programs are pure functions; many of the statements are more conventional assignments.

The process is initiated when, for one reason or another, the program decides to flip tospace and fromspace. One might think that this occurs when there is no more space from which to allocate free cells, as when EP = CP, but this is usually too late, particularly if EP ≠ SP. In this latter case there may still be unevacuated cells in fromspace referenced by unscavenged cells in tospace, but there is no room to move them to tospace.

The more appropriate time to initiate a flip is as soon as EP = SP. At this point we are guaranteed that all useful cells are in tospace, and that all their internal pointers are also to tospace. The **flip** procedure then resets CP to point to the top of the old fromspace, and resets EP and SP to point to its other edge. **Flip** then takes the roots of all lists visible to the program, (namely, those accessible from the machine's main registers, such as S, E, C, and D), and evacuates a copy to the new tospace.

The **evacuate** procedure looks at the address of the object to be evacuated. If it is already in tospace, no copying is necessary. If not, the tag of the cell it addresses is checked. If it is an invisible pointer, the evacuation procedure is repeated on the forwarding pointer value. If not, it is copied to memory (EP), the location in fromspace is replaced by an invisible pointer to EP, and EP is bumped.

The final basic procedure is **scavenge.** This function will be called at regular intervals by other operations and manages the completion of the processing of one element in the evacuation area. If there are no more objects to be scavenged, **flip** is called and the process repeated. If there are objects, the next one (from SP) is checked. If it is a nonterminal, each field of it that contains a pointer is read out, that referenced cell is evacuated (adding to EP), and the original field in the nonterminal is adjusted to point to the new copy.

8.4.3 Integration into System Functions

Figure 8-13 diagrams how the memory management functions of Figure 8-12 can be utilized by some typical SECD instructions. An auxiliary function, ***create-cons***, creates a new cons cell in tospace where both of its pointers are also to cells in tospace. This latter assertion is guaranteed by using **evacuate** on each argument and then storing the resulting pointer in a new cell from CP. In addition, a fixed number of **scavenge** cycles moves SP up toward EP. This should be a small number to minimize overhead execution time but should be equal to at least 2 so that SP at least keeps up with any additions to EP due to the evacuates.

```
procedure flip; Called when SP=EP
  ;Assume processor registers R1...Rn hold pointers to
  ;  all data structures still of interest to mutator.
  reverse Fromspace and Tospace;
  CP←- top of new Tospace;
  EP←SP←-bottom of new Tospace;
  for i=1 to n do Ri←evacuate(Ri); Initialize Evacuation area

evacuate(x) =
  ; Assume x points to some memory location whose
  ; contents should be moved to Tospace.
  ; Return pointer to new location in Tospace where
  ; new copy of object now resides (unscavenged area).
  ; The scavenger will move subcomponents pointed to by
  ; this object later.
  ; Leave a forwarding pointer behind in Fromspace
  if x points to Tospace
  then x
  else let z=memory(x) in; Look at the object
       if z="invisible pointer"
       then evacuate(z)
       else begin ; A sequence of assignments
            memory(EP)←z; Now copy object to Evacuation area
            memory(x)←"invisible pointer" to EP;
            temp←EP;
            EP←EP+1;
            return temp;
            end

procedure scavenge; Process one object in Evacuation area.
  if EP=SP ; See if done
  then flip
  else "evacuate cells referenced by memory(SP)" begin
    car(SP)←evacuate(car(SP));
    cdr(SP)←evacuate(cdr(SP));
    SP←SP+1;
    end
```

FIGURE 8-12
Major system routines for Baker's algorithm.

If CP equals EP at the beginning of a **create-cons,** there is no more obviously free space in tospace. If SP is not equal to EP at this point, then there may also still be cells in fromspace that need to be evacuated.

```
create-cons(a, d) =
  ; Create a new cons cell.
  ; Return pointer to Tospace creation area.
  ; Routine guarantees that all pointer objects are to Tospace.
  ; Call scavenge a few times in the process.
  if CP=EF
  then "storage may be exceeded"
  else begin
       CP←CP−1
       tag(CP)←"cons"
       car(CP)←evacuate(a)
       cdr(CP)←evacuate(d)
       for i=1 to #scavenge do scavenge
       return CP

do-car(x) =
  ; Take the car of x, making sure that it is in Tospace
  ; and that the car field of x is adjusted also.
  car(rplaca(x, evacuate(car(x))))

do-cdr(x) = ... similar ..

CONS: "the SECD instruction"
  S←create-cons(create-cons(do-car(S), do-cadr(S)), do-cddr(S))

CAR: "the SECD instruction"
  S←create-cons(do-car(do-car(S)), do-cdr(S))
```

FIGURE 8-13
Integration of garbage compaction.

The net result is either a need to invoke a conventional garbage collector in hopes of clearing enough space for these other cells, or simply an error condition from which a program is not allowed to continue.

The other auxiliary function, ***do-car,*** performs a simple **car** on its argument but has the side effect of guaranteeing that the pointer returned is to a cell in tospace, and that the car field of the original argument is adjusted to match.

The implementation of an SECD CONS instruction involves calls to both functions. Applying **do-car** and the equivalent to the stack S retrieves pointers to the two arguments to be **cons**ed. After these calls these arguments are guaranteed to be in tospace. Two calls to **create-cons** then combine the two arguments and link them back into the appropriate cells of the S list.

The implementation of the CAR instruction is similar.

8.4.4 Region Collectors

Baker's algorithm represents a great step forward, but it still leaves two major problems:

1. There is no distinction between very long-lived data (such as at the root of the environment) and very short-lived data at the top of S. Both are copied at each flip.
2. The nice performance characteristics fall apart when actual storage in use nears or exceeds one-half of the total available storage.

Both problems can be addressed by dividing memory into smaller ***regions*** that can be scavenged independently of other units, and then controlling when this scavenging occurs. For many modern computers supporting virtual memory these regions most naturally match up to a small number of memory ***pages*** of about 1K to 4K bytes each.

The general approach is to keep several regions: a pool of totally free pages, a region from which new cells are to be created (the ***creation region***), a region to which cells are evaluated (the ***evacuation region***), and all other regions which may contain useful data. When a creation or evacuation region fills up, a new one is started up from the free pool. The filled one becomes just another region with potentially useful data in it.

Garbage collection begins when a region (any region) is condemned. This announces our intention to remove from it all useful data and then return it to the free pool. Evacuating an object from a region works as before; a copy is stored in the evacuation region and an invisible pointer is left behind in the original cell.

The one problem with this is determining which cells in the region should be evacuated. Tracing through all of memory connected to the register roots seems no better than the original Baker algorithm. A better approach is to tag each region with a ***generation number,*** which is incremented each time a new creation region is started. Thus a region with a higher generation number was created later than one with a lower number. Under most circumstances cells in one region will point only to other cells in either the same region or earlier ones (ones with lower generation numbers). A trace through a list connected to a register can thus stop when all the pointers at the edges of the part explored so far are to regions with generation numbers less than that being condemned. During this trace, all objects in the condemned region can be evacuated, and when it is over there is nothing left of interest in the region. After the lists linked to all machine registers have been so scanned, and their copied values scavenged, the region is empty.

The circumstances when this is not possible come primarily from operations like **rplaca,** which can form loops, or join anything in one generation to anything in any other, even later ones. This will be addressed later.

The process of tracing through the most recent generations does not require a fancy marker or the equivalent. A slightly modified scavenger that works up by regions, starting with the one right after the condemned one, will automatically evacuate the useful cells and adjust the pointers in the scavenged region to reference the evacuation region. When scavenging has completed the most recent region, the condemned region can be released and a new region condemned.

The same evacuation region can be used as a target for as many condemned regions as will fit.

8.4.5 Forward Pointers

The one problem with the above algorithm is that in real programs forward references from old regions to new ones may very well exist, and if those pointers are not updated, then after freeing a condemned region such older regions may be left with a ***dangling pointer.*** Such forward references could come from either explicit programmer calls to functions such as **rplaca** or **rplacd,** or from recursive environments.

The most common solution to this is to append to each region an ***entry table*** which contains an entry for each forward reference from an earlier region to this one (see Figure 8-14). This entry includes an invisible pointer to the appropriate cell in the region, and a backward pointer to the cell in the earlier region making the forward reference. The actual forward pointer in this latter cell is to the entry table entry, not the desired cell. The invisible pointer automatically indirects any desired reference to the appropriate cell.

This entry table must also be scavenged before a condemned region can be released. Scavenging in this case means evacuating any indicated cells in the region and updating the original pointers in the earlier region to point to a new entry table for the evacuation region.

8.4.6 Lifetime-Based Scavenging

Scavenging need not be done on just one region at a time. We could condemn a whole set of regions, such as from some region k up to the most current region (or whatever it is when the scavenger gets there). In fact, scavenging all generations essentially recreates Baker's original algorithm. However, as was mentioned earlier, not all data is useful to a program for the same period of time. Much of the data in newer generations tends to be transient, and thus should be scavenged more frequently. Data that is still left from older generations is more permanent, and thus should be scavenged less frequently.

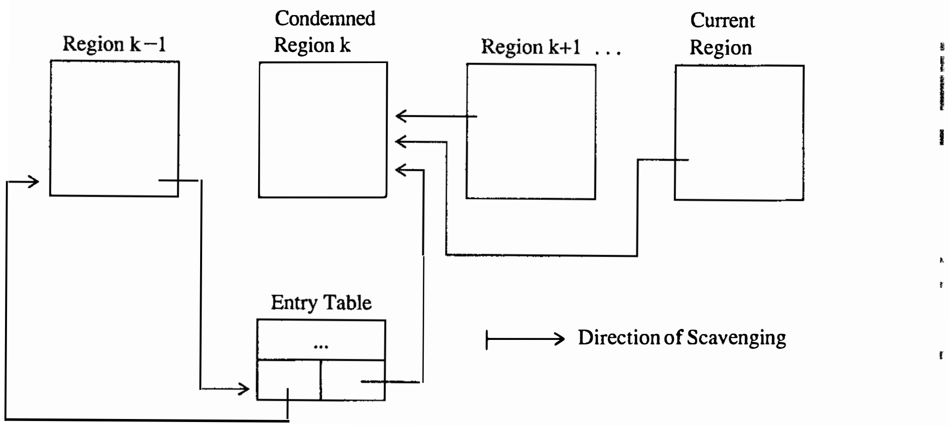


FIGURE 8-14
Entry tables for forward references.

Given this, it makes sense to send waves of scavenging down from the most recent generation that stop at some intermediate points, and only infrequently extend down to the earliest generations. This is sometimes called ***ephemeral garbage collection.***

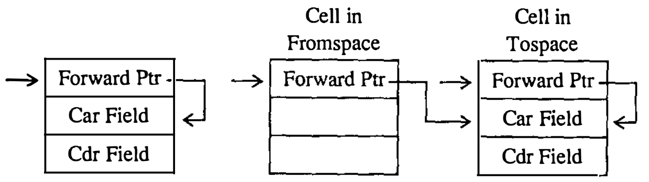
Another approach that is often used provides the programmer with different regions from which to allocate new cells. The difference is in how frequently, if at all, scavenging will be performed on them. Some regions, for example, might be used to contain data that the programmer (or smart compiler) knows to be long-lasting, such as global data, and that never needs to be scavenged, except perhaps under explicit program initiation. Other regions might be associated with specific modules or group of programs, and can be totally reused without scavenging when the modules are completed. While all these complicate the programmer's task, they can buy considerable increases in the number of machine cycles available to the actual computations to be done by the program.

8.4.7 Continuous Indirection

Although the above variations of Baker's algorithm all prevent a system from "going to sleep" at inopportune moments to mark and sweep memory, they still consume machine cycles, a little at a time when any of a variety of operations is performed. For example, each time an object is referenced, the contents of the memory cell at the pointer's specified address must be checked for the possibility of a forwarding pointer. When implemented on conventional machines without special tag-checking hardware, this often requires special shifting, masking, and testing.

One approach to avoiding this activity (Brooks, 1984) is to include in each cell an extra field that always holds a forward reference for this object (see Figure 8-15). All pointers to an object point to this field, with its contents pointing in turn to the actual cell value, wherever they are. Initially, when a cell is allocated, this forwarding pointer is set to point to the data fields of the cell. When the cell's value is evacuated to tospace, the forwarding pointer is adjusted to point to the new cell's value field.

Now, whenever an object is referenced, an extra indirection through this additional field will always find the appropriate value, wherever it is, without any extra tag checking. On some machines, such as the Motorola 68000, this can reduce machine cycles by as much as 2.5 times over a straight Baker implementation.



(*a*) At allocation. (*b*) After evacuation.

FIGURE 8-15
Trading space for tag checking.

8.5 ALTERNATIVE LIST REPRESENTATIONS

Ever since their invention, s-expressions have been a magnet for clever implementations other than the simple tag-car-cdr representation used to this point. This section presents a cross section of these ideas, many of which have been used in at least one real system. These ideas fall into three categories:

- Methods to compact the amount of storage needed to represent a dotted s-expression, primarily by collapsing unnecessary pointers
- Techniques for implementing tag bits
- Approaches to list representation that avoid pointers altogether.

The following subsections address each of these briefly.

8.5.1 List Compaction

The basic cons cell has room for two pointers, the car and the cdr. In many cases the number of bits taken up by these pointers exceeds the number of bits of information pointed to. This was recognized early on, and a pair of techniques called *car coding* and *cdr coding* have been developed to address them.

Car coding comes from the observation that the majority of cons cells have car fields which point to terminal cells holding constants. For many modern machines with large address spaces, the number of bits for these addresses is equal to, or greater than, the number needed for the constants. See, for example, SECD programs. If the space allocated to the constant is the same size as that for the cons cell, then we are wasting the equivalent of an entire cell. Significant savings are thus possible if the constant value replaces the car pointer in the cons cell. This in turn requires more tag bits to permit the machine to distinguish a car field which is a pointer from one that is a constant.

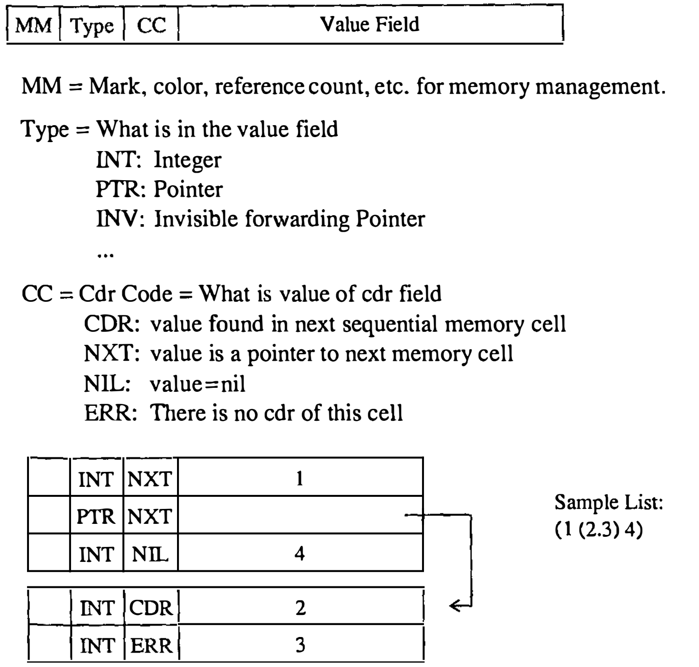
Similar savings, but for different reasons, are possible with the cdr field. Here, a very common occurrence is for the cdr to be simply a pointer to the next word in memory. This happens frequently when lists are being built and some sort of a sequential allocation mechanism is used, such as the original SECD incrementing F register or the Baker model [cf. Clark and Green (1977), who reported that in some systems up to 98 percent of all cdr pointers fell in this class]. A second common occurrence is for a cdr field to point to nil. In either case, we do not need a lot of bits to represent these possibilities. Ideally, a two-bit field is sufficient to indicate if the cdr field is actually present, and if not, whether its unwritten value is nil or the address of the next cell.

Both of these ideas can be combined as shown in Figure 8-16. Each cell has an expanded tag field and just enough value field to hold either a single pointer or a typical constant. The tag field now has three parts:

- Memory management bits
- Indicators of the type of value stored in the value field
- A representation of what the cdr of this field would look like if it were a cons cell

There are now no distinct value bits which are always car or always cdr or, in any case, always pointer.

Such a cell representation can save a very considerable amount of storage—up to a four-to-one reduction if all cars are to small constants and all cdrs point to next available cells. The problem, however, is that



MM = Mark, color, reference count, etc. for memory management.

Type = What is in the value field
INT: Integer
PTR: Pointer
INV: Invisible forwarding Pointer
...

CC = Cdr Code = What is value of cdr field
CDR: value found in next sequential memory cell
NXT: value is a pointer to next memory cell
NIL: value=nil
ERR: There is no cdr of this cell

FIGURE 8-16
Compacted memory cell representation.

the formerly simple functions such as **car, cdr,** and **cons** now become relatively complex, and when integrated into instructions like CONS require special handling if efficiency is to be achieved. To demonstrate this, Figure 8-17 lists what the internal machine operations would look like to implement **car** and **cdr.** These functions are called **do-car** and **do-cdr** in the figure, and rely on an auxiliary function **get-value** to interpret the value of a cell and return a dotted pair of the form (⟨**tag**⟩⟨**value**⟩).

8.5.2 Tag Implementation Techniques

Up to this point we have been assuming that tag bits (however many there are) are stored in bit positions contiguous with value field bits. This is fine when one is starting with a clean sheet of paper and no compatibility problems, but it often presents real challenges when one is trying to implement a cell memory structure in the memory of a computer that has no hardware support for them. Blindly allocating bits in a memory word for the tag may mean that a great deal of bit-level masking and shifting may be necessary every time a simple operation is to be performed. It may also limit the available address space and greatly complicate the encoding of constants.

On some machines, certain bit positions are better in this regard than others. For example, many machines have address spaces which are less than the width of a standard register or memory word. The upper bits in such cases are good candidates for tags, particularly if they are ignored by the address logic in the machine. As another example, many computers support addresses that are byte-level but include instructions that fetch a word of data, where a word must start on a byte address which has one, two, or three low-level 0s. Such lower bits are also good places for at least part of the tag bits, particularly if there are word load instructions which ignore them.

```
get-value(x) = "get value from cell x"
               if type(x) = "INT"
               then (INT.value(x))
               elseif type(x) = "PTR"
               then (PTR.value(x))
               elseif type(x) = "INV"
               then get-value(value(x))
               else...

do-car(x) = "get car of cell starting at x"
            if cdrtag(x) = "ERR"
            then error "this is not a cons cell"
            else get-value(x)

do-cdr(x) = "get cdr of cell starting at x"
            if cdrtag(x) = "ERR"
            then error "this is not a cons cell"
            elseif cdrtag(x) = "CDR"
            then get-value(1 + x)
            elseif cdrtag(x) = "NXT"
            then (PTR.1 + x)
            else (PTR.nil)
```

FIGURE 8-17
Machine functions for compact lists.

In either case, this may still cause problems with the packing and unpacking of constants, particularly integers and floating-point numbers. We lose bits of accuracy in such numbers, plus introduce shifting and unshifting operations, and increase greatly the complexity of determining when arithmetic operations overflow the available bit positions. To avoid this, we may want to place the tags elsewhere than in a word with data in it. On byte-oriented machines a extra byte next to a full word of data could be allocated for tags, but that also causes performance problems because it now requires multiple memory accesses, often to full words which are not aligned optimally with the physical memory organization of the machine.

It is also possible to place the tags in a totally separate area of memory, such as an array of bytes separate from the array holding data values. This still requires two memory accesses, one for tag and one for value field, but with care this latter one is always aligned properly with memory. The complexity is the need to compute two addresses, one for tags and one for values.

A final approach does away with data tags altogether by allocating different types of objects in different areas of memory. For example, the first 1 Mbyte of address space may contain only terminal cells that are integers, the next 1 Mbyte might contain only floating-point numbers, and the rest of memory might contain double-word-wide cons cells, where each word is a pointer. Because of the obvious fit to pages in virtual memory systems, this approach is often called ***BIBOP,*** for "BIg Bag Of Pages."

In this system there is no packing and unpacking of bits necessary for data fetching, because determining the type of a cell now requires only tests on its address to see which region it is in. While with care this can be quite fast, it does raise havoc with memory management, since now there is not one, but many areas of memory to mark, sweep, evacuate, scavenge, etc., each with different characteristics.

Often a mixture of approaches is appropriate. For example, mark or color bits can often be stored in bit arrays separate from the value and the rest of the tags, and another technique can be used for the rest of the tags. This may actually enhance some garbage collection algorithms, since now one memory fetch can gain access to many such bits in one swoop.

There is some evidence in the literature that the overall effect of such design decisions might not be as dramatic as one might guess from the above discussion. Steenkiste and Hennessy (1987), for example, give some results that indicate that only somewhere between a 22 and 32 percent penalty is paid for certain kinds of ***RISC machines*** versus machines with direct hardware support for tags. The cost is due to the extra instructions required to mask and shift out the tag bits of an object, and branch on their values. Even lower numbers can be achieved by compile time checking and the generation of specialized code sequences when objects are known to be of certain types.

8.5.3 Pointer Avoidance

(Sohi et al., 1985)

All the methods discussed so far implicitly assume that the only way to implement s-expressions is with pointers. This is not so. As long as the structure and order of the terminal nodes is maintained, the actual way in which they are chained together is immaterial. This section gives several interesting approaches based on explicit labeling of the terminals of an s-expression. This labeling results from assigning a number to each node of an s-expression when drawn in dot notation as follows:

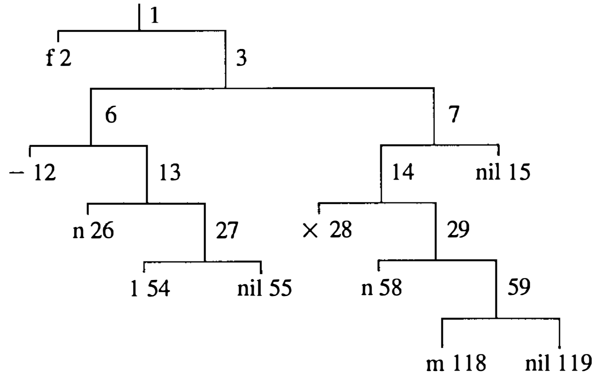
- The root node of an s-expression is assigned the number 1.
- For each nonterminal (cons node), if its label is k:

  The label for the node's car subnode is 2×k;
  The label for the node's cdr subnode is 2×k+1.

Figure 8-18 diagrams an s-expression and a labeling of all nodes, both internal and terminal. The labels that are important are the numbers for the terminals.

Now consider what would happen if we had an array of 119 entries,

(f (− n 1) (× n m)) = (f.((−.(n.(1.nil))).((×.(n.(m.nil))).nil)))



Value	Label	Binary
f	2	10
−	12	1100
nil	15	1111
n	26	11010
×	28	11100
1	54	110110
nil	55	110111
n	58	111010
m	118	1110110
nil	119	1110111

FIGURE 8-18
Labeling an s-expression.

and stored each terminal in the array location indexed by its label. All other entries are loaded with some special constant. If there is some way to compute an index for a particular terminal from any sequence of **cars** and **cdrs**, then we could access this array with one memory read, and never wend through a trail of pointers.

Without proof, the following is an algorithm that takes any composition of **cars** and **cdrs**, as in **c{ad}$^+$r**, and computes the binary form of such an index as follows:

1. Start with a leading 1.
2. For the sequence of **a**s and **d**s in **c{ad}+r**, create a sequence of 1's and 0's where each **a** is translated to a 0, and **d** to 1.
3. Reverse this sequence.
4. Append this sequence to the leading 1.

For example, the function **car** translates into the binary number 10, or index 2. As another, **cadaddr** should return the element n, which has an index of 58. The above algorithm reverses the binary sequence 01011 and appends it to a 1 to yield 111010=58 as desired.

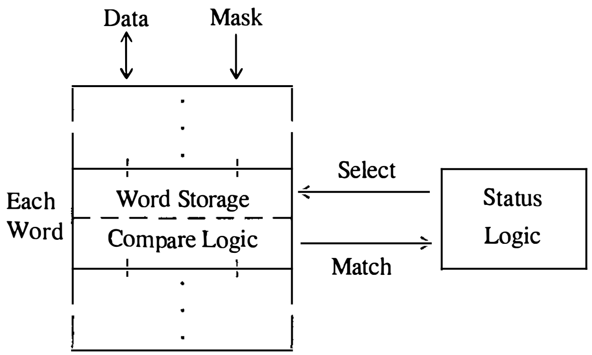
With this conversion algorithm available to a compiler, any access to a list can be converted into an index and then used by runtime code for very fast access.

Instead of an array that is largely empty, an alternative way to condense the same information is through an ***exception table*** that contains the pairs of indices and terminal values. Finding a particular terminal entry is now a process of scanning the exception table for the entry with the same index.

While it is slower than the sparse array approach, this approach can still be quite fast, particularly if the table is sorted by index. A standard binary search can locate any index in $\log_2(N)$ searches, where N is the number of entries. For many s-expressions which are flat lists, finding an arbitrary element by pointer chasing can take N/2 on the average.

ASSOCIATIVE MEMORY. Finally, keeping the table in an ***associative memory*** can accelerate such accesses to a uniform single machine cycle. An associative memory is one in which a data word can be presented to an array of memory words, each of which has its own comparator logic. With this logic each word signals over a separate ***match line*** whether or not it matches the input data word (see Figure 8-19). Very often an additional input ***mask*** provides a list of the bit positions in each word that have to match and those positions that can be ignored (are "don't cares" as far as the comparator logic is concerned).

With an exception table stored in such a memory, any index that leads to a terminal node requires exactly two memory accesses. The mask input for the first selects only those bits associated with the index, and the data input consists of the desired index padded by anything. If



Basic Operations:

Match: Each word whose stored data equals Data input in all bits where Mask is 1, signal on Match Line.

Read: The first word whose Select Line is 1, places its stored data on output.

Write: All words whose Select is 1, copy Data into storage.

Status: Indicate how many Matches. (At least 0, 1, >1)

FIGURE 8-19
A basic associative memory.

any entry matches, the match signal can be fed back into the ***select line*** to read out the appropriate entry.

Other standard list operations can also benefit from such an implementation. **Member,** for example, can invert the mask so that the index field are all ***don't care***s, and the value field is the one driving the match. Given a match, one can read out the associated index and know where in the s-expression the terminal occurred.

Finally, unlike the array or exception table format, an associative memory implementation can also give easy insight into internal nodes. Assume that the indices for each terminal are stored in binary in associative memory array entries, but have all the leading 0s deleted and the number left justified. Each entry is thus stored as shown in the table of Figure 8-18, with 0s padded to the right. Thus in an 8-bit field the index 58=00111010 becomes 11101000.

Searching for a terminal node of a specific index is similar to the previous procedure. The data pattern used for the search consists of a left justified index, with a mask that consists of all 1s in the index field except 0s in the right to match the number of 0s deleted from the left of the index. Thus, for the index 58, the data value for the search is 11101000 and the mask is 11111100. As desired, the only entry in the memory that matches is the desired one, the one with the value n.

The unique characteristic of this method is that the associative memory can give rational responses to indices that correspond not only to terminal but also to internal nodes. Consider, for example, the internal node 14 in Figure 8-18. Constructing a search pattern for it would have the data input of 1100000 and a mask of 11110000. There are multiple entries that match this, namely, those with indices 11100 (28), 111010 (58), 1110110 (118), and 1110111 (119). These, however, make up the terminals for the sublist (× n m), which corresponds exactly to the list connected to node 14.

Thus, by searching on any index, one can determine in a fixed number of machine cycles if that node is a terminal, a nonterminal, or not in the list, and if it is a terminal, what its value is. If we can count the number of responses, we can also determine immediately how many terminals are in a sublist, without ever having to employ a relatively complex recursive pointer chaser.

Despite the potential attractiveness of such an architecture, none of the above approaches are without problems. Clearly, having multiple sublists in the same memory requires some way to tag each entry as to which list it belongs to. For lists that have very deeply nested substructures, the number of bits required to express their indices may exceed the available word width. Implementing **cons** may be difficult because all the indices of the lists being joined need to be adjusted. Multiple references to the same list and circular lists also present problems in determining an appropriate index value.

8.6 PROBLEMS

1. Implement in the assembly language of your favorite microprocessor a simple three-color mark-sweep garbage collector, where the free cells are placed in a free list. Make sure you specify what a cell looks like in memory, and where the mark bits are located. Assume that all cells are the same size, and that a cons type contains two explicit pointers. Estimate how many machine instructions would be executed to clean a 1024-cell memory when it is half full with nongarbage linked together in a single full binary tree.
2. Repeat Problem 1 using a compacted format like Figure 8-16.
3. Devise a backup garbage collection mark-sweep algorithm to use when a system with 2-bit reference count fields runs out of easily recoverable storage (i.e., there may be loops of cells or cells that are no longer used but that had reference counts which had grown to the 2 case).
4. Assume that we added a third tag type to the basic SECD memory model, namely, a cons cell where the car field holds a short integer. By how much does this reduce storage requirements for a program list. How is the CAR instruction affected? What mechanism would you use to create such a cell and **cons** it onto an existing list?
5. Consider a parallel mark-sweep garbage collector. Assume that the collector can mark one cell in 5 μs and sweep it up in 1 μs. Also, the mutator maintains

a total linked list that averages 300,000 cells, and makes a new cell allocation every 20 μs (essentially creating a garbage cell at the same rate). Given a 1 million cell memory:

a. Roughly what does the collection cycle look like (put seconds on Figure 8-8)?
b. What kind of cycle do you think it is?
c. How much slow garbage is created?
d. What would happen if there were only 500,000 cells of available memory?

6. What would happen to Figure 8-10 if the decrementation of the reference count for the first cell of the original S list was done first?
7. The definition of **dec-rc** in Figure 8-9 does not recognize the possibility that a car or cdr pointer in a nonterminal cell might be pointing to nil. Rewrite the function's definition to fix this problem.
8. Repeat Figure 8-9 for the SECD ADD instruction. Then show what would happen to Figure 8-10 if that instruction were executed.
9. Assuming a reference count-based system, define which cells get incremented and decremented during the SECD instruction RAP.
10. Redo the description of an SECD ADD, assuming that invisible pointers are possible for any cell.
11. The SECD Machine needs only a handful of instruction opcodes, yet each uses up a full memory cell for each instruction, plus the car field in the program list. Devise a new memory model that uses varying numbers of 8-bit memory words to implement a more efficient storage arrangement. You can invent new tags as necessary. Assume at most 32 opcodes.
12. Discuss for the Problem 11 how the basic list processing instructions would be affected by such a new memory model, and what if any change would be needed in the compiler.
13. Describe in detail the low-level machine operations that would be needed to efficiently implement **append** as a built-in when memory is organized as in Figure 8-16.
14. How many cells would be taken by both the conventional two pointers per cons and by Figure 8-16 to implement the list of Figure 8-18?
15. Using Figure 8-17, describe how the SECD instruction CONS would work, assuming that free cells are allocated by incrementing the F register.
16. What labeling (as defined in Figure 8-18) would be applied to the elements of an s-expression of the form (0 1 2 3 4 5...n)? Is there anything special about the binary representation of these labels?

CHAPTER 9

DEMAND-DRIVEN EVALUATION

Looking back at our discussions on interpreting lambda calculus-based languages, there were two approaches to reducing expressions: normal order and applicative order. Both have problems when used as the sole execution model for a language's semantics. Normal-order reduction is guaranteed to give a normal-order solution if one exists, but it does so in a manner that is essentially sequential with a high probability of duplicate evaluation of the same argument. Applicative order permits some opportunities for parallelism and reduces duplicate argument reductions, but at the cost of possibly getting into an infinite loop and of evaluating arguments whose values are never used. These latter points particularly complicate trying to write an interpreter that handles recursion properly.

Is there a better way? The answer is yes, by combining the best features of both applicative- and normal-order reduction into ***demand-driven reduction.*** Here we avoid reducing an argument before an application as in normal-order evaluation, but at the first need for the expression's value we evaluate it and by one mechanism or another essentially replace all occurrences of the expression by the value. All duplicate or unneeded computations are avoided, while also avoiding the unnecessary infinite-loop problem. Each argument expression is evaluated at most once, as in applicative order.

This process can be taken one step further. Consider what happens in a demand-driven system if, when an expression finally must be evaluated, we evaluate only enough to satisfy the immediate demand, and package up the remaining computation until more pieces are needed. At that time the computation is restarted, but again carried just far enough to satisfy the immediate demand. The amount of computation is again minimized. For obvious reasons this is often called ***lazy evaluation.***

The data structure that holds the packaged or encapsulated computation is a variation on the concept of a closure. It is often called a ***recipe, future,*** or ***promise,*** since it represents a description of how to compute an object at any time in the future, with the promise that the value returned at that point is absolutely the same as if it had been computed initially, i.e., at the time the expression was formed. The major differences from a closure lie in what environment is packed in it, and what happens when it is restarted.

A typical example of where such delayed evaluation is useful is in the computation of some s-expression list. None of the list need be computed until a part of it is needed, as signaled by the attempted application of **car** or **cdr** to it. At this point, we need only compute the first element, and leave the computation of the rest to be performed at another time. If properly designed, this object can be left as the **cdr** of the original list, where future functions that operate on the list will find it. If no one ever needs the data, the object is simply never evaluated.

As another extension of this idea, instead of saving just some nested subexpression, we can checkpoint literally the whole state of the computation to this point into an object called a ***continuation,*** which can then be passed around like any other object. Instead of representing a value, the continuation represents an uncompleted computation that can be reopened at some later time, and often provided with values that were simply not available at the time the continuation was built.

The uses of these techniques are multiple. They can avoid lengthy computations unless absolutely necessary, and then perform only the minimal amount needed. They permit very small and concise expressions that correspond to very large, or even infinite, data structures. When recursively related, they permit a notation that resembles a very convenient, easily specified, co-routine structure. This in turn helps solve many real-world problems, such as representing input/output, describing executives and schedulers that manage the execution of other computations on a single processor, and providing specialized escape paths to handle errors and other exceptional conditions. Finally, paradoxically, they provide extensive opportunities for ***eager beaver*** parallelism, where free processors hunt out unexpanded computations and expand them before the values are actually needed.

This chapter covers the above topics. First, we introduce lambda calculus-based mechanisms for encapsulating and restarting computations at will, and then we discuss their use in what are called delay and demand evaluations. This leads to a model of (self-)interpretation that implements lazy evaluation directly, and to the introduction of ***streams*** to handle potentially infinite data structures. Following this we introduce continuations and how they can be used to manage computations. As before, the SECD architecture serves as a base, with extensions to this architecture used to discuss how these features could be implemented.

Although the topic is rather advanced, the material covered here is critically important to understanding almost any modern functional language at more than a relatively simplistic "how to express an expression" level. Consequently, it is suggested that even the casual reader at least glance through this chapter, emphasizing the first and last sections, which discuss explicit delays and forces, and continuations. Most other readers should spend the time necessary to absorb the material in the chapter fully. As with the SECD Machine, Henderson (1980) and Burge (1975) are good general references. Henderson and Morris (1976) go into more mathematical detail.

9.1 EXPLICIT DELAYS AND FORCES

A ***closure*** was defined in Chapter 7 as an object which contains a function definition and the environment which bound any free variables in it at the time the closure was built. The data structures to be developed here encapsulate not a function but a complete expression, and do so *before* the expression is reduced. As before, this encapsulation is ***referentially transparent.*** It does not matter when or where it is unpacked; the value of the expression after unpacking and evaluation is exactly what it would have been if it had been evaluated before its encapsulation.

For practical purposes, the simplest form of the package is the same as a closure. While we will continue to call it a closure, in the literature it is also called a ***thunk,*** particularly when it is used to pass arguments in languages with ***call-by-need*** semantics. The major differences are in when it is built, what is done with it, and how it is unpacked and restarted. The following subsections discuss these points, with emphasis on putting a sound mathematical foundation on them and describing how they can be implemented on the SECD Machine.

9.1.1 Lambda Calculus Basis

To put closures for expressions on a firm mathematical foundation we need two mechanisms, one to form them and one to force their evaluation. As usual, we want both mechanisms to be describable as well-behaved functions in their own right (even if they are implemented in a more efficient fashion).

The first of these functions, to build an appropriate closure, will be called the ***delay*** function; the latter, the ***force*** function. Thus, when used in a program, **delay** will accept an unevaluated expression as an argument and will form a closure as an output. This closure contains the code representing the expression plus the environment needed to provide values for all free variables.

Applying **force** to the result of such a **delay** will unpack the closure and evaluate the code in the specified environment. It will always be the case that for any arbitrary expression **E,force(delay(E))** ⇒ **E.**

To describe **delay,** first consider the expression **(λ|E).** This is a function with no binding variables. The expression **E** is "enclosed" by the surrounding function until the function is "forced" to give up its value by an application. In this case the application's argument is an empty expression ( ), and the application of the identifierless function above to this expression will be the same as a normal lambda application, except that no substitutions need be made. (*Note:* The same effect can be gained by using a binding variable which does not appear free in **E** and applying the function to some arbitrary argument which is then essentially discarded.)

If **E** is an application **[(λ*x*|B)A]** which we want delayed, then **(λ|E)** represents its closure and **(λ|E)**( ) an application that evaluates the closure. Thus, the process of constructing such a closure and then evaluating it can be defined mathematically by two lambda functions:

$$\begin{array}{ll} \mathbf{delay}(x) = (\lambda|x) & [\text{or } \mathbf{delay} = (\lambda x|(\lambda x))] \\ \mathbf{force}(x) = x(\ ) & [\text{or } \mathbf{force} = (\lambda x|x(\ ))] \end{array}$$

The **delay** function takes the argument expression and embeds it inside a new lambda function where it is safe from evaluation. Since this embedded lambda function has no binding variable, there is no danger of a name clash.

The **force** function takes a function produced by **delay** as input and forces evaluation of the function's body.

As proof that these definitions obey the above constraints on **force** and **delay:**

$$\begin{aligned} \mathbf{force(delay(E))} &= \mathbf{force((\lambda \mathit{x}|(\lambda|\mathit{x}))E)} \Rightarrow \mathbf{force((\lambda|E))} = (\lambda x|x(\ ))(\lambda|\mathbf{E}) \Rightarrow \\ &\quad (\lambda|\mathbf{E})(\ ) \Rightarrow \mathbf{E} \end{aligned}$$

Note that both of these functions assume normal-order reduction rules; otherwise the expression to be delayed would be evaluated before **delay** ever received it. Thus, if we were to embed **delay** and **force** into the interpreters described earlier, we would want to define both as starting a new special form where the arguments are not evaluated before computing the result.

9.1.2 Implementation in the SECD Machine

When translated into s-expression form, **delay** and **force** might look like (delay⟨*expression*⟩) and (force⟨*expression*⟩). A compiler could treat these functor symbols as ***special forms*** and generate code in a variety of ways. For (delay **E**) it could generate:

(. . . LDF **E1** . . .)   where **E1=compile(E,** . . . ,(RTN)))

At execution time the LDF returns a closure to be passed on as a value to the rest of the expression. The code for this closure, denoted **E1**, is a list which when executed evaluates **E**, leaves the resulting value on the stack, and then returns to its caller, just as for a traditional function.

The major difference between this closure and the normal closure created for a function is that this closure represents an entire expression and needs no arguments for its evaluation. Thus **force**ing it must require something different from a conventional AP or the equivalent. One solution would be to generate a dummy set of arguments and use AP anyway. This is possible but inefficient.

A cleaner solution is to add a more specialized instruction to the SECD architecture. This instruction, **AP0 (APply 0 arguments)**, works just like AP but assumes no argument list on the stack. The register transitions look like:

AP0: (f.e').s e (AP0.c) d→nil e' f (s e c.d)

The object (f.e') on the top of the stack is a closure which represents a function that needs no arguments.

Implementing a **force** thus requires only that the appropriate closure (as computed by **force**'s argument) be left on top of the stack and that this AP0 then be executed. The result will be to unpack the closure just as with AP but with no augmentation of the environment. At completion, a standard RTN instruction in the closure's code will return the value to the caller.

As a simple example, compiling the expression:

let x=1 in **force(delay(**x+2**))**

generates the SECD code:

```
(LDC (1) LDF (LDF (LDC 2 LD (1.1) ADD RTN) AP0 RTN) AP STOP)
```

The interior "LDF (LDC. . . RTN)" represents the code generated from the **delay(**x+2) expression. When AP0 is executed, only the closure created by this is created, with no extra argument list. As with the prior discussions of the SECD Machine, it is certainly possible to optimize these sequences by variations of AP0.

9.2 PROGRAM EXAMPLES

The simplicity and value of the **delay/force** functions are best shown by some potential uses. The ones covered here represent a very standard way to implement data structures that can be arbitrarily large or even infinite. Explicit **delays** embed in lists calculations for data that are not yet needed. Matching **forces** expand these uncomputed list fragments when needed and only to the extent needed for the current demand.

9.2.1 Infinite Lists

Perhaps the most common example of the use of **delay** and **force** is to build essentially infinite data structures in s-expressions, particularly infinite lists. One typical format for such lists is:

cons("first element," "closure for the rest")

The car represents the first element of the list and is a valid value. The cdr is a closure created by a **delay** and represents the computations that would lead to the rest of the list. The expression delayed by this closure is usually one that when forced will return an expression of the form:

cons("second element," "closure for the rest")

With this implementation, the first element of a list is accessible via a simple application of **car** to the original s-expression. Getting the second element requires applying **car** to the result of forcing the **cdr** of this list. Further elements of the list can be obtained by repeating this **car(force(cdr(**...**))** process. Basically, any function to be applied to the list which is a string of **cars** and **cdrs** is replaced by one where each **cdr(**...**)** becomes **force(cdr(**...**))**.

As one example, Figure 9-1 defines an expression that computes a list of all integers starting at 17. A call to **integers** builds a **cons** cell whose car is the current argument value and whose cdr is a closure for a call to **integers** with the next integer. Note that, unlike our previous recursive functions, this one has no terminating basis test; we want the result to go on to infinity, but with the delay mechanism used to prevent its execution from going wild.

```
letrec integers(m) =cons(m,delay(integers(m+ 1))) in integers(17)
→ (17.[integers(m+ 1),((m.17))])                                        .

SECD code: (...
   DUM NIL
   LDF (LDF (NIL LDC 1 LD (1.1) ADD CONS LD (2.1) AP RTN) RTN)
         LD (1.1) CONS RTN)
   CONS
   LDF (LDC (17) LD (1.1) AP RTN)
   RAP ...)
```

FIGURE 9-1
An infinite expression.

9.2.2 Filtering Delayed Lists

As a more complete example, Figure 9-2 defines the function ***first***, which returns the first *k* elements of a list, where the list passed as an argument to **first** is assumed to be one with a closure embedded in its cdr. The closure is evaluated only as often as needed to build the desired list of *k* elements. Note that this function works for any list with such a structure, and needs absolutely no knowledge of the embedded expression in the closure.

Figure 9-3 gives a more complex example, where the function involved in the closure itself relies on delayed infinite lists, including ones computed by itself. It was chosen for its structure, not its obvious inefficiency.

The function ***primes*** returns a **cons** in the above form, where the car is the next larger integer from the argument to **primes** that is a prime number and the cdr is a closure for all remaining primes. Internally, **primes** compares its argument against 1 and 2, and returns appropriate lists for these cases. In the more general case, **primes** checks its argument for divisibility against every prime number less than one-half of its value. These prime numbers are in turn developed by a recursive call to **primes**, starting at 2.

9.3 RECIPES, PROMISES, FUTURES

A close look at the **primes** function of Figure 9-3 reveals that it calls itself recursively from scratch each time a new number is investigated. This is

```
first(k,x) = "first k elements of infinite list x"
         = if k=0 then nil
           else cons(car(x), first(k−1, force(cdr(x)))
                     ↑                  ↑
                 1st element      closure for 2nd...

Thus: first(3,integers(17))
—→ first(3,cons(17,delay(integers(2))))

—→ cons(17,first (2,force(delay(integers(18)))))
—→ cons(1,  first (2,cons(18,delay(integers(19)))))

—→ cons(17,cons(18,first(1,force(delay(integers(19))))))
—→ cons(17,cons(18,first(1,cons(19,delay(integers(20))))))

—→ cons(17,cons(18,cons(19,first(0,force(delay(integers(19)))))))
—→ cons(17,cons(18,cons(19,nil))) = (17 18 19)
```

FIGURE 9-2
Explicit processing of closures.

```
primes(x) produces cons(np, delay(primes(np + 1))
where np is the next integer prime ≥x

primes(x) = "list of all primes starting at x"
  if x<3
  then cons(x,delay(primes(x + 1)))
  elseif indivisible(x,primes(2)) ; is x divisible by any prime
  then cons(x,delay(primes(x + 2))) ; x is next prime
  else primes(x + 2) ; try next odd number
      whererec indivisible(x,p) = ;Is x indivisible by all of p?
        if car(p)>x/2
        then T ;x indivisible by no element in p
        elseif remainder(x,car(p)) = 0
        then F
        else indivisible(x,force(cdr(p)))
```

FIGURE 9-3
A prime number generator.

an obvious inefficiency. We would really like to compute the partial lists of primes only once, and keep computed values around for future use.

As a simpler example of this potential inefficiency, consider

let **p**(x)= . . . in let **f**(x)=**p**(x)+**p**(x) in **f**(100)

The expression **p**(100) is evaluated twice, and if **p** is defined in terms of, say, **primes,** the amount of redundant calculations is substantial.

One solution to this is to somehow change AP0 so that the first time it evaluates a closure, it replaces that closure by the evaluated value. The cost of evaluation is borne only once, with all future uses simply picking up the value.

Although this sounds simple, there are at least two subtle problems. First, where is the result of the evaluation to be placed so that all other references to the original closure will get the value instead? Second, how do we distinguish between a closure that still has to be evaluated and one that might result from evaluating a closure?

One solution to this is to augment our definition of what is returned by **delay** to look like

cons(⟨***boolean-flag***⟩, ⟨***closure/value***⟩)

where the flag is initialized by **delay** to F.

This composite object is variously called a ***recipe, promise,*** or ***future.*** We will use the latter term here. Further, to avoid confusion with the function **delay** as previously defined, we will define the function ***future*** to act like **delay** but to return a future instead of a simple closure.

The ⟨***boolean-flag***⟩ in a future indicates whether or not it has been evaluated. If the value is F, the cdr of the future is a typical closure that must be forced as before. If the value is T, then the closure that was in the future to begin with has already been evaluated, and the cdr of the future holds the resulting value.

The definition of the **force** function must be changed to show the difference between a closure and a future, and to handle properly the two possible states that a future can be in:

```
force(x) = "unpack and evaluate x"=
           if not(is-future(x))
           then old-force(x); Handle a closure
           elseif car(x); Handle a future—test flag
           then cdr(x); Value available
           else rplacd(rplaca(x,T), old-force(cdr(x)))
```

The operations **rplaca** and **rplacd** are as defined earlier. They are both extrafunctional operations that return the first argument passed to them, but as a side effect they physically change the car or cdr fields of that first argument to point to the values given as the second arguments.

The function **old-force** corresponds to the definition of **force** given earlier in this chapter in Section 9.1.1. The new definition works as follows. If the object is a closure, it is handled as before. If it is a future, and the flag on the future is F, then **force** knows that this is the first time the object has been encountered since it was created by **future,** and thus it evaluates the closure as before. A pointer to this result is placed in the cdr of the future, and the car is replaced by T. Consequently, the next time any **force** encounters the future, it will find the flag set to true, and it will then simply return the value stored by the prior force. The evaluation sequence is done only once, no matter how often its value is needed.

Note that this replacement operation is not something we can explain by pure lambda calculus. It is a ***side effect*** that modifies an object directly.

9.3.1 SECD Extensions

(Henderson, 1980, sec. 8.2)

If **rplaca** and **rplacd** are available as built-in instructions to the SECD architecture, then the above definition is sufficient to permit compilation and execution of futures, albeit somewhat clumsily. Compiling a **future(E)** requires not just code like

(. . . LDF "**compile(E,**RTN)" . . .)

but something more like

(. . . LDF "**compile(E,** "update and return code")" LDC F CONS . . .)

The outer "LDC F CONS" is to build the flag part of the future, and the "update and return code" is to modify the future code via RPLACA and RPLACD before returning to the ultimate caller of the future.

Also, the code for forcing the evaluation of the future must change. Instead of a simple AP0, the compiler must code in a test of the car of the future, and if true retrieve the cdr part. Only if the car is false is an AP0 executed.

A further complication is that there is no way for the code associated with the future to gain access to the actual machine memory cell that contains the future to permit the update. Such information is lost when the AP0 unpacks the future. If the original cell containing the future is not the one updated, then no other references to that original cell will see the update.

The only solution to this is to have AP0 place a pointer to the future somewhere where the update code can find it. The safest place for such a pointer is on the dump, where, for example, we know that no other function will disturb it, and where even a copying garbage collector will guarantee that what remains is a pointer to the one and only copy of that future, wherever it is.

Also, for efficiency's sake it would be helpful to avoid wasting a whole cons cell just to add a true/false flag to a closure.

The net result is that it makes more sense to augment the SECD ISA slightly to support these operations more efficiently. Figure 9-4 diagrams a revised cell structure that saves a bit for recipes and three new instructions ***LFU (Load Future), APF (APply Future),*** and ***UFR (Update Future and Return).*** They are based on somewhat similar instructions defined in Henderson (1980).

F	B	Tag	Value

F = Future tag = Future/Not Future
B = Boolean Flag (Only if Future)
= Evaluated/Not Evaluated
Combinations: ff = Future, not evaluated
ft = Future, evaluated

(*a*) Tags to support future.

Instruction	Operation
LFU	s e (LFU c.c′) d → ff#(c.e).s e c′ d
APF	f.s e (APF.c) d → v.s e c d if f=ft#v → nil e′ c′e (f s e c.d) if f=ff#(c′.e′)
UFR	v.s e (URF.c) (f s′ e′ c′.d) → v.s′ e′ c′ d and memory[f]←ft#v

(*b*) Additional SECD instructions.

FIGURE 9-4
Processing futures with the SECD Machine.

The new cell structure adds two bits to the tag. The first indicates whether or not this cell represents a future. If not, the second bit is ignored, and the rest of the tag is as previously. If so, and the second bit represents the future's flag and is F, the tag field should be a cons to support the closure. If this second bit is T, then the future has been evaluated, and the rest of the tag describes what the remaining value is. Note that these two extra bits are used for cleanliness of explanation here; a real system might very well simply bury them somehow in the overall tag coding scheme.

The LFU instruction is used in place of an LDF instruction when compiling the code for **future**. When it is executed, it will build a future cell with a false flag on the stack. None of the previously described code for generating a cons cell is needed.

The APF instruction is used in place of an AP0 to initiate a future. This instruction expects a cell of type future on top of the stack, and it works in one of two ways. If the cell's flag bit is T, then the rest of the cell gives the value directly. If the flag is F, the closure in the cell is unpacked and activated just as with AP0, except that a pointer to the future itself is pushed on the dump along with the other return information.

The UFR instruction works like RTN with one modification. The object on the top of the dump should be the initiating future, and that memory cell is changed to one where the two future tag bits are changed to reflect an evaluated future, with a value as specified.

With these instructions, a **future (E)** would compile into

(...LFU (...code from **E**...UFR)...)

and **force (E)** would compile into:

(..."code to place future **E** on stack" APF...)

9.3.2 Opportunities for Parallelism

The introduction of futures also gives us some excellent opportunities for parallel execution and possible speedup of solution time. At the execution of a **future** function we have an opportunity for a ***fork.*** The processor that was executing the **future** can construct the future as above and continue execution. A new processor, however, can then take the future and start evaluating it. (It should set the cell's T as to yet a new code "busy" to prevent other processors from evaluating it also.) At completing of the evaluation, the future is replace by (T. "value") as before.

Adding a third value for the flag clearly cannot be done with the one bit allocated in Figure 9-4. However, it can be included by taking the two bits together and having four codes: future unevaluated, future in process of being evaluated, future fully evaluated, and nonfuture.

Now if the original processor (or any other) attempts to force the future, one of three alternatives can result. If the combination says evaluated future, the value is directly available as before. If it is busy, then some other processor is evaluating the future, and this one must loop until the flag goes to false. Finally, if the flag is false, then no one else has attempted to evaluate the future, and it is up to this processor to force the closure. Again, for protection against any other processor evaluating the future, the tag should be set "busy" while the closure evaluation is in progress. This operation should be uninterruptible from the time the tag is fetched until it is updated to busy. Otherwise two processors accessing the flag at the same time may start to evaluate it, with potentially chaotic results.

An interesting alternative to a "busy-wait" loop by a processor that finds a future in the process of evaluation is to package its current computation in a continuation (see Section 9-6), leave the continuation in the future's value cell, and declare it a "free" processor. The processor evaluating the future will, at its completion, test the future cell before update. If the cell contains a continuation, the processor will pickup that computation directly rather than declare itself free. This will tend to minimize interprocessor task switching and maximize the time available for processors to perform useful work.

This expanded definition of **force** is a very good match to the conventional definition of *join* as a synchronization operation after a fork in the execution sequence to parallel processors.

9.4 LAZY EVALUATION

(Henderson, 1976, 1980)

In conventional computing there are several semantic models for passing arguments to subroutines, procedures, or subprograms. Two common ones are:

> ***call-by-value,*** where actual arguments to a subprogram call are fully evaluated before passing the resulting values to the subprogram, usually by copying their value into storage associated with or allocated to the called subprogram
>
> ***call-by-name,*** where argument expressions are converted into simple subroutines which are called each time a reference to the matching formal argument is made in the subprogram's execution

Call-by-value corresponds to applicative-order reduction and is normally faster than call-by-name, but it can evaluate operands that are never needed, and it can potentially get caught in an unnecessary infinite loop. Call-by-name corresponds to normal-order reduction; it avoids the potential infinite loop that call-by-value might get caught in, but at the expense of duplicate evaluation of the same argument expressions.

The introduction of futures and the modified **delay** and **force** give us

an opportunity to introduce new forms of parameter passing (often termed ***call-by-need***), which combine the best of both the above forms. When a user-defined function is called, all actual arguments are automatically delayed, resulting in futures for each. When any argument is actually needed, the **force** function is applied to it. If this is the first time the argument has been accessed by this function, the **force** evaluates it. Otherwise, the value is already available (or is on the way).

This concept can be carried one step further, with both the **futures** and the equivalent **forces** extended down to the most primitive operation level. The result is that nothing is ever computed until its value is absolutely needed, and then only enough is computed to satisfy the particular need. For obvious reasons this is called ***lazy evaluation.***

The following subsections address both approaches. The interested reader is also referred to Bloss et al. (1988) for a detailed discussion of compiler optimization techniques when lazy evaluation is available.

9.4.1 Compiler Processing of Call-by-Need

The simplest way to automatically implement delayed evaluation of any sort in a functional language is to have the compiler, or a preprocessor to the compiler, insert the appropriate function calls into a program code. These calls are then expanded into machine code that mirrors the desired operations. At its simplest, this technique would surround each actual argument expression in any nonprimitive application by **future**(...), and each use of a function's formal argument identifiers by **force**(...). Thus, in something like

let **f**(x)=1+x in **f**(3+2)

A **future** would enclose the "3+2" because that represents the argument to a function evaluation, and a **force** would be added around the x in "1+x" because that represents a use of the variable in the function's body. Note that **future**s around either constants or arguments to a built-in function such as "+" do not make sense, and should be avoided.

Figure 9-5 gives a bigger example by diagramming how a programmer might have written the earlier prime-number example in a language where applications are automatically considered to be call-by-need and then what the code might look like after the compiler inserted the appropriate **future**s and **force**s.

One note about this automatic insertion of functions: Inside a function's body, if an argument to some other function is simply one of the outer function's arguments, then the above rules would surround the argument by a **force** (because it's a use of the argument) and then by a **future** (because it's an argument to another application). This is an obvious waste and should be avoided. In such cases neither the **future** nor the **force** should be included and the argument should simply be passed

```
primes(x) produces cons(np, future(primes(np+ 1)))
where np is the next prime ≥x

primes(x) = "list of all primes starting at x"
  if x<3
  then cons(x,primes(x + 1))
  elseif indivisible(x,primes(2)) ;x div. by any prime?
  then cons(x,primes(x + 2)) ;x = next prime
  else primes(x + 2) ; try next odd number
  whererec indivisible(x,p) =
    if car(p)>x/2
    then T ;x indivisible by no element in p
    elseif remainder(x,car(p)) = 0
    then F
    else indivisible(x,cdr(p))
```

(*a*) As written by programmer.

```
primes(x) = "list of all primes starting at x"
  if force(x)<3
  then cons(x,future(primes(future(force(x) + 1))))
  elseif indivisible(x,future(primes(2))) ;x div. by any prime?
  then cons(x,future(primes(future(force(x) + 2)))) ;x = next prime
  else future(primes(future(force(x) + 2))) ; try next odd number
  whererec indivisible(x,p) =
    if car(force(p))>force(x)/2
    then T ;x indivisible by no element in p
    elseif remainder(x,future(car(force(p)))) = 0
    then F
    else indivisible(x,future(cdr(force(p))))
```

(*b*) As processed by compiler.

FIGURE 9-5
Call-by-need.

unmodified. Figure 9-5(*b*) takes this into account. Appropriate compiler implementation can handle this.

9.4.2 Lazy Evaluation at the Lowest Level

Call-by-need seems such an ideal combination that it would seem reasonable to try to push it even further to include built-ins and other operations, and to implement these changes as close to the machine level as reasonable. This means inserting at least portions of **future** and **force** functions in appropriate SECD instructions. This section discusses where such places might be.

BUILDING FUTURES. A summary of the places in a program where futures should be built include:

- All argument expressions to user-defined functions
- The arguments for the unnamed applications defined by let and letrec
- Both arguments to any use of the **cons** function

The first two places correspond to a generalization of call-by-need. The last place comes from observing the advantages of the "infinite list" data structures discussed earlier. In this case, delaying both arguments permits data structures which are potentially infinite from either the car or cdr position.

Use of the LFU and UFR instructions to surround the code for each of the above expression types will build the futures at the appropriate time. Note again that we can avoid the future construction when the argument expressions are either constants or arguments from a enclosing function.

FORCING ARGUMENTS. The placement of **force**like operations should reflect the desire to minimize expression evaluation to the point where it is done only when absolutely necessary. In terms of our abstract programming this includes only:

- The arguments to all primitive and built-in instructions except CONS
- The conditional expression in a test
- The function expression in an application

The first of these represents places where, if argument values are not available, the operation cannot be carried out. The exclusion of the CONS instruction reflects the earlier decision to delay its arguments. Note that this does not include internal uses of the **cons** operation inside many instructions to perform basic operations, such as placing something on the dump.

The second choice represents the need to have a value to inspect before deciding which of two other expressions to evaluate. Again, this is an absolutely necessary place to force evaluation.

The final choice represents the need to have a closure to unpack and branch to when a function is invoked.

The **force**like operations that must be buried in the appropriate instructions are somewhat different from those described above because a compiler may not have any way of guaranteeing that an object won't end up buried inside several layers of redundant futures, all of which must be removed before a value is available. This requires that an object that is to be forced be forced repeatedly, until the result comes back as something other than a future. This can be defined with the help of two functions:

is-future(x) returns true if x is an unevaluated future.

repeat-force(x)=if **is-future**(x) then **repeat-force**(**force**(x)) else x

With these functions, all the SECD instructions that represent real computations on objects on the stack (with the exception of CONS) are modified to apply **repeat-force** to the arguments they read off the stack before computing the result. Thus, for example, the register transition for ADD becomes:

a b.s e (ADD.c) d→(**repeat-force**(a)+**repeat-force**(b)).s e c d

This includes not only the instructions corresponding to built-ins but also instructions like SEL (to permit a choice of targets), AP, AP0, and RAP (to evaluate the function object down to a closure), and WRITEC (to deliver a result). Other instructions, such as NIL, LDC, and LD, do not require such modifications. Any arguments they handle are passed unchanged.

The net result is truly ***lazy evaluation***—absolutely nothing is evaluated until absolutely necessary, and then only to the point of returning a nonfuture at the outermost level.

9.4.3 Equal Leaves Example

(Henderson and Morris, 1976)

As an example of the potential computational savings offered by lazy evaluation as described above, consider Figure 9-6. The main function **eqleaves** takes two arbitrary s-expressions, and verifies that the leaves of these expressions have the same values in the same order, without regard to internal list structure. Thus **eqleaves**((1.(2.(3.4))),((1.2).(3.4))) evaluates to true, while **eqleaves**((1.(2.(3.4))),(1.(2.(3.5)))) does not.

The function works by first calling **flatten** on each argument. This function takes the leaves of a list and "flattens" them into a single-level list. Thus **flatten** applied to both of the above arguments returns (1 2 3 4). The function **eqlist** then compares the two lists element by element. At the first mismatch it reports F. Given that the two lists are computed lazily, this means that the two original inputs are flattened only as long as their terminal leaves are the same.

```
eqleaves(x,y) = "test if 2 s-expressions x and y have same leaves in same order"
 = eqlist(flatten(x),flatten(y))

flatten(x) = "list of all leaves in x, in order of appearance"
   if atom(x)
   then (x.nil)
   else append(flatten(car(x)), flatten(cdr(x)))

eqlist(x,y) = "test if lists x and y are identical"
   if null(x)
   then null(y)
   elseif null(y)
   then F
   elseif car(x)=car(y)
   then eqlist(cdr(x),cdr(y))
   else F
```

FIGURE 9-6
A lazy leaf comparator.

Now consider what happens to

eqlist((1.**huge-list**), (2.**huge-list**))

where **huge-list** is some giant, possibly infinite list of arbitrary shape. The two lists are obviously unequal, but a conventional interpretation or compilation of Figure 9-6 would flatten both lists completely before attempting any comparisons. This represents a tremendous (possibly never-ending) amount of wasted effort.

On the other hand, a lazy interpretation of exactly the same program never ever touches the **huge-list.** As Figure 9-7 shows, a nearly minimal amount of computation is involved, with the critical comparison being done quite early. Lazy evaluation has proved successful.

9.5 STREAMS

(Burge, 1975, pp. 133–146; Henderson, 1980, pp. 231–239)

Now consider the typical ***producer-consumer problem*** as diagrammed in Figure 9-8. This is representative of computing expressions of the form **f(E)**, where the expression **E** is a ***producer*** of a potentially infinite list of objects and the function **f** is a ***consumer*** of them. In the real world many such problems have ***time constraints*** of some sort, often dictated by the rate at which the expression **E** generates objects or at which the final results must be available. An example would be a producer that reads a file from a disk and a consumer that formats and prints it out. The program cannot run any faster than the rate at which data can be obtained from the input (such as a disk drive), but ideally fast enough to drive a printer at full rate.

An even better example is the ***read-compute-print loop*** or ***listener*** described in Chapter 7 as a way to provide an s-expression interpreter with a series of expressions to be interpreted and to display the results. These input expressions usually come from a human typing at a terminal at a decidedly varying rate. However, they might also come from a disk file or even from the output of the interpreter itself (as it is interpreting an interpreter for some other language).

A typical problem with such problems is ***rate matching*** the output from the producer with the input to the consumer. Usually some sort of buffer is needed between the two to smooth out the differing rates. Some classical solutions to this include:

```
[1]eqleaves(x0,y0) where
  [1]x0 = (1.huge1) where huge1 is some huge list
  [1]y0 = (2.huge2) where huge2 is some huge list
= [2]→eqlist(x1,y1) where
  [2]x1 = flatten(x0)
     = [3]→if atom(x0) then ... else append(...)
     = [4]→ append(x2,x3)) where
       [4]x2 = flatten(car(x0))
          = [5]→ if atom(x4) then cons(x2,nil) else ... where
            [5]x4 = car(x0)
               = [6]→ 1
          = [7]→ cons(1,nil)
       [4]x3 = flatten(cdr(x0))
          = [8]→cons(x5,x6) where
            [8]x5 = car(x2) = [9]→ 1
            [8]x6 = append(cdr(x2),x3) = [10]→ cons(1,x6)
  [2]y1 = flatten(y0) = [12]→ ... = [19]→ cons(2,y6)
= [11]→if null(y1) then false
  elseif car(x1) = car(y1)
  then eqlist(cdr(x1),cdr(y1))
  else F
= [20]→ F
```

Numbers in [ ] refer to following points in the computation:

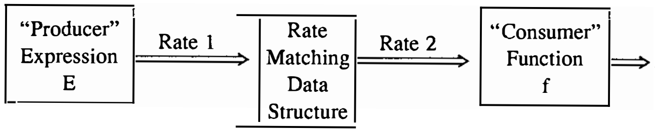
1. Start of calculation.
2. Expansion of eqleaves, argument evaluation x1, y1 deferred.
3. Need x1 for null(x1), so force its expansion.
4. x0 is not an atom, so x1 becomes the else of [3] = call to append with new deferred arguments x2 and y2. Expand append.
5. Need null(x2) inside append, so force x2's expansion. Defer its argument x4.
6. Need atom(x4), so force x4. It evaluates immediately to 1.
7. Expansion from 5 becomes then part cons(x4,nil). Defer arguments to cons.
8. Now x2 is not an atom, so the current x1 expansion reduces to a cons.
9. The car part of the cons itself needs a car, forcing x2 from [7] to 1.
10. The cons part defers the rest of the arguments, leaving a lazy x1.
11. Now the first null in the expansion from [2] fails (x1 a cons), leaving an equivalent test of y1.

12–19. Equivalent expansion of y1.

20. The null y1 from [11] fails, forcing car(x1) and car(y1) to return 1 and 2. These are unequal, the = test fails, and the expression reduces to F.

FIGURE 9-7
Sample lazy evaluation.

1. Following a nonlazy lockstep execution, where each **E** output must be consumed by **f** immediately
2. Producing the entire **E** output first, then giving to **f** one item at a time
3. Using co-routines by which **f** and **E** alternately call/return to each other



Functional expression form: z = f(E)
FIGURE 9-8
Typical producer-consumer problem.

All of these solutions have problems. The first two include no opportunities for parallelism. In approach 1, the system runs at best at the slower of the two rates. In approach 2, each run proceeds at its own rate, but there are large, potentially infinite, intermediate data buffers involved, with a significant time delay between start of production and start of first consumption. The third approach is more reasonable, but it is much more difficult to program explicitly.

A better solution is based again on closures and delayed evaluation. The key data structure is a generalization of a future and is an s-expression list where all but the last element are data items produced by **E** (in order) and not yet consumed by **f**. The last element is a future for **E**. The function **f** consumes this list from the front at its own rate, and the future for **E** produces items at the back end as either processors exist to evaluate it or as real-time constraints force its evaluation (someone hits a keyboard).

Formally, there are several key definitions to consider. First, a ***sequence*** is a recursively defined s-expression where the car is the first element and the cdr is either a closure or a sequence to the rest. The list of primes defined in Section 9.2.1 is an example of this, as is the intermediate data between E and f above.

A ***stream,*** alternately called a ***stream function,*** is a function that when applied to an empty argument list (i.e., forced) returns a cons pair where the car is the first element of the sequence and the cdr is a stream function that can produce the rest. The expression E either contains such a function or is itself such a function (constructed by delaying some expression).

When used in a future in the terminating position of a list, evaluating the stream function will replace the future by a new cons pair. The net effect is the appending of the newly computed object to the back end of the overall sequence, with a new future as the last cdr. Forcing this new future will continue to extend the sequence.

Note that this definition of future is slightly different from our earlier examples of infinite lists, where the first element of the list is always available. In this definition the stream itself is a future; only when forced does it give up even its first element. Either definition is possible. The choice here is simply for contrast.

9.5.1 Stream Processing Functions

There are a variety of standard forms for functions that process streams. Figure 9-9 lists some generic ones that can be tailored to a specific application by inserting a function as the first argument. These definitions use explicit **future** and **force** calls to clarify when things happen. An underlying language with fully lazy semantics would not need them.

A ***process function*** applies a function provided as an argument to every element of a stream (or set of streams), and produces a stream as an output.

A ***filter function,*** **filter**(*p, s*), takes a stream *s* as one input and a property function as another, and produces a new stream whose elements are all those from *s* which satisfy some predicate function *p*.

A ***merge function*** takes a predicate function as one argument and two streams as two other arguments, and produces a stream as a result. This output represents a merge of the two input streams, where applying the predicate argument to the top two elements of the input streams determines which should be placed on the output stream.

```
Note: force(<stream>) → (<object>.<stream>)

process(f, s) = "apply f to every element of s"
  let x=force(s) in
      future(cons(f(car(x)), process(f,cdr(x))))

process2(f,s1,s2) = "apply f to evey element of s1 and s2"
  let x=force(s1)
  and y=force(s2) in
       future(cons(f(car(x),car(y)), process2(f,cdr(x), cdr(y))))
```

(*a*) Some generic process functions.

```
filter(p,s) = "pass only elements of s that pass predicate p"
  let x=force(s) in ; evaluate the stream
     if p(car(x)) ; Does 1st element have desired property?
     then future(cons(car(x), filter(p,cdr(x)))) ;Yes
     else filter(p,cdr(x)) ; No, wait for next element
```

(*b*) A generic filter.

```
merge(p, s1, s2) = "p selects car(s1) or car(s2)"
  let x=force(s1)
  and y=force(s2) in
       let carx=car(x)
       and cary=car(y) in
            if p(carx, cary)
            then future(cons(carx, merge(p, cdr(x), y))
            else future(cons(carx, merge(p, cdr(x), y))
```

(*c*) A generic merge.

FIGURE 9-9
Generalized stream functions.

9.5.2 Stream Example

Figure 9-10 diagrams a set of streams which use most of the functions of Figure 9-9 and together compute three streams, that of the integers starting at 1, the integers starting at 2, and the factorials of all positive integers. Note in particular that both **integers** and **factorials** are streams that are recursively defined in terms of themselves.

9.6 CONTINUATIONS

(Haynes et al., 1986)

A closure encapsulates a function and its environment. A future encapsulates an expression. Although this would seem to be complete (and it is theoretically), there is yet another alternative that provides some extraordinarily powerful programming features. This alternative, termed a ***continuation,*** turns an expression evaluation inside out and repackages it into an object that looks somewhat like a closure, but with some critical differences:

- It contains not just a starting code address and an environment, but the entire state of the machine, including the stack and the dump.
- It represents a function of exactly one argument which, if restarted with an argument value, resets the machine to the enclosed values and restarts the computation, with the argument value passed to it left as the value of the expression that created the continuation.
- The only thing left of the expression evaluation that restarted the continuation is the value passed back to it as an argument. The rest of its state is discarded.

There are variations in which this last point is not strictly true, and the result of the continuation could be returned to the caller of the continuation.

Uses of such a mechanism include fast escape paths from inside a computation, easy manipulation of one whole program by another, an elegant approach to permitting two or more entirely separate programs to exchange messages and pass control back and forth, and the ability to "nondeterministically" select from some set of expressions a particular

```
integers=future(cons(1,process((λx|1+x),integers)))

s1=filter((λx|1<x),integers)

factorials=future(cons(1,process2((λxy|x×y),s1,factorials)))
```

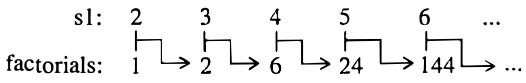


FIGURE 9-10
A stream of factorials.

one that has some properties, without explicitly programming in loops of some sort. This latter ability is of particular interest, because a very similar mechanism will form one of the linchpins of the most common of logic-based systems discussed in the second half of this book.

The following subsections give a formal definition of continuations, describe examples of their uses, and discuss methods of implementing them in the SECD Machine. Other examples of their use will be mentioned in conjunction with real function-based programming languages, particularly ***SCHEME.***

9.6.1 Definition

A ***continuation*** represents an expression evaluation that is stopped just as some particular subexpression is to be evaluated. This subexpression by itself may or may not have a value, and is there primarily to trigger the encapsulation process. Given the semantics of expressions, restarting the computation requires providing a value for this subexpression.

In this book, the subexpression that causes a continuation to be formed is patterned after that from SCHEME and is of the form:

call/cc(f)

where ***call/cc*** stands for a built-in function named ***call with current continuation.*** This function takes exactly one argument **f,** which must itself be a function of one argument, that is, it must be of the form **f**=(**λ***c*|**E**).

When executed, **call/cc** takes the current state of the machine, packages it into a continuation object we will designate here as **C,** and applies it as an argument to the function **f.** Thus **call/cc(f)** $\Rightarrow$ **f(C),** where **C** is the state of the computation when **call/cc** was started.

The function **f** can do anything, including ignoring this continuation argument value and simply returning some value. If it does so, the computation of the expression containing the **call/cc(f)** continues exactly as if the continuation were never built and **f** were a simple expression.

The interesting capability arises when we consider what could be placed inside **f.** This function could do anything. In particular, it could, at some totally arbitrary point of its own evaluation, take its argument value **C** and treat it as a function in an application involving exactly one argument also. The result of treating the continuation as a function is to unpackage it, reset the machine to the unpackaged values, and restart that computation with the argument to the continuation treated as the value of the expression **call/cc(f).** Thus, if **f** were of the form (**λ***c*|...(*c* **E**)...), and (*c* **E**) was evaluated at some point, we would have the reduction string:

$$
\begin{aligned}
&\mathbf{g}(\ldots,\mathbf{call/cc(f)},\ldots)\\
&\Rightarrow (\boldsymbol{\lambda} c|\ldots(c\ \mathbf{E})\ldots)\mathbf{C}\\
&\Rightarrow [\mathbf{C}/c](c\ \mathbf{E})\\
&\Rightarrow (\mathbf{C\ E}) \Rightarrow \mathbf{g}(\ldots,\mathbf{E},\ldots)
\end{aligned}
$$

Where things get really interesting is when the expression **E** returns as part of its value a closure, or even another continuation, from inside itself. It is then possible for the outer function **g** to reenter **E** long after **E** has been "completed" in the normal sense and after the computation surrounding **E** has been supposedly "discarded."

9.6.2 Continuations in Action

Figure 9-11 gives several examples of the use of continuations. Example (*a*) diagrams a common use—a fast exit from the middle of a program. In this case the function **product** returns the result of multiplying together a list of numbers. One optimization for such a function is to quit early if an element which is a 0 is found. The answer will be 0 regardless of what the rest of the list is.

There are a variety of ways to avoid the extra multiplications. An accumulating parameter provides one approach. Figure 9-11(*a*) gives another, where **product** calls a subfunction via the **call/cc** mechanism. The subfunction, (**λ***c*|**prod**(*c, val*)), is provided with a continuation as an argument for *c* that will return a value to the **product** application. In turn, this

```
product(val) = call/cc((λc|prod(c,val))

prod(c,val) = "tail recursive product, with escape to c at a 0"
    if null(val)
    then 1
    elseif car(val) = 0
    then c(0)
    else car(val) × prod(c,cdr(val))
```

(*a*) As a fast basis termination.

```
letrec listener(c) = (λx|listener(c))(write(eval(c,read()))) in
    write-error-msg(call/cc(listener))

eval(c,e) = "evaluate expression e with c as error continuation"
    else c("error code") ; escape listener to error handler
```

(*b*) As an error exit.

```
letrec listener(c) = (λx|listener(c))(write(eval(c,read()))) in
    debug-program-state(call/cc(listener))

eval(c,e) = "evaluate expression e with c as error routine"
    else call/cc((λc2|c(c2))) ; return machine state to debugger

debug-program-state(c) = "fix up and return buggy expression"
    let x = print-state(c)
    and y = fix-up-state(c) in c(y)
    ; return fixed up value to eval continuation
```

(*c*) Interprocess communication.

FIGURE 9-11
Using continuations.

causes **prod(C,*val*)** to be evaluated. If the list *val* has no 0s in it, execution is just as expected, with a recursive walk through the list, multiplying the values. The final value is returned as the value for **product.** The continuation provided as the first argument is never used.

If, however, a 0 is found, this continuation argument is invoked as a function, with a 0 as its argument. This restarts the **product** computation immediately, with a value of 0 returned. All intermediate nested function calls, environments, return information, pending multiplication, etc., are discarded.

Figure 9-11(*b*) gives a more common example of continuations. The ***listener*** function is basically an infinite loop that reads expressions, evaluates them, and prints the result (see Chapter 6). The unique addition here is the extra argument to **listener,** which should be a continuation to be restarted in the event of an error in the evaluation. This argument is passed as an extra argument to a modified version of **eval,** which does do error checking. When **eval** finds an error, it uses the continuation as a function and passes it an error code. This in turn skips out of the **read-eval-write** loop directly to the **write-error-msg** function.

What is convenient about this is that it makes it unnecessary for programmers of any intermediate-level functions to worry about how to pass back error or exception conditions generated either by them or by functions that they call.

Finally, example (*c*) in Figure 9-11 represents a case much like example (*b*), except that the error routine is actually a debug routine which is passed as an argument to the continuation of **eval** at the time the error is detected. This permits the debugger to examine the entire evaluation state, perhaps ask the user what the expression should be, and then restart the evaluation with a replacement value.

The astute reader will notice that in many of these cases functions are defined that include in them free variables bound by higher-level functions. Given the definition of closures, this is handled properly by binding the environment containing these variables to the code at the time the function is defined. It does represent, however, a case where the ***funarg*** problem described in Chapter 8 must be handled properly by the underlying implementation.

9.6.3 Co-routines

Many important programming problems can be solved by a technique called ***co-routining,*** whereby two or more programs pass data and control between themselves from intermediate and potentially varying points in their code. A classic example is the ***producer-consumer problem,*** in which one program, the producer, generates data to be passed to a second program, the consumer. What makes this different from a simple recursive loop of one calling the other is that each program may have an internal state which needs to be remembered from call to call, and where the point to which the code should return after a call completes can vary dramatically in time. The internal state, for example, might consist of the number of buffers that are full, which I/O device will be used next, what error code was returned by the other co-routine on the last call, etc.

One way to handle this elegantly with continuations is with a doubling of Figure 9-11(*c*). Each of the co-routines passes to the other at each call a continuation representing the point where it should be restarted when the other is done. This continuation can differ at each call.

Figure 9-12 represents an example. The producer function is assumed to be the original expression started. It performs an initial call to the consumer function with an argument that consists of the cons of an initialization value and a continuation back to the producer. Note the way this argument is formed and embedded via a lambda function as a single argument function to **call/cc.**

After the consumer expression initializes itself, it calls this continuation with an argument which represents the cons of a response value and a continuation back into the consumer, formed exactly the same way as before.

This continuation sends execution back into the middle of the producer, which, after arbitrary processing on the response value from the consumer (and values bound to internal variables), performs another **call/cc** back to the consumer, using the continuation provided by the con-

```
producer= ‘‘assume this is starting process’’
  let y1 =call/cc((λc|consumer(cons(‘‘initial value’’,c)))) in
  ;Assume response value y1 =(value.continuation)
  ;Resume coroutine by restarting continuation from y1
  ; with value of (‘‘f1 applied to y1’’.‘‘producer continuation’’)
      let y2 =call/cc((λc|(cdr(y1))cons(f1(car(y1)),c))) in
      ;y2 is (value.continuation) that coroutine returns.
      ;Resume coroutine by restarting continuation from y2
      ; with value of (‘‘f2 applied to y2’’.‘‘producer continuation’’)
          let y3 =call/cc((λc|(cdr(y2))cons(f2(car(y1)),c))) in
             ’...’ ; more of same—exchanging values with consumer

consumer(x1)=  ; Assume this is coroutine initialized by producer
     ;Argument x1 =(‘‘initialization value’’.‘‘producer’s continuation’’)
     ;Resume producer coroutine by restarting continuation from x1
     ; with value of (‘‘g1 applied to x1’’.‘‘consumer continuation’’)
     let x2=call/cc((λc|(cdr(x1))cons(g1(car(x1)),c))) in
     ; x2 is (value.continuation) that producer returns.
     ; Resume producer by restarting continuation from x2
     ; with value of (‘‘g2 applied to x2’’.‘‘consumer continuation’’).
         let x3 =call/cc((λc|(cdr(x2))cons(g2(car(x2)),c))) in
              more of same—exchanging values with producer
```

FIGURE 9-12
A basic co-routine structure.

sumer as a function. This causes the consumer to pick up exactly where it left off, but with a new value from the producer.

The process of alternating between producer and consumer can continue indefinitely, and with complete safety. Each side always knows exactly where it will be started up again, without having to code that information into the other side's code.

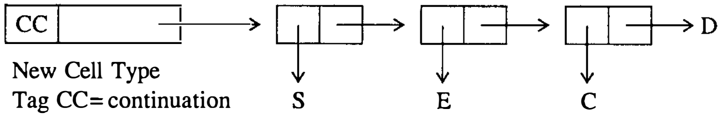
This concept has obvious applications in many operating systems and multitasking applications.

9.6.4 Modifying the SECD Machine

Modifying the SECD Machine to support continuations can be done in a variety of ways. The approach described in Figure 9-13 is not necessarily the most efficient, but it is perhaps the simplest. We start by defining a new cell type with a tag of "continuation," abbreviated cc. The value part of such a cell is a pointer to a list that provides in turn copies of the S, E, C, and D registers at the time the continuation was built.

To accompany this, the SECD ISA needs one instruction change, to ***AP***, and one new instruction, ***CC*** (for ***Call Continuation***).

The AP change involves making it recognize the difference between a closure as a function providing argument and a continuation. In the former case, operation is exactly as before. In the latter case, the S, E, C, and D registers are reloaded from the continuation, with the second element of the original stack added to the reloaded environment as the argument list for the continuation's evaluation. Normally this is a single item.



(*a*) Continuation data structure.

Instruction	Operation
AP	(x v.s) e (AP.c) d → nil (v.e') c' (s e c.d) ;if x a closure [c'.e'] → (v.s') e' c' d' ;if x = cc#(s' e' c' d')
CC	(x.s) e (CC.c) d → nil ((v).e') c' d' where x is a closure [c',e'] and d' = (s e c.d) and v = cc#d' (a continuation)

(*b*) Instruction changes.

FIGURE 9-13
SECD modifications to support continuations.

The new CC instruction operates somewhat like AP0 in that it does not expect any argument lists from the caller, but also like AP in that it does augment the new environment with an object, in this case a copy of the current D register but with a "cc" tag. This is the continuation.

9.6.5 Nondeterminism and Backtracking

Another class of problems amenable to solution with continuations are termed ***nondeterministic.*** In such problems there are a variety of values possible for some expression, none, some, or all of which are inapplicable when the expression is embedded in some larger one. From the programmer's point of view, it is immaterial which of the correct values is returned. A simple example might be determining a combination of values on two dice that equals some specific number.

Such problems are nondeterministic because they do not demand that we try the possible values from the subexpression in any particular order, only that we find one that works. It is possible that two different evaluations of the program might then give two different answers, even if nothing had been changed in their representation.

The classical approach to solving such problems is through ***generate and test.*** One value is generated from the subexpression with multiple possibilities and passed to the rest of the expression for test. If that passes, the other expression returns its value. If for some reason that value is inappropriate, the original subexpression is reentered, a new value is chosen, and the outer expression is retried.

Figure 9-14 is an extended abstract program that demonstrates this. The two subexpressions bound to *dice1* and *dice2* are computed by the ***try*** function. This function may take an arbitrary number of arguments, and returns one of them, the value of which will "magically" be guaranteed not to cause the rest of the expression evaluation ever to end up in an expression ***fail.*** Which value is returned is immaterial.

Continuations are an excellent mechanism to implement this. They can capture the computation at the inner point of the **try** expressions where some value is chosen. If, when passed through the rest of the expression, that value turns out to fail some outer tests (as triggered by ex-

```
let dice1 = try(1,2,3,4,5,6)
and dice2 = try(1,2,3,4,5,6) in
    if dice1 + dice2 ≠ 7
    then fail ;try guarantess this will NOT be result
    else print((dice1.dice2))
```

FIGURE 9-14
Sample nondeterministic expression.

ecuting the **fail** function), the continuation built by **try** can restart the entire computation with the next value. This is usually called ***backtracking.***

Nested **try**s are possible, and if all the solutions for one **try** fail, a backtrack to some other **try** could be attempted, which in turn might restart the failed **try** all over again from the beginning, but with new values for other bindings.

One method for extending the SECD Machine to implement such functions is shown in Figure 9-15. A fifth register B (for ***backtrack register***) is added to the basic machine. This register will function as yet another stack, this time one of continuations. Each time a **try** function is evaluated, a continuation is pushed onto B to record a restart point. The ***TRY instruction*** performs this operation. Likewise, each time a ***FAIL instruction*** is executed, the top continuation on this stack is removed and used to restart the machine.

From a compiler's viewpoint, the code generated for a **try** is straightforward. The whole **try** expression is treated something like a closure, with an LDF used when the **try** is first encountered. Inside the code bracketed by the LDF, each expression is compiled into a TRY instruction followed by a list of code that computes the value and terminated by a RTN. The last instruction in the code loaded by the LDF is a FAIL. This will indicate that there are no more possibilities.

An AP0 instruction executes the closure loaded by the LDF. This adds no arguments to the environment but leaves a return address on the dump.

Any expression of the form **(fail)** translates directly into a FAIL instruction.

Instruction	Operation
TRY	s e (TRY ca.c) d b → s e ca d (cc#s e c d).b
FAIL	s e (FAIL.c) d (cc#s′ e′ c′ d′).b → s′ e′ c′ d′ b

b represents a new SECD register—the Backtrack Stack

Compiler Modifications: (see Figure 7-20)

Syntax: (TRY expr$_1$ ... expr$_n$)
Code: (LDF)‖((TRY (*expr$_1$‖(RTN)) ... (TRY (*expr$_n$‖(RTN)) FAIL)

Syntax: (FAIL)
Code: (FAIL)

```
Example: (IF (GREATER (TRY 4 3 2 1) 3) (FAIL)...)
         → (LDC 3 LDF (TRY (LDC 4 RTN) TRY (LDC 3 RTN)
                         TRY (LDC 2 RTN) TRY (LDC 1 RTN))
            FAIL) AP0 GREATER SEL (FAIL JOIN) (...) ...)
```

FIGURE 9-15
Implementing nondeterminism via backtracking.

In execution, encountering the LDF (...) AP0 combination causes the code inside the LDF to be entered, but with the address of the instruction following the AP0 pushed on top of the dump. Supposedly the code after the AP0 will process the value returned by the **try** expression.

Upon entering the code bound by the LDF, a TRY pushes onto the B register stack a continuation where the code address is the cddr of the cell containing the TRY. This should be the point where other code would return some other value for the **try.**

Inside the list immediately following a TRY is a sequence of code to compute one of the argument expressions. The last instruction here is a RTN, causing that value to be returned to the stack and control resumed after the AP0.

Execution of a FAIL instruction causes the top continuation on the B stack to be removed and reactivated. This will reset the machine to exactly the state it was in when the prior TRY was executed, except that the C register points to the instruction beyond the sublist that computed the last value. In a **try** expression with multiple arguments, this code will be yet another TRY (... RTN) sequence, which will restack a continuation on B, compute another value, and return that to the rest of the code as if the first TRY sequence had never been attempted. If there are no more TRY segments, a terminating FAIL will back up to the previous continuation before this one, and restart that.

This whole mechanism greatly resembles that used in the ***WAM*** abstract machine discussed in Chapter 17 for logic programming.

9.7 PROBLEMS

1. Rewrite **integers** from Figure 9-1 and **first** from Figure 9-2 so that the value returned by **integers** and used as the second argument of **first** is a closure (not a cons), which if forced would return a cons of a number and another closure. Generate the corresponding SECD code.
2. Modify the SECD compiler of Chapter 8 to handle explicit delays and forces as per Section 9.1.1. (*Hint:* The major difference is in the association list passed to the recursive call of the compiler.)
3. Is **delay(force(E))** the same as **E**?
4. How many explicit **delay**s and **force**s are executed in the expression **first**(3,**primes**(2)), assuming the definitions of Figure 9-2 and Figure 9-3.
5. How would you represent an evaluated future whose value is itself a future, either evaluated or unevaluated?
6. Compile the following, assuming that fully lazy evaluation is in effect and the instructions from Figure 9-4 are available:

   letrec **integers**(*m*)=**cons**(*m*,**integers**(1+*m*)) in **integers**(17)
7. Rewrite the **primes** function given in the text to use an accumulating param-

eter to avoid recomputing **primes**(2) for each attempted x. Under what constraints does this program work, and how does it differ from the original?

8. Extend the syntax of pure lambda calculus (Figure 4-1) to include empty lists and the interpreter of Figure 6-3 to handle lazy evaluation.
9. Modify Figure 6-9 to handle lazy evaluation.
10. Convert let x=**delay**(3×5) and **f**(z)=$1+z$ in **force**(x)×**f**(x) into SECD Machine code.
11. Consider the abstract function **lazymap2**(f, $x1$, $x2$), where f is a function of two objects and $x1$ and $x2$ are infinite streams of the form (⟨***object***⟩.⟨***closure***⟩). This function should return a similar infinite list where each element is the result of applying the function f to the matching elements of the other two inputs.
    a. Write an abstract program that implements this function using explicit **delays** and **forces**.
    b. Write another abstract program for the form where $x1$, $x2$, and **lazymap2**'s output are all closures themselves, which if forced returns (⟨***object***⟩.⟨***closure***⟩).
    c. How do the above two programs differ from a fully lazy implementation?
12. Consider the expression:

    letrec $s1$=**cons**(1,**delay**($s2$))
    and $s2$=**cons**(1,**delay**($s3$))
    and $s3$=**lazymap2**(($\lambda xy|x+y$), $s1$, $s2$)
    and **lazymap2**(($\lambda xy|x+y$), $x1$, $x2$)=... (as in previous problem)
    in $s3$

    a. What exactly is computed by this expression?
    b. Compile it into SECD code.
13. Write a form of **listener** in which the function **readexpr** is a producer for **evalexpr,** which is in turn a consumer for **writeexpr.** Use continuations to pass control. You do not need to show details of **eval** or any I/O routines.
14. Implement in SECD code using the CC instruction the expression of Figure 9-11(a).
15. Compute the **factorials** stream of Figure 9-10 but using versions of **reduce** and **first** that are adapted to streams. Show your definitions of these latter two functions.
16. Develop SECD code for Figure 9-14, assuming the modifications of Figure 9-15.
17. Discuss how the suggestion at the end of Section 9.3.2 might be implemented.

CHAPTER 10

LISP: VARIATIONS AND IMPLEMENTATIONS

Of all the hundreds of high-order programming languages that have come and gone over the last 30+ years, it is interesting to note that the first two to achieve any kind of prominence are still in use today. This pair of oldsters are ***FORTRAN*** (FORmula TRANslation) and ***LISP*** (LISt Processing). The former is the epitome of conventional Von Neumann computing; the latter is the first language to be based on lambda calculus. In contrast to the numerical calculation and production orientation of FORTRAN, LISP has been from its beginnings a language of choice for the highly experimental and complex needs of the advanced computer science community. As with abstract programming and the s-expression notation discussed earlier, this is due in no small measure to LISP's ability to describe itself, and extensions to itself, in itself. In fact, many of the language features assumed as basic parts of conventional languages today were first prototyped by experimental extensions to LISP that were found valuable to a larger community.

This flexibility proved so useful that major variations sprang up at almost every major research site where it was installed. Each of these solved some problem in the original LISP, added some new programming convenience, or introduced a new mode of computing. In turn, many of these variations spawned yet other variants with even more exotic capabilities. A partial list of these variants and their approximate year of introduction would include:

- The original LISP (late 1950s) at MIT
- ***LISP 1.5*** in the early 1960s as the first "standard"
- **Mac*LISP*** (late 1960s)—an MIT upgrade (see Pitman, 1983)
- **Inter*LISP*** (early 1970s)—a West Coast variant (see Teitelman, 1978)
- **Zeta*LISP*** and ***LISP Machine LISP*** (late 1970s)—commercial variants of MacLISP
- **SCHEME** (mid-1970s)—a major ***LISP*** variant much closer to lambda calculus (see Spring and Friedman, 1990)
- **Portable Standard *LISP*** (***PSL***) (early 1980s)—an efficient version of ***LISP*** from the University of Utah that was written largely in itself, and is easily transported to new computers (Griss, 1983)
- **Franz *LISP*** (early 1980s)—another variant of MacLISP optimized to run in UNIX (AT&T) environments (see Wilensky, 1984)
- **Common *LISP*** (early 1980s)—a standardized combination of many of the previous variants (see Steele, 1984)
- **Multi*LISP*** (mid-1980s)—SCHEME with explicit support for parallelism (see Halstead, 1985, 1986)

This chapter addresses four of these, the original LISP for background, Common LISP for its portability and standardization, SCHEME for its highly functional semantics, and MultiLISP for its introduction of parallelism. The references listed above, plus Chapter 2 of Gabriel (1986), cover many of the others.

As a language, LISP has also spawned an important family of specialized computing machine architectures that have features to support it. This chapter covers the major variations, with frequent reference to our previous discussions on the SECD Machine and on memory management for list structures.

Finally, given that LISP is so different from conventional languages and that LISP machines have many exotic features, comparing different implementations is difficult. Thus, we will end with a short section describing some standard benchmarks for LISP systems and how they might be used.

For the reader interested in trying his or her hand at actually implementing a LISP system, we recommend Allen (1978), Henderson (1980), and Henderson et al. (1983). The latter two include sources of actual listings of program code for both interpreters and compilers for a simple version.

10.1 THE ORIGINAL LISP

(McCarthy, 1965; Weissman, 1967)

LISP is by far the most popular function-based language in use today. Its originator, John McCarthy, designed the language with lambda calculus firmly in mind. The basic computational model is that of expressions built by applying functions to other expressions. Functions are themselves expressible in the same format as any other data objects, namely, s-expressions. Further, functions can be passed to, and returned by, other functions.

LISP also includes many extensions that increase its usefulness, such as a very extensive set of basic data types, ***polymorphic*** built-in instructions that accept many of these types in mixed combinations without programmer intervention, a mix of applicative- and normal-order evaluation that tends to maximize efficiency in a relatively safe manner, a collection of special forms, and the ability to extend the basic language interpreter easily. Modern variants of the language also permit the use of compilers that will produce fast code.

For a variety of reasons, LISP—in both its original and many of its current forms—is not a pure functional language. It has many conventional von Neumann features that if used improperly can result in all the problems of conventional programs. Perhaps the most significant of these is the treatment of identifiers. Scoping rules are somewhat different from the relatively static ones described earlier. However, more significant from a perspective visible to the programmer, many identifiers are bound explicitly to locations in memory. Operations that are integral to the language permit different values to be bound to the same variable at different times through the equivalent of a classical store instruction. This means that the same identifier can take on a time-sensitive sequence of values. Finally, the same symbol can have many different values at the same time, each of which is accessed in a different way.

Functions in LISP are also not quite first-class citizens. While they can be passed as arguments, doing so requires somewhat special coding, and care must be taken to avoid strange funarg problems.

Expression evaluation in LISP is also not in total agreement with our previous functional models. Many LISP forms include variants where a sequence of expressions are evaluated in a very specific sequential fashion. All but the last of these are executed for their "side effects" of modifying memory.

In terms of interpretative implementations, however, LISP's typical model does resemble the ***read-eval-print*** loop discussed earlier. The language was designed to be highly interactive, with a programmer permitted to enter both new function definitions and expressions to be evaluated as he or she sees fit. Internally, this is often implemented by functions very close in form to the ***eval*** and ***apply*** functions described earlier. This in turn means that the language can be extended easily.

The following subsections describe many of these issues in more detail. The reader should remember, however, that the notation and semantics described here are for the classic original LISP 1.5. Later sections in this chapter address modern versions that have significant variations.

10.1.1 Major Language Components

An ***atomic data type*** is a type which has no internal structure that is accessible to the programmer. A ***structured data type*** is a data type that does have internal components that are accessible to the programmer. Our prior s-expression-based language had only one such atomic data type—integers—and one structured data structure—lists (with minor variations for closures and recipes). Identifiers receive values exactly once, when they are defined.

LISP, in contrast, has a variety of atomic data types, including integers, floating-point numbers, literals, character strings, booleans, and big-nums. A ***literal*** is a character-string name of an object which has no value other than the name (e.g., *red, white, Monday,...*). Literals with different names are different objects. A ***big-num*** is a representation that permits numbers of arbitrarily long precision to be represented. An internal string of bits of indefinite length supports this representation. The other data types are as found in conventional languages.

Any object of one of these types is called an ***atomic symbol*** or ***atom,*** and responds T to a predicate atom.

Most of the built-in functions will accept arguments of a mix of atomic types and automatically convert to an appropriately common type. This is usually implemented by some sort of multiway branch (in the underlying microcode or interpreter code) on the basis of the tag bits of the two operands. Separate code at each target handles that specific combination of argument data types. Thus (ADD x y) will add anything bound to the x and y at runtime that evaluates to any kind of number. Automatic data type conversion is employed as needed.

The major data structures supported by LISP are s-expressions. However, many LISPs also include an ***array*** as a data structure. Speed of access to individual elements is similar to that of arrays in conventional languages. With arrays, it is not necessary to chase down a long list of pointers to gain access to the ith element. A typical implementation, however, is somewhat unusual. An array is defined by passing the desired name and dimensions to a built-in function, which in turn binds a block of storage and the start of a machine-language routine to the identifier name. Then, when the identifier is used in the function position of an expression, the machine-language routine is invoked, and arguments from the expression are used as the subscripts.

Programs as input by the programmer are expressed as lists, with functions in the car position and arguments in the rest. As with our previous s-expression functional languages, special keywords in this car position represent both built-in operators and designators for ***special forms.***

These programs may be compiled in several fashions, with varying degrees of visibility at execution time into their actual representation in memory.

10.1.2 Variable Scopes and the Property List

LISP's method of binding values to variables is a mix of that from functional and von Neumann computing. When using an interpreter to execute a LISP program, formal arguments for a lambda function are bound to values in essentially the same way as our earlier s-expression interpreters bound them, namely, on an ***association list*** or ***a-list*** of (⟨***name***⟩.⟨***value***⟩) pairs. Accessing a variable in the body of a function causes this association list to be searched. The topmost entry of the same name is the desired one; entries further down of the same name are said to be ***shadowed*** by it and are not visible until the context containing the topmost pair is removed.

Lambda variables in compiled code are also handled much as with our SECD compiler. A stack is maintained for lambda arguments, and references to such identifiers are compiled into offsets into this stack. This is similar to the SECD environment, except that storage is sequential rather than in lists.

Such lambda arguments are ***statically scoped***; that is, the expressions that give them values can be determined by a review of the static program code. Although they are the only kind of variables used in our abstract program models, they are not the only type of variables in a LISP system. A LISP programmer can identify certain identifiers to the system as ***dynamically scoped***; that is, the expressions that bind values to them cannot be determined until runtime. A single such variable can take on many different values through the execution of a program, with the value changing at the whim of the programmer. Further, not only can the value change, but at any one time there may be many different "values" associated with a variable at the same time. Each such value is used for a different purpose and is accessed or changed in a different way.

Such variables themselves come in two forms. First are the ***global variables*** or ***special variables,*** which are visible to all expressions whenever they are executed in the program (unless they are "masked" by a local variable of the same name). Second are ***program variables,*** which are dynamically allocated at the beginning of execution of certain special forms and then act like global variables until that special form completes. Again, these variables can shadow or be shadowed by other identifiers of any type.

There are at least two standard ways of implementing binding for such variables. First is via an association list as described above (often called ***deep binding***). This results in relatively slow access (one must traverse link by link through a list of lists). However, it does provide relatively quick time to unbind when a piece of code that defined such a variable completes, and any variables of the same name that were shadowed now become prominent. Usually, a simple CDR is all that is necessary.

Also, because of the dynamic nature of such bindings, this kind of support often requires more complex data structures than simple stacks, particularly when such variables are referenced in closures of unknown lifetimes. Heap storage, cactus stacks, etc., have all been suggested as remedies.

The second approach to handling such variables within a LISP system is through associating with each of them some unique global storage which contains their names as character strings (their ***print names***) and a list called the ***property list*** or ***p-list*** for values. This property list is an ordinary list except that a variety of special LISP functions assume that it has a specific format, namely, a list of an even number of elements of the form

(...⟨***property-name***⟩ ⟨***property***⟩...)

The odd component in an even-odd pair is the ***property name,*** the following even element is its matching ***property value,*** and together they are called a ***property*** of the identifier. Special LISP operations allow the programmer to add new properties, remove existing properties, or even modify the value of an existing property. This latter capability is similar to an ***assignment*** in a conventional language. Whatever value was there before a modification is lost, and whoever references that property in the future will see the new value.

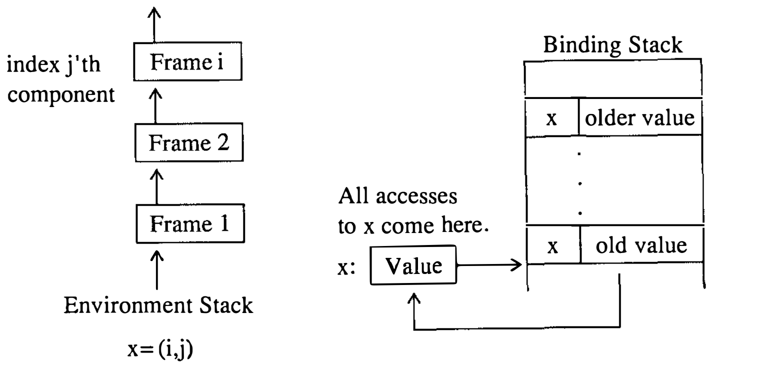
Two of the more common properties typically found on a property list are a global value for the identifier when used as a variable in the argument position of an expression [as in (ADD *x* 3)] and code to associate with that identifier when that identifier is in the function position of an expression [as in (*x* 3)]. The former is often called the ***APVAL*** property, the latter its ***FEXPR*** property. There may actually be several forms of the latter type, namely, code to be used by an interpreter, compiled code, a pointer to an internal subroutine, or even a macro definition to be used at interpret/execute time to identify a new kind of special form.

Note that this really does mean that the same symbol used in two different places of an expression can represent two radically different values. The expression (*x x*), for example, would bind the current contents of the FEXPR property with the first *x*, and the contents of the APVAL property with the second.

The advantage of using such a data structure for special and program variables is speed of access. Very often the APVAL and FEXPR properties are in fixed offsets from the print name, meaning that a compiler can generate a complete address at compile time for such variables. Accessing such a variable is then a simple LOAD or STORE, with no time-consuming searches.

This technique of associating a specific memory location with the value of a variable is called ***shallow binding,*** and while it provides fast accesses, it is not without problems of its own. Shadowing such a variable becomes difficult (one must save the old value in the value cell before creating the shadowing value and then restore it at the proper time). The ***funarg problem*** also becomes more complex.

Both shallow and deep binding have been used in real implementations. Figure 10-1 gives a summary of each. Which is faster overall depends to some extent on how frequent functions are called relative to variable lookup and the optimization capabilities of the compilers. In the early days of LISP, when interpreters were common, shallow binding tended to be preferred. Today, with 20 years experience with optimizing compilers for conventional languages using stacks as a key implementation structure, deep binding predominates.



(*a*) Deep binding.

(*b*) Shallow binding.

Operation	Deep Binding	Shallow Binding
Function Call:	Allocate New Frame	Copy Values to Stack
Variable Reference:	Double Index	Address Cell Directly
Function Return:	Reset Stack Pointer	Reload Values from Stack

FIGURE 10-1
Shallow versus deep binding.

The property list is often used as more than just a place for global variable values. A common programmer view of a property list is as a single object with multiple characteristics. Each characteristic has a different name and a different value. Thus, for example, one could have an object named *BOX11* and keep on its property list its *weight, color, shape, size,* and so on as follows:

BOX11: (PRINT-NAME ‘BOX11 APVAL 11 SIZE (2 3 4) WEIGHT 12 COLOR RED...)

In a sense this resembles at a primitive stage what today we would call a ***record*** or ***composite data structure*** with multiple fields. The major difference is that there is little innate support for defining such objects explicitly, nor in handling them as anything other than global variables. The programmer can, however, dynamically generate and retract properties from a symbol.

The operations to manipulate such property lists will be defined in Section 10.1.4.

10.1.3 Built-in Pure Functions

LISP has a rich variety of built-in functions—see Figure 10-2. For the most part the evaluation process for expressions with these in the function position is as with our previous s-expression interpreters. The argument expressions are evaluated first (***applicative-order evaluation***), and the resulting values are passed to some underlying machine code for the specific function. The result returned from that machine code is the value of the expression.

One major extension from the prior interpreter is in the handling of different types. Many of the built-ins accept a mix of atomic types for their arguments, test for them at execution, and perform automatically any data type conversions needed. The arithmetic instructions are typical of this.

Another major difference is that many of these built-ins accept a variable number of arguments. ADD, for example, will add up all the arguments it is given, regardless of the number. LIST is another such function, which will generate a list from all the arguments given it, whether that is 1 or 100.

Finally, the QUOTE function simply returns its argument as an unevaluated s-expression. Thus (QUOTE (PLUS 3 4)) returns (PLUS 3 4), not 7.

There are also functions (SPECIAL and UNSPECIAL) which control whether or not identifiers are to be considered special variables and are to have memory cells bound to them.

General Form: (<builtin-name> <expression>*)

Arithmetic
(ADD1 E) = Add 1 to number E
(PLUS E1 ... En) = Add up E1 through En
(TIMES E1 ... En) = Multiply E1 through En
(DIFFERENCE E1 E2) = Subtract E2 from E1
(MAX E1 ... En) = Find maximum of E1 through En
(MINUS E) = minus E

List Processing
(CAR E) = car of non-atomic E
(CDR E) = cdr of non-atomic E
(CONS E1 E2) = cons of arguments
(LIST E1 ... En) = n element list
(APPEND E1 E2) = Append top level elements of two lists

Predicates
(ATOM E) = T if E is an atom, nil otherwise.
(EQ E1 E2) = T if two arguments are same atomic value, or are same list in memory (at same address).
(EQUAL E1 E2) = T if two arguments are idential s-expressions.
(NUMBERP E) = T if E is a number.
(FIXP E) = T if E is a fixed point number.
(FLOATP E) = T if E is a floating point number.
(ZEROP E) = T if E is any type of number with a 0 value.
(GREATERP E1 E2) = T if E1>E2
(LESSP E1 E2) = T if E1<E2
(NULL E) = T if E is nil
(MEMBER E1 E2) = T if E1 is a top level element of list E2

Other:
(QUOTE E) = E unmodified
(SPECIAL (V1 ... Vn)) = Make V1 ... Vn special variables
(UNSPECIAL (V1 ... Vn)) = Release associated storage

Note: If not specified, above functions return nil.

FIGURE 10-2
Some basic LISP built-ins.

10.1.4 Modifying Values

The functions of Figure 10-2 are pure functions. They have no side effects. In contrast, the LISP functions of Figure 10-3 are used primarily for the modifications they make to the values of special variables.

There are two ways of classifying these functions: in terms of what kind of variable they modify, and whether or not they find the name of the variable by evaluating a function.

Functions which modify global variables have names that start with CSET. Those that modify program variables are simply SET.

The first argument for each of these functions specifies which variable to modify. Normally a programmer knows the variable's name directly. In such cases the function names end with Q. In other cases the variable's name must be computed; functions that evaluate their argument first do not end with Q.

Format: (<keyword> <address-expr> <value-expr>)

Operation:
1. Get symbol name from <address-expr>
2. Evaluate <value-expr>
3. Change value bound to symbol
4. Return value as result

Function Name	Kind of Symbol Modified	Is Address-Expression Evaluated First?
SET	Program Variable	Yes
SETQ	Program Variable	No
CSET	Global Variable	Yes
CSETQ	Global Variable	No

FIGURE 10-3
LISP functions with side effects.

As an example, consider the two expressions:

(SETQ *X* (PLUS 3 4))
(SET *X* (PLUS 3 4))

The first of these changes the value of the variable with the name *X* to 7; the second changes the variable whose name is bound to the value of *X*. This latter case would be typical when the name of a variable is passed as an argument to some function and is to be modified by it internally.

The expression (SET (QUOTE *X*) 7) is equivalent to (SETQ *X* 7).

Another common class of LISP functions (see Figure 10-4) performs decidedly nonfunctional operations on lists and other objects. The functions ***RPLACA*** (REPLACe cAr) and ***RPLACD*** (REPLACe cDr) have been mentioned earlier, and modify the contents of a cons cell. The function ***NCONC*** (Noncopying CONCatenate) produces an appended form of two lists. Unlike **append,** however, it does not copy the first list. Instead, it traces down that list to find the last cons cell (the one whose cdr is nil), and performs a **rplacd** on it by replacing the **cdr** field by the address of the first cell of the second argument.

Besides a fast, but destructive, append, this function also permits construction of ***circular lists***. Consider what happens as a result of (NCONC *X X*). The cdr of the last cell of the list *X* is replaced by a pointer to the first cell in *X*.

The above functions include no assumptions about exactly how or where a special variable's value is stored. As mentioned previously, this information is often on the property list associated with the symbol (often with a property name APVAL). Figure 10-5 lists a standard set of functions that permit direct modification of property lists. One argument is the symbol name whose property list is to be modified; other arguments specify the name of the property and the value. All such arguments are evaluated first, permitting any of them to be computed dynamically. As before, to avoid this evaluation one would surround the argument expression by (QUOTE...).

10.1.5 Special Forms

A ***special form*** is an expression whose function is a special keyword known to the system (like built-ins), but where the order of argument evaluation is somehow special. LISP has many more such forms than our s-expression languages of prior chapters. Figure 10-6 lists most of the more common ones. The following paragraphs address each.

(RPLACA X Y) = replace car of cell X by Y

(RPLACD X Y) = replace cdr of cell X by Y

(NCONC X Y) = replace last cdr of X by Y

Note: In all cases the first argument after modification is returned as result.

FIGURE 10-4
Cons cell-modifying LISP functions.

Terminology:
<proplist> = expression that evaluates to a symbol name with a property list.
<propname> = evaluates to a property name (a symbol itself).
<propvalue> = evaluates to a value.
<propfcn> = an expression.

Functions:
(GET <proplist> <propname>) = Retrieve associated value
(PUT <proplist> <propname> <propvalue>) = Set associated value
(PROP <proplist> <propname> <propfcn>) = If property name on specified list, return rest of list, else evaluate <propfcn>.
(REMPROP <proplist> <propname>) = Remove specified property

FIGURE 10-5
Manipulating the property list.

In Figure 10-6, a ⟨**body**⟩ is a replacement for a single expression in our earlier languages. It is a series of arbitrary LISP expressions. When executed, they are evaluated from left to right, with only the value from the rightmost expression returned as a value. They correspond to a series of program statements in a conventional programming language and are executed for their side effects (such as a SETQ on some variable). Thus, if one had the body

(SETQ *X* (PLUS *X* 1)) (PRINT *X*) (DIV *X X*)

the value returned would be 1, but when executed it would increment *X* by 1 and print out the result on the terminal.

(LAMBDA (<id>*) <body>) = a function expression

(DEFINE ((<id> <expr>)*)) = global function definitions

(LABEL <id> <expr>) = global recursive function

(AND <expr>$^+$) = evaluate expressions until first F

(OR <expr>$^+$) = evaluate expressions until first T

(COND (<testexpr> <body>)*) = nested conditionals

(PROG (<id>*) <progbody>) = sequential form execution

(GO <id>) = branch inside of PROG

(RETURN <expression>) = Exit <body>

where <body> := <expr>*
and <progbody> := (<expr> | <id>)*

FIGURE 10-6
Some typical special forms.

The special forms LAMBDA, DEFINE, and LABEL correspond almost directly to LAMBDA, LET, and LETREC of our earlier languages. The major differences are that LAMBDA permits a list of expressions for its body, DEFINE joins each symbol to its value in a separate list, and LABEL defines only one recursive function at a time.

The forms AND and OR are special in that they process their arguments one at a time, sequentially from left to right. The AND stops at the first F result, or the end of the list, whichever comes first. The OR is similar, stopping on the first T result.

The form ***COND*** is LISP's conditional expression. Each argument should be a two-element list of expressions. COND evaluates these arguments one by one from left to right. At each step, it evaluates the car of the argument. If that value is T, it evaluates the rest of the argument and returns the final value as the entire expression's value. If the car value is F, the next argument is processed.

The form PROG has no analog in abstract programming. Instead it is similar to a ***begin-end block*** in a conventional language. The **cadr** of the form is a list of identifiers to be considered special. Each of them is initialized to a value of nil. These variables are used as ***local variables*** to the body in the program, and can be read and modified at will. Any prior special variables of the same name are shadowed. At the end of the PROG the variables lose their bindings, and any shadowed bindings reappear as the topmost.

The ⟨***progbody***⟩ in a PROG is a series of LISP expressions like a ⟨***body***⟩ and resemble a series of statements that modify these local variables. As before, they are executed from left to right, with the value of the last one executed being the value of the PROG. Normally, these intermediate expressions will be executed to have some side effect such as modifying one of PROG's variables.

There are also several special formats that an expression in a ⟨***progbody***⟩ can take on. One is an expression that is a simple atomic identifier not in parentheses. This is called a ***label*** and represents a point in the program's code to which the programmer may later want to refer. It has no value and is simply skipped during execution.

The LISP statement that uses such labels is of the form (GO ⟨***label***⟩). When executed, the ⟨***progbody***⟩ containing the GO is searched for a matching label, and when one is found, interpretation restarts with the next expression after it. This corresponds directly to an ***unconditional branch*** or ***goto*** in a conventional language, and can be used to build any of the common language structures such as ***do loops, while loops,*** or any other kind of spaghetti code (cf. Figure 10-7).

A final way of modifying program flow inside a ⟨***progbody***⟩ is through a RETURN expression of the form (RETURN ⟨***expression***⟩). When executed, this form causes the ⟨***progbody***⟩ containing it to be immediately terminated, and the value returned that of the expression in the **cadr** position. All bindings to PROG local variables are discarded.

```
(PROG (x) (SETQ x 0)
        LOOP
        (SETQ x (PLUS x 1))
        (PRINT x)
        (COND ((EQ x y) (RETURN T))
                ((LESS x 10) (GO LOOP))
                (T F)))
```

FIGURE 10-7
Sample PROG expression.

10.1.6 Function Objects

One of the key functional-language capabilities of LISP is its ability to pass function definitions around as arguments. The major built-in that permits this is ***FUNCTION,*** which, when used in an expression such as (FUNCTION **F**), returns the expression **F** unmodified. The **F** should be either the name of some function or a lambda expression (LAMBDA...). It is virtually identical to QUOTE in operation but permits compilers and interpreters to recognize when an argument expression provides a function result.

Inside the body of a user-defined function, an argument can be used as a function simply by placing it in the first position of an expression when code is expected, as in the body for the following function **F3:**

(DEFINE (*F3* (LAMBDA (*F*) (*F* 3))))

To ensure proper operation, when *F3* is used in a function position, its argument should be defined at some point as a FUNCTION or equivalent.

Semantically, FUNCTION does nothing other than pass its argument unevaluated as a result. Consequently, if the expression is something like

(SETQ **F3** (FUNCTION (LAMBDA (*X*) (...*Y*...))))

there is a question of what is bound to *Y* when the object is executed as a function. If *Y* is a special variable, it gets its value from the appropriate global cell. If *Y* is not defined as special, the result often depends on the implementation, with interpreters looking up the name in the runtime a-list and compilers unable to define a consistent binding.

This is one area when more modern LISPs have made improvements. Figure 10-8 gives several functions included in LISP systems that accept functions so marked as arguments.

10.1.7 System Functions

LISP has visible to the programmer a variety of direct hooks into its internal operation. In particular, there are two functions callable by the

(FUNCTION <expr>) = returns <expr> marked as a function.

(MAP X F) = applies F to all cdrs of X, returning last cdr.

(MAPLIST X F) = returns list of F applied to cdrs of X.

(MAPCAR X F) = returns list of F applied to elements of list X.

(MAPC X F) = applies F to elements of X, but return last cdr.

Assume X = (1 2 3), F = (FUNCTION (LAMBDA (X) (PRINT X))):
 (MAP X F) prints (1 2 3), (2 3), (3). Returns nil
 (MAPLIST X F) prints (1 2 3), (2 3), (3). Returns ((1 2 3) (2 3) (3))
 (MAPCAR X F) prints 1, 2, 3. Returns (1 2 3).
 (MAPC X F) prints 1, 2, 3. Returns nil.

FIGURE 10-8
Function-processing functions.

programmer that permit pieces of dynamically generated code to be executed. These are EVAL and EVALQUOTE. They both accept one argument, which should be an s-expression representing an expression, and return the result of evaluating that expression. The difference is that EVAL has its argument evaluated before it evaluates it, and EVALQUOTE does not. EVAL thus does a double evaluation of the expression.

As might be expected, EVAL and EVALQUOTE are key components in an interpreter-based implementation of LISP where expressions are presented to the system, evaluated, and the answer returned to the user. Each time an expression needs to be evaluated, the interpreter crawls through the expression token by token. In some cases, such as for DEFINE, the execution of the expression results in information being stored with the identifier which says that when it is used as a function, this is the code that should be invoked. Often this code is as a special property on the property list for that identifier.

A ***LISP compiler*** could be invoked similarly. One could write (COMPILE ((⟨*id*⟩ ⟨*expr*⟩)*), which would be interpreted to cause the expressions for each identifier to be compiled into basic machine code, and bound (again on the property list) to the identifier as an FEXPR property. Now whenever such an identifier is found in the function position of an application, this compiled code could be invoked, with the evaluated arguments passed to it.

Finally, consider how to expand a compiler to handle user-defined special forms, for example, one with a variable number of arguments. An interpreter can handle this without difficulty because it has the original text and can process it as it sees fit. The machine language from a compiler, however, does not have such capabilities. It should be optimized for the specific expression that it is presented. The trick in many LISP systems for doing this is via ***macros***. These are functions which, when they appear as a function in an expression passed to the compiler, are executed by the compiler before code is compiled. Arguments to the macros are the unevaluated s-expressions that are its arguments in the original expression. The s-expression resulting from this is passed back to the compiler for compilation.

As an example, most real computers can add only two numbers at a time, while our LISP definition permits expressions of the form

(ADD $\mathbf{E}_1 \ldots \mathbf{E}_{10}$)

The compiler needs to be told how to handle this specific case.

If ADD has been defined as a macro to the system (in a form like DEFINE, namely (MACRO (ADD...))), then when the above statement is found by the compiler, the ADD macro is executed, resulting in an s-expression of the form

(ADD2 $\mathbf{E}_1$ (ADD2 $\mathbf{E}_2$ (ADD2...(ADD2 $\mathbf{E}_9$ $\mathbf{E}_{10}$))))))))

where ADD2 is a form that adds exactly two numbers.

Assuming that the compiler knows how to generate code for ADD2, it could then generate code for this specific expression.

10.2 SCHEME—A PURE LISP

(Steele and Sussman, 1978; Abelson et al., 1986; Dybvig, 1987; Clinger, 1988; Spring and Friedman, 1990)

SCHEME is an example of a LISP variant taken "back to basics" and then combined with modern concepts about expressing and managing computation. It is a LISP in which functions are truly first-class citizens, identifiers are statically scoped to avoid many of the problems with special and program variables in ordinary LISPs, tail-recursive expressions are truly optimizable, and a great deal of attention has been given to a clean continuation mechanism. Its implementation can be focused on a small core of forms, with extensions to the language added cleanly and simply.

The following subsections describe many of these unique features. They do not, however, even begin to describe the full language; this is left to any of the excellent manuals on SCHEME.

10.2.1 Basic Syntax

SCHEME has a basic set of s-expression forms that represent the core of the language (Figure 10-9). An elegant mechanism described in the next section permits arbitrary expansion of the set of forms actually available to the programmer.

The first four of these basic forms map directly into the basic lambda calculus forms discussed in earlier chapters, with three simple extensions. First, the body of a lambda expression can consist of a series

```
<basic-form> := <constant>
             | <identifier>
             | (LAMBDA <arg-list> <body>)
             | (<basic-form> <basic-form>*)
             | (QUOTE <expression>)
             | (IF <basic-form> <basic-form> <basic-form>)
             | (BEGIN <body>)
             | (SET! <identifier> <basic-form>)

<body> := <basic-form>+

<arg-list> := (<identifier>*)
           | <identifier>
           | (<identifier>+.<identifier>)
```

FIGURE 10-9
Basic SCHEME forms.

of expressions and, as with the original LISP, when executed this series is executed from left to right, one at a time, with the value of the last expression returned as the value of the lambda expression.

Second, multiple arguments are permitted in an application expression, and there is no defined order for their evaluation. Any order, including parallel, is permissible. Note that in SCHEME the function expression is evaluated in a manner identical to that for the arguments.

Finally, the specification of arguments in a LAMBDA expression has three variants. The first is simply a list of the 0 or more identifiers, and is what we have used in the past. The second variant is an argument description that consists of a single identifier, without surrounding "( )." When such a function is executed, all the actual arguments present in the expression are evaluated, formed into a single list, and then bound to the single identifier. This is quite useful when the number of arguments to a function can vary dynamically. For example, the function **list** could be defined trivially in this fashion by (LAMBDA *x x*).

The last argument list form requires n ≥ 2 identifiers, and has a "." between the last two of these. This is again used for handling functions with an arbitrary number of actual arguments, but where there are always at least n of them. The first n−1 actual arguments are bound to the first n−1 identifiers; the rest of the actual arguments are collected into a list and bound to the last identifier. Thus, for example, in

((LAMBDA (*x y . z*) ⟨***expr***⟩) 1 2 3 4)

inside ⟨***expr***⟩ *x* has the value 1, *y* has the value 2, and *z* has the value (3 4).

The final basic forms of Figure 10-9 are also similar to things we have seen before. The IF form is exactly as defined for the s-expression form of abstract programs. The BEGIN form begins a list of one or more expressions, all of which are to be executed in a left-to-right sequential fashion, with the value of the last expression returned as the result of the BEGIN.

Finally, the SET! form resembles the SETQ form of LISP. The second argument is an identifier (which has already been created somewhere) whose value is to be changed to the value returned by evaluating the third element of the SET! expression.

10.2.2 Extended Syntax

The basic evaluation mechanism of SCHEME understands only the forms of Figure 10-9, the core evaluation functions, and whatever built-in functions are provided by the particular implementation. To extend the spectrum of special forms seen by the programmer, SCHEME employs a very elegant mechanism that is an outgrowth of the ***macro*** mechanism of the original LISP, but that is more fully integrated into both compilation and interpretation.

This mechanism has two parts, one that records how a new ***special form*** should be translated into an expression involving currently defined forms and a function that does the actual translation into something the standard **eval** can handle.

The latter is called the ***system syntax expander*** and consists of a database of ***pattern/expansion pairs*** and a function to search and use them. This function, ***expand,*** takes a single unevaluated expression as an argument and looks recursively at any keywords found in the function position of the expression itself or any nested subexpressions. If this keyword matches any of those in the database, a match against the patterns is performed, and the expression is rewritten according to the selected expansion half. The result of this expansion is returned as the value of the original expression.

The normal place at which **expand** would be called would be in an interpreter or compiler for the basic forms, and it would be invoked after discovering that a particular expression matches none of the basic forms. After a successful expansion, the interpreter/compiler would be recalled on the expanded expression. Note that such a recursive call would expand any special forms found in the previously expanded expression, even if they were the same form that was just expanded.

With this capability, a programmer can design a new special form and enter it into the database by using a call to ***EXPAND-SYNTAX*** as pictured in Figure 10-10. The syntax of an EXPAND-SYNTAX expression consists of two or more arguments. The first is a list of keywords, the first of which is the keyword (⟨car-name⟩) whose appearance in a car of an expression would trigger an attempt to find a matching pattern/expansion pair out of the remaining arguments to the EXPAND-SYNTAX expression. The remaining keywords (⟨key⟩*)in this first argument list are identifiers which, if encountered in an expression being expanded, should be left alone.

```
<extend-syntax form> : = (EXPAND-SYNTAX
                              (<car-name> <key>*)
                              (<pattern> <expansion>)*))

(EXPAND-SYNTAX
    (LET)
    ((LET ((id val) ...) e1 e2 ...)
     ((LAMBDA (id ...) e1 e2 ...) val ...)))

(EXPAND (LET
            ((w 4) (x 1) (y 2))
            (PRINT w)
            (PRINT x)
            (+ x (LET ((z 3)) (× z y))))

→ ((LAMBDA (w x y)
            (PRINT w)
            (PRINT x)
            (+ x (EXPAND (LET ((z 3)) (× z y)))))
   4 1 2)

→ ((LAMBDA (x y)
            (PRINT w)
            (PRINT x)
            (+ x ((LAMBDA (z) (× z y)) 3)))
   4 1 2)
```

FIGURE 10-10
Sample pattern expansion syntax.

The rest of an EXPAND-SYNTAX expression consists of zero or more two-element lists. The car of each of these lists is a ***pattern expression*** to be compared against the expression being expanded. Nonkeyword identifiers in this pattern are ***pattern variables*** and are bound to matching elements of the expression being expanded. A "..." after some subpattern in a pattern means that the subpattern may be repeated as often as necessary, with new identifiers used in each case whose names are the identifiers from the subpattern, but subscripted.

The second element of each list is the ***expansion expression*** to be used on a successful pattern match. In this expansion, any nonkeyword identifier that also appears in the pattern is replaced by the value that is bound to it. Again a "..." means to repeat the prior subexpression, but with invisible subscripts on the identifiers.

The example of Figure 10-10 is a doubly nested LET expression, where LET is expanded exactly as was described for lambda calculus. In the outermost pattern matching, the pattern variables get bound as follows:

$id=w$
$id_1=x$
$id_2=y$
$val=4$
$val_1=1$
$val_2=2$
el=(PRINT w)
$e2$=(PRINT x)
$e2_1$=(+ x (LET ((z 3)) (× z y))))

Making these substitutions into the expansion pattern for LET expression yields the intermediate form of Figure 10-10. This also has a nested LET in it. A recursive call to EXPAND performs another pattern match with a different set of assignments, yielding the final result.

10.2.3 Bindings

As with the original LISP, identifiers in SCHEME also appear to come in two kinds, lambda arguments and others. All lambda arguments are statically scoped, that is, in expressions such as (LAMBDA ⟨***arg-list***⟩ ⟨***body***⟩), the values bound to the arguments defined in the ⟨***arg-list***⟩ are visible *only* to uses of them inside the ⟨***body***⟩. Once the lambda expression is complete, the bindings are discarded.

The other mechanism for defining variables is the ***DEFINE*** form:

(DEFINE ⟨***identifier***⟩ ⟨***expression***⟩)

When executed at the top level, such expressions define ***global variables*** that are accessible to all expressions. They may also be changed by SET! expressions.

Such expressions may also be used inside the body of a lambda definition, as in:

(LAMBDA ⟨***arg-list***⟩...(DEFINE ⟨***identifier***⟩ ⟨***expression***⟩)...)

When this LAMBDA is executed, storage is allocated for the identifiers from ⟨***arg-list***⟩ and values from the argument expressions that are bound to them. When the DEFINE form is executed, it augments the current environment defined by the LAMBDA to include space for the new variable, and initializes it to the value from the accompanying expression. Then, for the rest of the execution of the lambda's body, any references to that variable will be captured by this binding. At return from the lambda, the binding is discarded along with the other argument bindings. It is just as if an extra argument has been added to the LAMBDA, with a value bound to it.

Finally, one of the problems with global variables in LISP (and many other conventional languages) is that it is difficult to maintain consistency, especially when a group of functions are to share variables, and these functions are to be called in several different environments, potentially at the same time. The approach taken in SCHEME is to define a special globally accessible ***fluid environment*** for ***fluid variables,*** and to use a ***FLUID-LET*** expression surrounding expressions that will reference then. This would have the form:

(FLUID-LET (((⟨*identifier*⟩ ⟨*expression*⟩)*) ⟨*body*⟩)

Any function executed in this body, even one whose definition is not within the FLUID-LET's scope, can refer to a fluid variable and get the current value.

In execution, a FLUID-LET will save all current fluid bindings to the variables defined in its first argument and develop new bindings with values as specified. Inside any expression called by the body, a (FLUID ⟨***identifier***⟩) expression will return the current binding in this fluid environment. When all the expressions of the body are complete, these fluid bindings are reset to what they were at entry to the FLUID-LET. A SET-FLUID! form is available for modifying such a value.

Note also that, unlike the original LISP (or Common LISP), there is only one value assigned to an identifier; there are no separate nonfunction and function values. Thus, whenever an identifier is used in a function position of an expression, its value—whatever it is—is used for the function definition. Also unlike the other LISPs, a SET! can be used to modify any identifier which scopes the expression.

10.2.4 Suspending Evaluation

(Haynes et al., 1986)

SCHEME's clean treatment of function objects permits inclusion in the language of three different mechanisms for suspending and resuming evaluation of an expression: ***delay*** and ***force*** functions for supporting futures, an enhanced ***continuation*** implementation that permits sophisticated co-routining, and an ***engine*** mechanism that permits specification and control of multitasking of individual SCHEME evaluations.

The most basic of these mechanisms is an implementation of special forms (DELAY ⟨***expression***⟩) and (FORCE ⟨***expression***⟩) which act exactly as we discussed in Chapter 9. Figure 10-11 diagrams a sample implementation. An expression of the form (DELAY E) creates a function object (of no arguments) that has as its body the expression to be delayed, and binds the function to a local variable *future*. It then returns as its value:

```
(EXTEND-SYNTAX (DELAY)
    ((DELAY e)
     (LET ((future (LAMBDA ( ) e)))
          (LAMBDA ( ) (LET ((v (future)))
                           (SET! future (LAMBDA ( ) v))
                           v)))))

(DEFINE FORCE (LAMBDA (future) (future)))
```

FIGURE 10-11
Special forms for delayed evaluation.

(LAMBDA ( ) (LET ((*v* (*future*))) (SET! *future* (LAMBDA ( ) *v*)) *v*)))))

where *future* is bound in a closurelike environment to (LAMBDA ( ) E).

Now the first time FORCE is applied to this object, the value of the variable *future* is evaluated [it is in the function position of (*future*)] and bound to *v* by the let. This value is then encapsulated into (LAMBDA ( ) *v*), bound (via the SET!) back into the variable *future,* and returned as the value of the FORCE. The next time the expression is FORCEd, the process is repeated, but now evaluating *future* returns the evaluated form of the original expression, without more evaluations.

SCHEME also includes direct support for ***continuations*** almost exactly as described in Chapter 9. The form is (CALL/CC ⟨***function***⟩), where CALL/CC stands for "call with current continuation." This expression packages the computation up to the current point into a function of one argument (called a ***first-class continuation*** in SCHEME), and passes it as an argument to the function given as the argument to CALL/CC. This latter function (which should be of one argument) is then executed. Inside this execution, if its argument is ever used as a function in an application involving one argument, the argument expression is evaluated, and then the rest of the current expression evaluation is discarded. The continuation is restarted at the evaluation of (CALL/CC...), with the value returned by the (CALL/CC...) being the value passed to the continuation by the now-defunct expression inside the (CALL/CC...). What is different from Chapter 9, however, is SCHEME's ability to save this continuation in a global or fluid variable, and then to recall it any number of times from other programs.

The final SCHEME mechanism for managing computation is an ***engine*** (see Figure 10-12). This is a device that contains a continuation for some expression which, when activated, runs that continuation for some specified period of time. The programmer has total control over how long, and what to do if the computation does not complete in that time. This permits elegant multiprogramming executives to be built directly in SCHEME without resorting to any lower-level programming languages.

A function ***MAKE-ENGINE*** takes an expression, treats it as a function of no arguments (essentially delays it), and packages the result in a

(MAKE-ENGINE <expr>) → <engine>
where <engine> = (LAMBDA (ticks complete expire) ...) and

- Ticks = amount of "time" engine can run.
- Complete = function of two arguments to call if computation completes (arguments expected: time remaining and final value).
- Expire = function of one argument to call if time runs out (argument expected: a new engine).

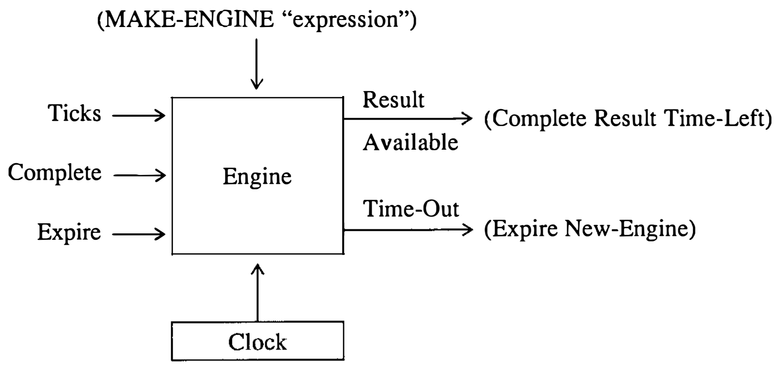


FIGURE 10-12
Engines in SCHEME.

continuationlike object that accepts three arguments: the amount of time to run, a function to call upon successful completion before time runs out, and a function to call if time runs out first.

After activating an engine by providing argument values, the packaged computation begins to run. If the computation completes in that time, the second argument to the engine is used as a function in an application which includes the result and the time remaining as arguments. The result of this call is the result of the engine.

If time runs out before the computation completes, the third argument to the engine's activation is called as a function with a single argument that consists of a continuation for the rest of the computation. This permits the called function to restart, reschedule, or otherwise handle the interrupted computation.

10.2.5 Implementations

SCHEME was designed from the outset with efficiency of implementation in mind. Identification of a small core language and an integrated way of extending the language from that core means that optimization techniques can focus on a relatively small set of forms. Careful consideration of closures and static scoping means that for the most part a compiler can determine at compile time exactly what binding is associated with most symbols in the body of an expression. This permits a compiler to know exactly when a set of bindings is no longer useful and can be deleted. In turn, this means that much of a function's environment can be kept in an easily managed stack of sequential cells rather than in a list of lists. Additionally, this also means that ***tail recursion*** can be implemented optimally by branches back to the beginning of a routine and by reusing the same set of memory cells for each set of arguments in the sequence of calls. No wasted growth in memory is needed for the recursions.

The literature contains several descriptions of SCHEME compilers that use these ideas to advantage. One of the first SCHEME compilers, named ***Rabbit*** (Steele, 1978), followed almost exactly the pyramid diagram of Figure 6-1. A good compiler for the core set of forms was written in MacLISP, which generated LISP code that could be compiled by the MacLISP compiler. This was then rewritten in SCHEME, with optimization features added to it. The entire compiler came to about 50 pages of code.

An example of a more recent implementation can be found in Bartley and Jensen, (1986). Here a similar core compiler was built, but the code generated was for an abstract machine with the following characteristics:

- A ***BIBOP*** approach to tagging, where different areas of memory are allocated to different kinds of objects
- A conventional stack to hold most environments, with a heap (actually multiple heaps) to save objects whose lifetimes extend beyond a function call
- A very ***RISC***-like (Reduced Instruction Set Computer) instruction set, with 64 general-purpose registers for arguments.
- An implementation of this architecture using ***threaded code*** techniques (see Kogge, 1983).

Arguments are passed in these general-purpose registers, with R1 containing the first argument, R2 the next, and so on. Embedded defines and lets use additional registers for their bindings. Saving of these registers is done by the caller of a function, just before the call. Together this minimizes memory traffic.

When targeted for a typical microprocessor, the Intel 80286, execution time of the abstract machine interpreter for such code was about two to six times faster than a pure interpreter, and only two to three times slower than compilers that generated (lots of) native 80286 code.

10.3 COMMON LISP—A MODERN STANDARD
(Steele, 1984)

The multiplicity of LISP dialects was a healthy aspect of the first quarter-century of LISP development. Many new ideas about expressive power, implementation techniques, and program development in a highly interactive invironment were developed and tested. This diversity was fine as long as the primary users were academics. However, by the early 1980s it had become apparent that LISP was outgrowing its roots, and that an "industrial-strength" LISP was needed. ***Common LISP*** is the result.

Common LISP is a strong descendant of MacLISP and ZetaLISP, with a flavoring of features from SCHEME and InterLISP. The goals for its design included commonality and compatibility with these earlier LISPs, with features chosen for their expressiveness and consistency, and the ability to build implementations that are both portable and easily made consistent with both compilers and interpreters. The ability to import powerful software development tools and a feature set big enough to obviate the profusion of dialects was also important.

Common LISP has caught on in a big way. An ANSI standards committee is deliberating, specifying both a core language and a verification procedure to determine compatibility. Most major computer vendors have LISP products and tool sets compatible with the standard.

Common LISP is not a small language; Steele's classic 450+-page reference manual lists approximately 600 functions available to the programmer, with many functions having large numbers of options that can be specified in a variety of ways. Given this, we will not even attempt to introduce the whole language here. Instead the emphasis will be on those unique features that diverge most from previous discussions or that are of particular importance for implementation.

10.3.1 Data Types and Structures

Common LISP includes seven basic types of numbers, with at least seven more types of complex numbers. There are two types of characters, with each character having variations in font and other attributes. Bits, symbols, ***random states, unreadable data objects,*** and ***common*** round out the atomic types.

Data structures include not only cons-based lists, but arrays, vectors, and strings of consecutive locations, streams, hash tables, packages of code and data, file names, structures, and functions. For compatibility with older LISPs, all symbols have ***property lists,*** with property names and values arranged in even-odd pairs. New types can be defined in terms of old types by expressions that give both the old type and one or more predicates on that type which determine when objects from the supertype are elements of the new type.

Structures are similar to records in Pascal and are defined by a form:

(defstruct ⟨***structure-name***⟩ ⟨***slot-name***⟩*)

where a ***slot*** is equivalent to a Pascal field.

The ***defstruct*** function uses these names to automatically construct a set of functions that permit programmer access. An expression of the form (make-⟨***structure-name***⟩ {:⟨***slot-name***⟩ ⟨***expression***⟩}*) creates a new object with the matching internal structure and returns a pointer to that object. An expression of the form (⟨***structure-name***⟩-⟨***slot-name***⟩ ⟨***expression***⟩) will, if the ⟨***expression***⟩ evaluates to an object of type ⟨***structure-name***⟩, return the field with the name ⟨***slot-name***⟩.

All these types are arranged in a ***partial ordering*** or ***class hierarchy*** of subtypes and supertypes that all tier up to a single universal supertype ***t*** and tier down to a common subtype ***nil.*** This permits a very clear description of the limits of built-in functions and what happens when "incompatible types" are combined. It also forms the basis for a powerful ***object-oriented*** language extension to be developed.

10.3.2 Forms

Common LISP has both a variety of special forms and some novel ways of personalizing them. An example of the latter is the ability to include ***keyword arguments*** that personalize the function in forms involving many built-in functions. For example, in most LISPs there are a variety of **member** functions to test if an argument is a member of some list. The variations come from the type of equality test used (same location versus same value versus same terminal leaves...). In Common LISP, a single function **member** covers all of these, with an extra argument pair of the form ":test #'⟨***name-of-equality-test***⟩." The ***keyword*** ":test" signals that the next argument selects the function to use for the test.

Many Common LISP functions accept keywords of all sorts to select options for particular expressions. In general, if the programmer does not specify a keyword value pair for some function that accepts it, a system-provided default will be used.

Another set of special forms permits a function to compute several different results and pass them as a set, ***without*** having to explicitly cons them together and then have the programmer remember which obscure sequence of cars and cdrs will take the parts out again. Figure 10-13 gives a feeling for what such forms look like.

(MULTIPLE-VALUE-LIST <form>*)
- Collect all values returned from <form>s into a list.

(MULTIPLE-VALUE-CALL <function> <form>*)
- Collect all values resulting from using <form>'s as arguments to <function>.

(MULTIPLE-VALUE-BIND (<symbol>*) <value-form> <form>*)
- Multiple values from <value-form> bound to <symbols>'s.
- Resulting bindings good during evaluation of latter <form>'s.

(MULTIPLE-VALUE-SETQ (<symbol>*) <form>)
- Multiple values from <form> "SETQ" to variables.

FIGURE 10-13
Common LISP forms for multiple results.

Although Common LISP does not have any direct analogs to SCHEME's continuations, it does have support for some similar features. For example, when executed inside some sequence of code, an expression of the form

(*CATCH* ⟨*tag*⟩ ⟨*progbody*⟩)

will evaluate the ⟨*tag*⟩ expression and record it on a stack as a ***catcher*** along with the return address to the CATCH and the current state. It will then execute the ⟨*progbody*⟩ one expression at a time. If the last expression completes normally, its value is returned from the CATCH and the catcher entry is removed from the stack.

If, however, during the execution of the ⟨*progbody*⟩, any expression is executed of the form

(*THROW* ⟨*tag*⟩ ⟨*expression*⟩)

the ⟨*tag*⟩ is evaluated and the stack is searched for an object with a matching value. If such a match is found, the entire computation between the matching CATCH and the current point in the computation is aborted, and the value from the THROW's ⟨*expression*⟩ is returned as the value of the CATCH form.

10.3.3 Scope and Extent

Common LISP inherits from SCHEME static scoping of variables defined in lets and equivalents. A variable defined as an identifier in a let can only be referenced by program text inside the body of the let. The ***extent*** or lifetime of such a binding is only for the duration of time the body of the expression is being evaluated.

In addition to this, however, Common LISP also permits ***special variables*** whose bindings exist until the programmer terminates them, and which are accessible anywhere in the program text. This is similar to SCHEME's ***fluid binding***. A (DECLARE (SPECIAL ⟨*id*⟩)*) identifies that all future references within the current body to the specified identifiers are to be treated as references to special variables. Similarly, a (DEFVAR ⟨*id*⟩ ⟨*expression*⟩) creates a special variable and initializes it.

10.3.4 Streams

Common LISP has direct support for streams, particularly as used for I/O. In this context a ***stream*** is an object that can serve as a potentially never-ending source or sink for data. Streams may handle two types of data items, character or binary, and may be marked as input, output, or bidirectional. In most Common LISP applications, streams reference files in the computer's mass store.

There are several standard streams, including:

- *standard-input*
- *standard-output*
- *error-output*
- *query-io*
- *debug-io*
- *terminal-io*
- *trace-output*

There is a wide variety of functions to generate, test, accept input, and deliver output to streams. For example, a simple form to read the next character from a stream might look like

(READ-CHAR ⟨*input-stream*⟩ ⟨*eof-error-p*⟩ ⟨*eof-value*⟩)

where ⟨*input-stream*⟩ should evaluate to the name of a stream. Each time this form is executed, the next available character from the stream should be returned.

In this form the other arguments control special circumstances. ⟨*eof-error-p*⟩ evaluates to a boolean that signals how to handle the ***end-of-file*** indication from a stream. A T indicates that an error should be signaled if a READ-CHAR is executed against a stream that has run out of data. An F indicates that no error should be signaled, and instead the value from evaluating ⟨*eof-value*⟩ should be returned.

Output is similar.

10.3.5 Evaluation

Common LISP provides several functions to permit users to control evaluation of expressions. The most basic of these is (EVAL ⟨*form*⟩), which evaluates the argument assuming the current special environment and a nil static environment. A variation of this,

(EVAL-WHEN {compile | load | eval} ⟨*form*⟩)

specifies when the ⟨*form*⟩ should be evaluated. The keyword "compile" indicates that evaluation should proceed only if the EVAL-WHEN expression is encountered during a compilation. The keyword "load" indicates that the form should be evaluated whenever the current file being compiled is loaded; "eval" covers other cases.

EVAL references two special variables during its execution that permit user control over evaluation, such as for building debuggers. If the variable ***evalhook*** is found to be non-nil, its value is assumed to be a function of two arguments, a form and an environment. Instead of evaluating the form, EVAL passes this argument to this function along with the current environment. Whatever value is returned from the function is the value of the EVAL.

Similarly, the variable **applyhook** is tested just before evaluating an application, and if it is not nil, replaces the normal apply.

10.4 MULTILISP—A PARALLEL LISP

(Halstead, 1985, 1986)

As discussed before, pure lambda calculus provides opportunities for parallelism in both applicative-order and lazy evaluation. Although LISP has its roots in lambda calculus, none of the variants described so far were built with parallelism explicitly in mind.

MultiLISP is such a variant. It was designed as an extension of SCHEME that permits a programmer to specify simply opportunities for parallelism in his or her program, and then support such parallelism relatively efficiently on real hardware.

10.4.1 Functions for Parallelism

MultiLISP as a programming language consists of SCHEME plus several new forms (Figure 10-14). The simplest, using PCALL ***(Parallel CALL),*** has n+1 arguments, all of which the programmer is willing to have evaluated in any order whatsoever, including in parallel. When the evaluation on all of them is complete, they are recombined as a normal MultiLISP expression and evaluated. Then the first of these arguments becomes the function of the final expression, and the remaining n arguments become its arguments.

The next form is DELAY. This is very similar to the definition given in Chapter 9, with the one-time evaluation feature of a future included. The single expression that is its argument is packaged in a closurelike structure that can be passed around like any other object. The first time the value of this object is required, it is **forced**, and the resulting value is substituted for the closure. All future references see the evaluated value.

(PCALL F E_1 ... E_n)—permit parallel evaluation of F, E_1, ... E_n. Then evaluate (F E_1 ... E_n)

(DELAY E)—package E in a closure.

(TOUCH E)—do not return until E is evaluated.

(FUTURE E)—package E in a closure and permit eager beaver evaluation.

(REPLACE-xxx E_1 E_2)—replace the xxx component of E_1 by E_2 .

(REPLACE-xxx-EQ E_1 E_2 E_3)—replace xxx of E_1 by E_2 only if = E_3.

FIGURE 10-14
MultiLISP functions supporting parallelism.

The function TOUCH is similar to a **force** and allows a programmer to "touch" an object that might have been returned by a DELAY and guarantee that it has been evaluated. It is essentially an identity function which returns the value of its single argument, but with some synchronization built in. If the object is the result of a DELAY or a FUTURE, and some other processor is already at work on it, TOUCH will wait for the evaluation of that future to complete before continuing.

The form of FUTURE is identical to that of DELAY except that the resulting object can be evaluated by a spare processor at any time after its creation. As before, once it has been evaluated, such an object need never be reevaluated. Thus, liberally sprinkling FUTUREs in a program would leave behind a trail of closures that a flock of processors could evaluate in parallel.

At first glance it would seem that PCALL could be implemented by **FUTURE** as in:

(PCALL **F** $\mathbf{E}_1 \ldots \mathbf{E}_1$)=((TOUCH (FUTURE **F**)) (FUTURE $\mathbf{E}_1$)... (FUTURE $\mathbf{E}_1$))

This would permit all the arguments and the function to be evaluated in parallel as with PCALL. However, PCALL demands that all argument evaluations be done before evaluating the new expression. The FUTURE-based version above does not.

The final two forms supporting parallelism are REPLACE-xxx and REPLACE-xxx-EQ, where xxx is either CAR or CDR. These are essentially multiprocessor versions of RPLACA and RPLACD that permit controlled modification to storage. The first takes two arguments and, in an uninterruptible sequence of atomic actions, reads the cell designated by its first argument and replaces it by the value of the second. The value of the form is the original value read.

REPLACE-xxx-EQ is similar, except that it does the replacement only if the original value of the cell to be changed equals the value of its third argument. The value returned is an indication of whether or not the replacement took place. This is essentially the same as a ***COMPARE-AND-SWAP*** on many conventional computers, and is useful for atomically testing and setting ***semaphores*** and other objects used for synchronization of separate tasks.

10.4.2 An Implementation

The first implementation of MultiLISP was on the ***Concert*** machine at MIT—a 24-way Motorola 68000-based shared-memory multiprocessor. MultiLISP programs were compiled into an SECD-like abstract machine ISA called ***MCODE*** which was then interpreted by an interpreter written as 3000 lines of C. Each processor had a copy of this interpreter in its local memory. A common garbage-collected heap was distributed among all processor memories to hold all shared data (including the MCODE programs themselves).

MCODE programs manage data structures called ***tasks*** that are accessed by three pointers: a program pointer, stack pointer, and environment pointer.

The MultiLISP function FUTURE creates a new task and leaves it accessible for any free processor. The environment of the task is that of the parent expression at the time of creation.

The initial stack is a single object representing the ***future*** to be expanded. In turn this object contains a value (initially a closure), a flag (initially F) indicating the state of the object's evaluation, a ***task queue*** (initially empty) representing a list of tasks that wish this value after evaluation, and a ***lock*** to serialize access to this queue if several tasks attempt simultaneous access.

When a task wishes the value of a future, it checks the flag. A T indicates that the value is evaluated. An F indicates that no one has evaluated it, so the task changes the flag to "busy" and starts evaluation. A "busy" indicates that some other task is evaluating this value, so the task places itself on the associated task queue, and the processor executing it goes to find other work. When the evaluation is complete, all tasks on the task queue are added to a queue of runnable tasks.

Deciding which task to run next is done using an ***unfair scheduling policy*** to prevent an explosion of tasks that might otherwise choke the system (see also Keller and Lin, 1984). For example, in a PCALL expression, all of a processor's resources are devoted to evaluating one argument at a time, with the other arguments left as pending tasks on a queue where other free processors might pick them up. This prevents the situation where if each of the argument expressions itself expands into a PCALL, and if the processor stopped as soon as it hit a nested PCALL and picked up the next argument, then the queue of tasks could grow explosively, using up all storage to the point where there would be none left to do evaluations. (FUTURE E) is handled similarly. The future is evaluated first, with the task creating the future place on a queue.

Garbage collection is distributed across all processors. Each has its own set of ***semispaces*** (see Chapter 8) and employs a variant of Baker's algorithm to scavenge its area. To avoid problems with pointers from other memories, all processors synchronize their flip of fromspace and tospace to occur at the same time. Also, to avoid multiple processors trying to evacuate the same object to several different tospaces at the same time, each cell has a ***lock bit*** associated with it which guarantees that the first processor to scavenge and copy an object will delay all others until the operation is complete.

10.4.3 Experimental Data

Initial experiments with MultiLISP on Concert gave mixed results. Overall the performance of a 24-way machine was slow. This was due to the interpreted nature of the MCODE implementation, to the need to perform frequent bit field operations, and to bounds checking of stack pointers at each push. Hardware support for all these would greatly accelerate performance.

When looking just at relative data, however, speedup due to parallelism was achieved. A quicksort program employing futures for each half of a list to be sorted achieved speedups very nearly equal to the number of real processors made available. A tree insertion program, on the other hand, stopped at a speedup of about four to one after about six processors were made available.

A later version of MultiLISP, called ***Butterfly LISP,*** has been implemented for the ***Butterfly*** parallel processor (Steinberg et al., 1986; Allen et al., 1987). This machine employs up to 256 faster 68020s, with an interconnection network between them to permit shared memory access. On a 16-way version, significant speedup was demonstrated for both a matrix multiply routine and a ***Boyer-Moore theorem prover,*** which is a classical LISP benchmark (see Section 10.8).

10.5 THE CADR MACHINE

(Greenblatt et al., 1984)

The early 1970s saw interest develop in machine architectures to support LISP in hardware, particularly for very-high-performance personal workstations. Perhaps the first of these was the ***CONS machine*** designed and constructed at MIT in 1975. Although it was operational, there were a variety of architectural weaknesses discovered in its design, so in 1978 a new version, the ***CADR machine,*** replaced it at MIT. This machine was specifically designed to support the MacLISP dialect. It represented an important watershed in computer architecture for specific languages and drove many later designs, both academic and commercial. For this reason we describe it in some detail here.

Some of the specific characteristics of the CADR design included:

- Optimization for high-performance, single-user interaction
- LISP as the primary language for applications, interpreters, compilers, and operating system functions
- Data type checking support directly in the hardware
- Large memories (for the time)
- Built-in memory allocation and garbage collection

10.5.1 Basic Machine Organization

The basic machine organization (Figure 10-15) is representative of computer architecture of the time. For flexibility in building new functions into the machine, it was heavily microprogrammed, with provisions to dynamically modify the microcode. Individual microinstructions could be executed in 180 ns. The main memory was very large for its time (up to 4 million words), with a disk backup to give a much bigger virtual memory space.

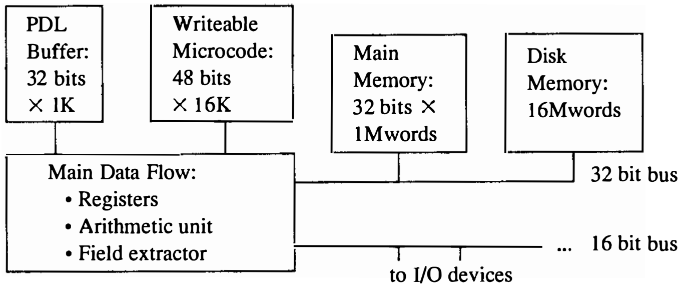


FIGURE 10-15
CADR Machine organization.

Major hardware support was included for a "push-down list" (or ***PDL buffer***), which would keep up to the top 1K entries of a stack very close to the main dataflow. This stack served many of the functions of the S, E, and D stacks in the SECD Machine. Invoking new functions involved constructing ***frames*** of control information on it that contained (in approximate order) the following information:

- A pointer backwards in the stack to the previous frame
- Cells for argument values
- Cells for local variables and constants
- Cells for special variable access

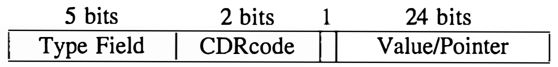
Operands for built-in functions invoked inside a function's body were push/popped from stack space in front of the current frame. These frames were kept in contiguous memory cells, with the topmost 1K cells in the PDL for rapid access. Over/under-flowing the PDL caused access from the rest of the PDL in memory.

10.5.2 Memory Structure

Memory in the CADR machine consists of a single 16-million-word virtual address space, with up to 1 million words of real memory. Thus, data and instruction addresses in their complete form take 24 bits, and are translated through a relatively conventional page table translation process (Baer, 1980, chap. 6).

The format of an individual memory cell (Figure 10-16) reflects SECD influences. Each cell is 32 bits long and contains a tag, a cdr code, and a value field. The diversity of the tag types and matching data fields (versus the three or four in our SECD Machine) reflects the real-world set of atomic data types supported directly by the hardware.

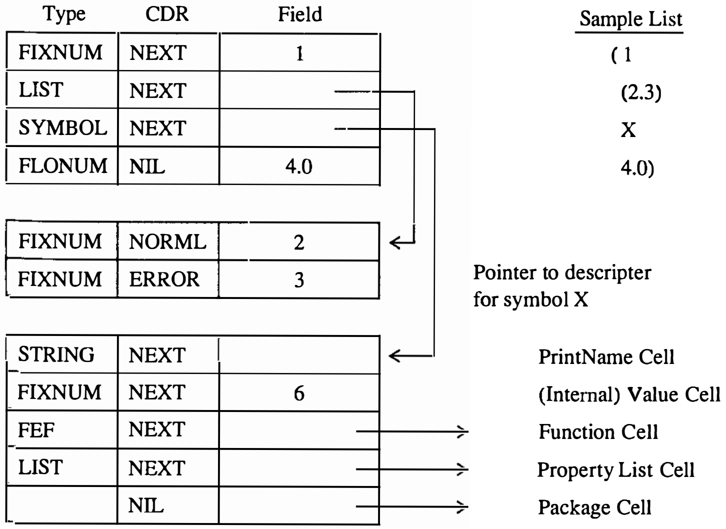
The representation of cons cells reflects many of the optimizations mentioned in Chapter 8. In particular, this includes ***cdr coding*** and ***car field packing***. Only the car field is always represented directly, and if the car value is a basic type, that value is stored directly in the value field. Figure 10-17 diagrams a simple example.



Typical Types:	Value/Pointer Field
SYMBOL	pointer to 5 word block
FIXNUM	24 bit integer
SMALL FLONUM	24 bit floating point number
EXTNUM	larger (multiword) numbers
STRING	pointer to a character string
LOCATIVE	"invisible" pointer to a value
LIST	pointer to 1/2 word cell (see CDRcode)
U-ENTRY	entry point for a microcoded function
FEF POINTER	start of code for macrocoded function
CLOSURE	pointer to a function and "closed variables"
...	

Note: pointers are 24 bit addresses to virtual memory words.

FIGURE 10-16
A CADR memory word.



Type	CDR	Field
FIXNUM	NEXT	1
LIST	NEXT	
SYMBOL	NEXT	
FLONUM	NIL	4.0
FIXNUM	NORML	2
FIXNUM	ERROR	3
STRING	NEXT	
FIXNUM	NEXT	6
FEF	NEXT	
LIST	NEXT	
	NIL	

FIGURE 10-17
Sample CDR coding and symbol object.

The value of the cdr field is controlled by the cdr code. There are four cases, exactly as discussed in Chapter 8:

- Normal—the value of the cdr field is found in the next sequential memory cell.

- Next—the cdr value is the address of the next memory cell, and thus need not be stored at all.
- Nil—the cdr field value is nil.
- Error—there is no cdr.

Different types of objects are kept in different ***areas*** of the memory, with different attributes provided by the user or the system. This is not for ***BIBOP***-like tagging but for simplifying garbage collection. Examples include stack space, array space, list space, symbol space, etc. These attributes reside in page tables in the virtual memory system maps, and help determine how to allocate space from them and what mechanisms (if any) should be used to recover them. For example, an area declared ***static*** is one where the programmer does not expect rapid change, and thus one the system garbage collector should avoid. The programmer assumes responsibility for its management. ***Garbage collection*** of unused memory in this machine is an incremental compacting variation of Baker's algorithm (discussed in Chapter 8). Each time a CONS instruction is executed, a few more words of storage are reclaimed and copied to another region of memory of the same type. As listed in Figure 10-16, one of the cell tags is a ***locative pointer.*** This is used to give a "forwarding address" to an object that has been scavenged out of one area of memory, but where not all references to it may have been fixed up yet.

10.5.3 Object Values

Values for arguments to a function are found in consecutive locations of the stack frame. Arguments for the function which called the current one are in the frame next below on the stack. Accessing is via indexing. To get the i-th element of the j-th function back in the calling sequence, one goes back j frames and then gets the i-th element. Conceptually this is identical to our SECD implementation, but it is much faster because the computation to get access to the i-th argument of a frame is a simple addition of i to the frame base.

Similar speedups are gained by using consecutive memory cells for dense regular structures such as arrays or character strings. The LISP functions for building such structures invoke microcode that allocates sufficient space out of an appropriate area of memory, as described above. In addition, LISP functions which access such objects can then tell what kind of object they are dealing with and can optimize their addressing of memory to locate it.

Storage for special and global variables is in its own area of memory and, as above, is optimized for speed. Each identifier has its own 5-word block of memory, as pictured in the bottom of Figure 10-17. The first of these is a pointer to the character string representing the variable's ***print name,*** i.e. the character string the programmer entered in the program. The next two words support a ***shallow binding*** mechanism for special and global variables that holds the current values associated with the identifier. As mentioned previously, many LISPs maintain separate values of an object for when it is used as an object and when it is used as a function. This shows up clearly in this storage allocation.

Following this is a pointer to the ***property list*** associated with the identifier. Finally, there is a pointer to the ***module*** that contains the definition of this variable.

The shallow binding of special variables causes ***funarg*** problems when functions with free variables are passed as arguments. Creating a closure requires "closing" each occurrence of a special variable inside the function's body. To facilitate this, MacLISP includes a special form for closure creation:

(CLOSURE ⟨***expression***⟩. ⟨***function-expression***⟩)

where the expression would evaluate to a list of identifiers to be closed. In CADR the resulting object would be a cell with tag ***closure*** and value equaling a pointer to the code and two pointers for each closed variable, one to a cell containing its value at the time the closure was created (its ***external value cell***) and one to its ***internal value cell*** as described in Figure 10-17.

When a closure is invoked, the current values of these variables are saved, and invisible pointers are installed in their value cells to point to the external values built at closure time. Completing the evaluation of the closure then includes restoring the value cells.

10.5.4 Program Forms

Programs for this machine can be represented in three forms. The topmost level is MacLISP, and is used for all applications and much of the system. At the most basic level, ***microcode*** provides most of the major system functions (such as virtual memory management and garbage collection), many of the built-in LISP functions, and an "interpreter" for the next level of programming. It was designed to permit a compiler (written in LISP) to convert a LISP function expression directly into microcode, permitting extremely high levels of performance.

The middle level is termed ***macrocode.*** It supports stacks much as the SECD Machine did, but in format it more resembles a conventional instruction set that does not store instructions as lists. These instructions are 16 bits long, packed two to a 32-bit word, formed into multiple-word sequences, and interpreted by a microprogram to carry out their functions. The general form of such instructions is

⟨operation⟩, ⟨operand⟩, ⟨destination⟩

where the operation is usually a very generic one (**add, cons,**...). The operand subfield indicates where an operand value for that instruction (if one is needed) might be found. This could be on the top of the stack, an argument to the most recent function application, or some special address in memory. The destination field indicates where to put the result (if any), and is similar.

Individual macroinstructions include a variety of wrinkles that improve performance. For example, when building a list of n elements, where n is known, an initial instruction allocates from the appropriate area of memory a set of n consecutive cells and returns the address of the first to a machine register. Later instructions then have a destination "NEXT-LIST," which stores their value in the next list entry and increments the pointer, without any extra storage allocation needed. The cdr-code tag is NEXT. The last such instruction returns the beginning of the object, as desired.

Calling functions is a similar inversion of what we are used to. An instruction of the form ***CALL ⟨function⟩*** precedes any code that computes arguments. It allocates a stack frame and stores all information necessary to return to the current context, along with the address of the code to be entered. Unlike a CALL on a conventional machine, however, control is not passed immediately to the called function's code. Instead a series of instructions follow for each actual argument, each ending with an instruction much like that above for lists, namely, a push of the argument value on top of the stack frame. The last such argument code ends with a LAST instruction, which pushes the value to the frame, stores the PC in the current frame, and branches to the code start.

Note that this procedure allows the called function to add local variables on the stack frame directly in back of the arguments and to access them in a very similar fashion. This makes the implementation of LISP functions like ***DECLARE*** or ***DEFINE*** easy. The description of the Symbolics machine in the next section diagrams this more precisely.

Finally, the macroinstructions for built-ins always test on the tags associated with their operands on the stack. If the tags match the normal cases (no data type conversions needed), normal processing continues. If the tags do not match the expected cases, multiway branches in the microcode go to the appropriate type conversion routines.

10.6 CADR DERIVATIVES

Although the CADR Machine was designed as a one-of-a-kind research machine, it rapidly spawned a series of commercial derivatives that were optimized for LISP execution in a high-performance, single-user workstation environment. The following sections discuss three of these derivatives as designed by Lambda Machines, Inc., Texas Instruments, Inc., and Symbolics, Inc. As with the descriptions of LISP variations, the emphasis here is on features that are architecturally unique. The references should be seen for more details of each design.

10.6.1 The LMI Lambda LISP Processor

(Smith, 1984)

The ***Lambda LISP processors*** from Lambda Machines, Inc., were some of the first commercially available machines tailored for LISP. They were fairly close to CADR-based designs (Figure 10-15) with the following additions:

- A richer set of tag values and matching support in the built-ins
- Two memory word sizes: a 32-bit size that supports a 25 bit (32m word) virtual address space, and a 40-bit size that supports a larger 32-bit address space and bigger floating-point number representations
- Larger PDL stack buffers of up to 2K words
- An internal 4K word ***A memory,*** which can be used by the microcode for internal and temporary results
- A sectored 4K word cache (16 words per sector) in front of memory to speed up memory accesses
- A standardized bus, the ***Nubus,*** to connect the back of the cache to memory and I/O cards
- A larger microinstruction word length of 64 bits
- A virtual microstore mechanism that permits up to 8K words of microprograms to be paged in as needed from a larger 64K word space on disk
- Direct hardware support for many arithmetic functions, such as multiplication and dispatching macroinstructions
- A faster, 100-ns clock speed
- A Motorola 68000-series microprocessor as a coprocessor for many operating system and console functions

The internal data flow of the Lambda Machine consists of an ***arithmetic logic unit*** or ALU that performs most of the basic operations. One of its two inputs is from the A memory, while the other can be driven by a copy of either the first 64 entries of the A memory (called the ***M memory***) or 32 other internal machine registers.

There are four classes of microinstructions out of which microprograms can be assembled. First is a relatively conventional ALU operation. The second permits bytes to be extracted or deposited into various registers. Third is a conditional jump class that permits testing of a wide variety of bits in different machine registers. Finally, a dispatch class permits up to 7 bits of information to be extracted from the M input and then used to compute a microbranch address. Together, these last three classes of microinstructions provide extensive and easy processing of tags, opcodes, and other random data fields—exactly what is needed to support a language like LISP.

The support for macroinstruction decoding consists of a ***Dispatch register*** which functions much like a prefetch buffer and permits two 16-bit instructions to be stored and decoded in a pipelined fashion inside the

CPU (Kogge, 1981). Special decode logic is connected directly to one of these, and lets opcode bits be vectored directly through a 256-entry ***Dispatch RAM*** (random-access memory) to select an appropriate starting microinstruction routine for that instruction. By using a RAM that can be loaded under program control, the mapping of instruction bits to microprograms can be quite dynamic, allowing major changes in the macroinstruction architecture to be made without hardware modification.

In many cases this decode through the Dispatch RAM can be overlapped with the previous instruction's execution, greatly enhancing performance over the original MIT design.

10.6.2 The Texas Instruments Compact LISP Machine

(Texas Instruments, 1984; Matthews et al., 1987; Wolfe, 1987)

Another close cousin to the CADR design is the TI Explorer series of machines, the most recent of which (called the ***Compact LISP Machine***) is based on a VLSI chip with 550,000+ transistors. Figure 10-18 diagrams at a high level the architecture of this chip and its usage in a complete processor.

The overall architecture of the machine resembles that of the Lambda machine, with the major differences being in the cache (a more modern, two-way set associative design), in support for garbage collection (8 status bits for each 8K-word region of memory), and in the use of page bits to extend the real memory address space to the full capability of the Nubus, namely, 1 gigaword.

The processor chip includes a significant amount of onboard memory (114,000 bits), which improves overall performance by minimizing memory traffic. This includes:

- A 1K×32-bit PDL buffer, like the CADR
- A 1K×32-bit A memory, which feeds the other side of the main data flow and is used by the microcode for internal values
- A 64×32-bit M memory, which feeds the same side of the data flow as the PDL, and again is used by the microcode
- A 2.5K×18-bit Dispatch Table to permit fast branching on up to 7 bits at a time
- A macroinstruction prefetch buffer capable of holding four 16-bit instructions
- A 32K×64-bit ROM for self-test and startup microcode

The processor is also pipelined, with the capability of initiating a new microinstruction each microcycle. A goal of 25 ns per clock cycle offers performance substantially in excess of earlier machines.

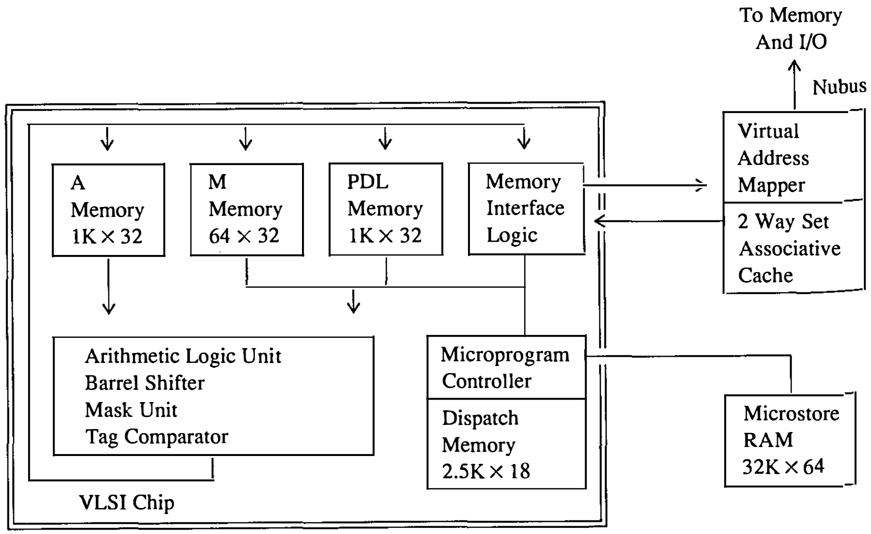


FIGURE 10-18
Compact LISP Machine.

10.6.3 The Symbolics 3600

(Moon, 1985; Symbolics, 1984)

Of all the CADR descendants, the Symbolics Corp. 3600 series of machines is perhaps the biggest divergence from the original design. These machines range from cabinet-sized machines with multiple cards of electronics to support the basic CPU to modern custom VLSI designs that capture must of the same logic in a single chip (the ***Ivory processor***).

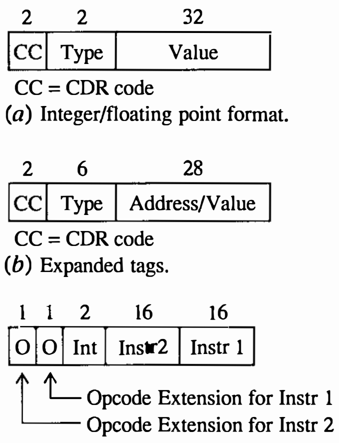
Although initially built to support ***ZetaLISP,*** a commercial equivalent to MacLISP, the basic architecture also adapts well to Common LISP and to supporting both traditional languages like FORTRAN and logic languages like PROLOG.

Perhaps the most immediate difference is in the size and format of memory words (Figure 10-19). A 36-bit word is used in several formats. In all cases the top 4 bits are used as a major tag consisting of a 2-bit cdr coding and a 2-bit initial type tag. For tags of integers and single-precision IEEE standard floating point this gives 32 bits for a data field, offering ranges and accuracies equivalent to those found on conventional computers.

The other two tag combinations signal an expanded mode where the next 4 bits of the word contain more tag information, and the remaining 28 bits contain a word address pointer or short immediate value. As with earlier machines, the addresses are virtual, with pages of 256 words brought in and out of disk as required.

Instructions are packed two to a word, with a tag of integer and the cdr-code field used as extra opcode extensions.

CONTROL STACK. Both the architecture of the machines and the ZetaLISP control software permit separate processes to be created and



(*a*) Integer/floating point format.

(*b*) Expanded tags.

(*c*) Insruction format.

FIGURE 10-19
A 3600 memory word.

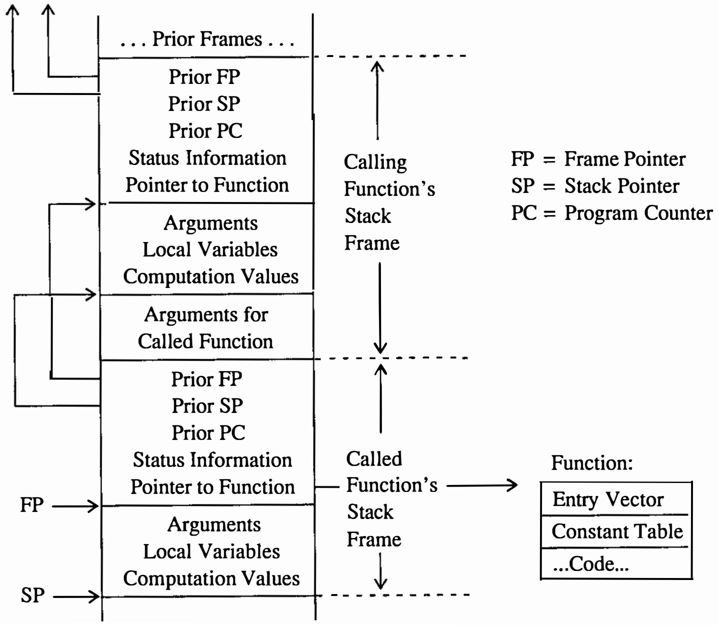


FIGURE 10-20
A 3600 control stack frame.

switched in and out of control. The current state of each process is governed by a ***stack group*** consisting of three stack pointers and their associated areas in memory. The ***control stack*** (see Figure 10-20) represents a combination of all four of the SECD Machine's registers. It contains a ***frame*** for each nontrivial function application that is still pending, with

the currently executing one on top. Each of these frames contains information about previous frames, about the current function, and about the current argument bindings. As with the SECD Machine, built-in functions use the area following the most recent frame as a conventional stack where operands can be pushed and popped as required.

There are two pointers into this stack area, the ***stack pointer*** (SP), which points to the topmost stack entry in use, and the ***frame pointer*** (FP), which points to the first argument in the most recent frame. Using FP as a base register permits indexed access back into this fixed block and forward into the arguments. The SP is thus similar to the S register of the SECD Machine, as the FP resembles the E register.

The fixed part of each frame (its ***header***) consists of the PC, SP, and FP at the time of the call of this function, similar to what is pushed on the dump of the SECD Machine. A fourth entry consists of a pointer to another block of memory built by the LISP compiler for the called function, which gives such things as the number of arguments to expect, a list of the constants the function uses, and the function's actual code. The final word of each frame contains status on how arguments are to be assembled, various options on how to return results to the caller, and so on.

FUNCTION CALLS. Calling a new function and building such a frame is more akin to conventional techniques rather than the inverted technique used in the original CADR. Argument expressions are evaluated one at a time, and their final values are pushed on the stack, as for built-ins. After this, a call instruction checks that there is sufficient room in the PDL buffer for the new frame. For this it uses information from the called function's ***entry*** (a 4-word block of cells). If space is insufficient, the most ancient frame in the PDL is pushed to the memory. The 5-word header can then be formed. It then verifies from the entry vector associated with the function to be called that there are enough arguments and how the function wants to view them, and copies them to the stack locations immediately following the header. This latter set of operations is to accommodate some of the special forms available in ZetaLISP that handle arbitrary numbers of arguments and default values for them. After forming all these arguments, control is then passed to the code for the called function.

A return instruction at the end of a function's code checks that the frame associated with the function to be returned to is in fact in the PDL, copies the return value(s) to that frame, and deletes the topmost frame. Again, ZetaLISP's options for handling multiple value returns is handled here. Bits in the frame header indicate whether none, one, or more than one value from the top of the stack are returned back into where the caller expects to see them. Other ***cleanup bits*** handle special conditions for popping stack frames, such as catch/throw or debugging.

The net result for most function calls is that after the return from the call, the arguments that the caller had placed on the stack have been replaced by the return values. Further, the overhead for just the CALL and RETURN instructions in many circumstances is as little as 20 machine cycles.

OTHER STACKS AND INSTRUCTIONS. The second stack in a stack group is the ***binding stack,*** which supports ***shallow binding*** of special and global variables. When required, the current bindings of such variables are pushed onto this stack in 2-word pairs. The first of these is a pointer to the special variable's value cell, and the second is the value at the time of the push. The cdr-code field of these cells is used to designate such things as use of the variables in a closure. Completion of an expression that releases a set of special bindings causes this stack to be popped an appropriate number of times, with each pair of words giving an address of a cell to change and the value that it should resume. Information in the stack frame indicates how much of the binding stack should be unwound at each function return.

This whole process is very similar to the ***trailing*** used in the WAM abstract machine for logic programs (Chapter 17) to remember and undo bindings when required.

The final stack is the ***data stack,*** which is akin to a heap in a conventional machine. Its purpose is to give an area of memory out of which large objects whose size is not known until runtime can be allocated.

Many instructions are derivatives of similar ones on the original CADR machines, but their implementation is optimized for performance. For example, the top entries on the stack are kept very close to the dataflow, with tag comparison logic operating in parallel with the ALU. Thus an ADD instruction (which will add any pairs of numbers regardless of their types) will compare both tags to INT at the same time as the values are being added by a standard adder circuit. If both are in fact integers (the normal case), the instruction completes in one machine cycle. Only if the tags are different is a multiway microlevel branch taken to specialized microcode to handle specific cases. The integer add result is discarded as irrelevant.

GARBAGE COLLECTION. (Moon, 1984). Garbage collection in the 3600 is a sophisticated descendant of Baker's algorithm. Given the potential size of the memory space, 256M words, and its structure as a virtual store where perhaps one-thirtieth of all memory is in fast RAM and the rest is on disk, it is impossible to conceive of a mark-sweep type of algorithm. It is also infeasible to consider a pure scavenging and copying algorithm—the space is simply too big.

Several solutions are employed in the 3600. First is to separate memory into ***areas*** that contain objects of projected different lifetimes. The ***ephemeral area*** contains objects whose lifetime is likely short. Most objects created during program execution for intermediate results fall into this category. This is scavenged and compacted frequently.

Next is a ***dynamic area*** that contains objects with a longer but not indefinite lifetime. Globals and special variables fall into this category. Garbage collection occurs here only when it has exceeded some predetermined capacity.

Finally there is a ***static area,*** whose objects are assumed to exist forever. System code and tables are examples. Garbage collection usually occurs here only when the user so commands.

Objects (actually the pages in which they reside) are marked with a ***level*** indicating the number of times they have been collected since their creation. When that level exceeds some other threshold, the objects are promoted to the next area.

Preventing accidental collection of ephemeral objects requires knowing when any reference to such an object is stored anywhere. This is done by hardware that monitors every single store into memory. If the value being stored is an address, and that address is to a memory page allocated to the ephemeral area, a bit associated with the page containing the location being modified is set. Conceptually, collecting the ephemeral area requires scavenging all pages whose bit is set for references to that area. Actually, a variety of specialized tables are employed to minimize time-consuming swapping in and out of pages from disk just to be scavenged.

Next, the actual process of scavenging and copying is also done to minimize future work. Instead of splitting all storage, or even all of a region, in half, ***tospace*** is incrementally increased as needed by allocating a new page of memory to it. By setting maximum capacities for areas, and promoting pages when full, the number of flips can be kept down.

Finally, a depth-first scavenge and copy tries to keep all of an object copied into a single new page whenever possible. This again minimizes the cross-page references that have to be scanned in later collection cycles.

10.7 OTHER LISP MACHINES

Although they have had the most impact, CADR-based machines are not the only specialized processors to be developed to support LISP. This section describes several others with interesting features.

10.7.1 The SCHEME-79 Chip

(Sussman et al., 1981)

The ***SCHEME-79*** chip started out as a class project at MIT to build a simple VLSI chip capable of supporting a subset of SCHEME. From the start of the class to delivery of the completed chips was about 6 months. This included developing all required support software, including much of the the VLSI chip development tools, most of which were themselves written in either SCHEME or MacLISP.

Despite some trade-offs to simplify design, performance appeared good. On a Fibonacci program, the resulting machine was from three to to nine times faster than a SCHEME interpreter running the same program on a DEC KA10. Further, of the time actually spent executing, almost 80 percent of it was for garbage collection, a number that could be reduced dramatically by better techniques (and more hardware support).

The memory format for SCHEME-79 assumes a 32-bit word where 1 bit is for garbage collection, 7 bits are for tag, and 24 bits are for address or value. Two sequential words make up a cons cell.

The garbage collection mechanism implemented on the chip is a ***mark-sweep*** system, where the spare bit in a car cell is used as a ***mark bit*** and the same bit in the cdr cell is used as an indicator that the cell has not been traced yet. The sweep phase actually compacts marked cons cells to the bottom of memory, making new cell allocation equivalent to the simple sequential allocation SCHEME mentioned in Section 8.1.1.

The machine language executed by this chip is called ***S-code,*** with programs constructed out on linked lists of cons cells. The type field of each cell contains an opcode, with the value field either an immediate operand value or a pointer to other instructions or associated data lists.

Figure 10-21 outlines the contents of the chip. There are 10 registers implemented in an arraylike fashion, with specific functions allocated to each register and specialized logic attached to the latch bits making up the register. Each register is formatted the same as a memory word, although in several cases either the tag field need not be implemented or can be hardwired to specific values. Examples of the specialized logic attached to a register include an incrementer for the register pointing to the next available memory cell and a comparator (separate tag and value) between a register used to hold values and the memory bus.

Of these registers the ones most resembling SECD registers included:

- the ***VAL register*** which acts like the top of the S register by holding the last value generated

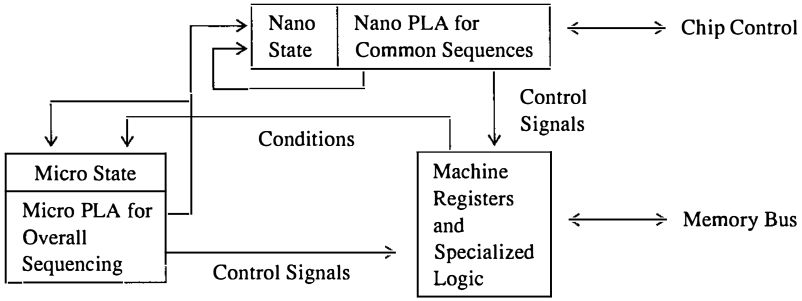

Nano State
Nano PLA for Common Sequences
Chip Control
Control Signals
Conditions
Micro State
Micro PLA for Overall Sequencing
Machine Registers and Specialized Logic
Memory Bus
Control Signals

FIGURE 10-21
SCHEME-79 chip.

- the ***ARGS register*** which holds the list of arguments being prepared for the next function call and corresponds to the rest of the S stack
- the ***DISPLAY register*** which acts like the E register to point to a list of argument lists for active functions
- the ***EXP register*** which acts like the C register in pointing to the current expression cell to be executed
- a ***STACK register*** which acts like the D register to point to a list of information permitting return to suspended functions
- the ***NEWCELL register*** which acts like the F register by indicating the next available free cell in memory

Other registers consist of pointers to the top of memory, temporaries, and even one hardwired to correspond to the nil value.

Instead of a microprogrammed controller, the SCHEME-79 uses a two-level hardwired finite state machine controller. Both are built from ***Programmable Logic Arrays*** (PLA) which consist of a regular array of AND gates whose outputs drive the inputs of an array of OR gates. The outputs of these OR gates are the control signals to the rest of the machine. The inputs to the AND array consist of the current state (as indicated by the contents of a ***state register***) and all the various status signals from the machine (such as the result of the tag compares mentioned earlier). Together this combination of AND and OR logic permits any logic function of the status and state signals to be computed directly.

The higher of the two controller levels is the ***micro-PLA***, and is responsible for overall sequencing of instruction execution and garbage collection. When some standard sequence of operations, such as a car, is required, it activates the second level ***nano PLA*** to perform it, and provides the nano PLA with signals indicating which registers the operation should be performed on.

Two latches, one for each level, remember where in each sequence each controller is. In some circumstances small micro- or nano-level "subroutines" are useful. In such cases, the appropriate return state registers are temporarily stored in one of the machine's ten registers.

The observant reader may notice that there is no ALU here; that is correct. Given the nature of this chip as a class project, the only arithmetic operations supported were a compare and an increment.

10.7.2 FACOM ALPHA LISP Machine

(Yuhara et al., 1986; Niwa et al., 1986; Akimoto et al., 1985)

The ***FACOM ALPHA*** is a processor designed as a backend machine to a conventional mainframe to support LISP and PROLOG. Unlike most of the previous machines, it was designed to support up to eight users simultaneously, with all I/O to the users and to mass storage handled through the mainframe host.

The architecture of the processor has many of the features of other machines. It is heavily microcoded, has a virtual memory system with separate spaces per user, and supports tags on memory words, a copying and compacting garbage collector, and significant stack buffering. Support. for the latter in particular reflects the multiuser nature of the machine. Each of the potentially eight concurrent users has his own 64K word virtual stack space, with a single 8K word hardware buffer shared among all processes.

Like the Symbolics machines, the design of the garbage collector was tied closely to that of the virtual memory. Memory is divided into two spaces, with each space divided into areas. Each area consists of a set of pages whose contents are all objects of the same kind (either symbols, strings, cons cell, or vectors). Since cons cells represent the greatest source of quick garbage, having all such cells in one area permits a scavenger to cover it quickly, with a minimum of page traffic to and from disk. An exponential smoothing algorithm is built into the memory management microcode to predict how much memory will be needed for each area during the next cycle, permitting main memory to be allocated accordingly, and thus minimize page faults.

Although performance of individual programs on the machine was good (up to six times faster than that for a good LISP implementation on a DEC 2060 or a FACOM M150F mainframe), its real benefit showed when it was used in its expected role as a backend accelerator for multiple users. Response time was up to 20 times better than for LISP on the mainframe when 30 users were on the system.

One of the valuable things about this machine is its instrumentation. Detailed measurements are available on how different features of the machine were used. Of these the most interesting were statistics about stacks and tag checking. Anywhere from 37 to 49 percent of all microinstructions references the hardware stack buffer in one way or another. Forty-eight to 55 percent of all microinstructions manipulated the pointers that control either this buffer or the full virtual stack. Only 5 to 11 percent referenced or set tag bits. Overall it is estimated that the machine is 3.55 times faster than a machine without these hardware features, with 75 percent of the speedup coming from the stack support and 8 percent from tag hardware.

10.7.3 The SPUR RISC

(Taylor et al., 1986)

One of the strongest trends in current computer architecture is toward ***Reduced Instruction Set Computers*** or ***RISC***s. Although there are major variations, in general all RISCs have the following characteristics:

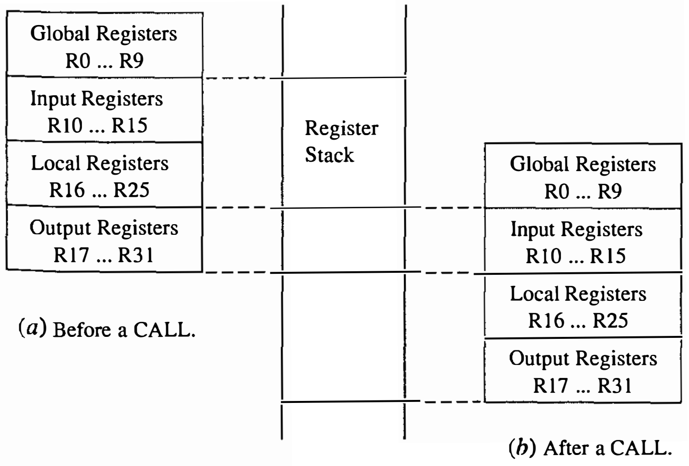
- A small number of simple instructions
- A large number of registers
- Most, if not all, access to memory is through just load and store instructions, (i.e. no memory to register adds, etc.)
- Almost all operations like add, compare, etc., are register to register, with the result going back into a register

- In most cases new instructions can be started at a rate of one per machine cycle
- An overall architecture which made generating optimized machine code from a high-level language easy

One of the first such RISCs was the Berkely RISC-I (Patterson, 1982), shortly followed by the RISC-II. This architecture included, in addition to the above, the concepts of:

- ***register windows*** where the registers accessible to the program are just part of a larger stack of registers, and at each call to or return from a subprogram, all or part of this set slides down and up as pictured in Figure 10-22.
- ***delayed branches*** where one or more instructions immediately following a branch will be executed even if the branch is taken.
- an efficient ***trap mechanism*** that will suspend execution of a program when certain circumstances have been detected, and start up some prespecified routine in its place.

The ***SPUR*** is a variant of the RISC-II optimized for handling languages such as LISP, PROLOG, or SMALLTALK. The major modification is the growth of each register and memory cell from 32 to 40 bits, with six of the extra bits used to hold tags, and the other two holding a


Global Registers R0 ... R9
Input Registers R10 ... R15
Local Registers R16 ... R25
Output Registers R17 ... R31
Register Stack
Global Registers R0 ... R9
Input Registers R10 ... R15
Local Registers R16 ... R25
Output Registers R17 ... R31

(*a*) Before a CALL.

(*b*) After a CALL.

• During a Program's Body, Arguments are loaded into Output Registers.
• After a Call, Old Output Registers Before New Input Registers,
with a new set of Local and Output Registers.
• At A Return, old Input and Local Registers recovered, and current
Input Registers revert to Output Registers (To carry results).
• Global Registers are never stacked.

FIGURE 10-22
Register windows in the RISC-II.

generation number used by the garbage collector to see how old an object is. As pictured in Figure 10-23, the other changes to the RISC-II ISA include specifying what happens with the tag fields on various operations, plus additional instructions that process the tags directly.

The trap facilities of the RISC-II are also used to advantage by streamlining many instructions to work only for the "expected" cases, and to trap to predefined routines to handle the atypical ones. The philosophy here is that this saves a great deal of code involved with pulling out tag bits and performing multiway branches on them. As an example, generic instructions like ADD perform in parallel a test of the tag fields of both input registers and an add of their values. If both tags equal the code for integers, the instruction completes in a single cycle without further work. If, however, either tag is not an integer, a trap is taken, suspending the current operation and starting a subroutine that will handle the type conversions.

Another example is the CAR/CDR variation of the LOAD-40 instruction. This loads the specified destination register from the specified memory address, but traps if the base register used in the address has a tag of anything other than cons or nil. Thus a displacement of "0" from a base gives us a CAR function in one instruction, while a "1" gives a CDR.

The net effect of all this is that most LISP functions compile very easily into very short sequences of SPUR code that execute in a very few machine cycles (very often one, as with CAR or CDR). CONS, for example, compiles into a four-instruction sequence. Again, special tests for conditions such as "stack overflow" are handled by setting up "inaccessible pages" at the ends of the allocated stack area in memory and letting a memory management trap handle the cases where something must be done.

Instruction	Function
load-40	Load tag and value from memory to register
store-40	Store tag and value to memory. Trap if gen. of data < gen. of address.
car/cdr	Identical to load-40 but trap if address tag not a cons or nil code.
load-32	Load just data from memory to register. Tag = int
store-32	Store just value to memory. Tag unchanged.
read-tag	Transfer tag field to value field.
write-tag	Transfer lower 7 bits of value to tag.
add,sub,...	Do fixed point operation if both tags = int. Otherwise trap.
cmp-branch	Compare 2 regs. (tag and data) and branch if result meets specified condition. Trap in some cases if one or more a pointer.
tag-cmp-branch	Similar to cmp-branch but just test tags.

Note: most instructions specify one or more registers to use as sources or destinations of data and/or memory addresses.

FIGURE 10-23
Partial SPUR instruction set.

Projected LISP benchmark performance for a 150-ns SPUR indicates that with the exception of heavy floating-point benchmarks, it is as much as 4.9 times faster than a Symbolics 3600 (with a slightly slower cycle time) and more than 10 times faster than a DEC VAX 8600 (which issues instruction twice as fast as the SPUR).

10.8 BENCHMARKING

(Gabriel, 1986)

Many of the machines and implementations of LISP described in this chapter have been interesting in their own right; almost all have features that one does not find in conventional computers. Other than some relatively general comparisons, however, there has been no attempt to determine which is the fastest or most efficient, and why. From an architect's viewpoint this is unsettling.

For conventional computers such performance comparisons are done by means of ***benchmarks*** or ***instruction mixes.*** The former represent some relatively complete programs that some group of people believe is in some way relevant to the kinds of problems of interest. Executing the same benchmarks on different computers and measuring execution times gives a measure of performance. Examples include the ***Whetstone program*** and the ***Dhrystone program.***

Instruction mixes represent an attempt to quantify as a percentage how often different instructions execute in a typical program, or stream of programs, of interest. The classical ***Gibson mix*** (based on analyses of IBM 7090 and S/360 instruction streams) indicates that perhaps 20–30 percent of all instructions are loads, 15 percent are stores, 20 percent are branches, and the rest are scattered among other functions. By measuring the average number of instructions executed per second (in units today of ***millions of instructions per second*** or ***mip***) one can get a relative comparison of two machines.

There are problems with both approaches. The benchmark approach suffers from different technologies, and thus different cycle times, on different machines, and from potentially different compilers (even for the same machine). Thus it is possible for a "fast" machine to come out slow against some benchmark because of limitations in the compiler's optimization capabilities, and for a "slow" machine to come out just the opposite.

The mix approach works only when the instruction sets of two machines are close, if not identical, and is often still very misleading because of the technologies used in the machines, assumptions that might have to be made relative to such things as cache hit ratios, the actual memory latency experienced by certain references, what kind of traffic might cause interference on various busses, how many page faults occur, etc.

Rather than give up at this point, architects for computers that execute conventional languages often take the approach of defining lots of rather small benchmarks that should expand with any decent compilers into relatively short sequences of predictable instructions (i.e., some predictable "mix"). Then, by comparing tables of results for different benchmarks, one can begin to identify the potential strengths and weaknesses of a particular machine. The ***Livermore loops*** and ***Linpack subroutines*** are just such examples for heavy numeric and scientific processing.

In addition, technology differences for different machines are often washed out by dividing measured execution times by the native machine cycle times, and expressing results in terms of ***cycles per instruction*** or ***cpi.***

LISP and the machines that implement it are no different. All of the LISP variants described here have different implementation needs, which in turn reflect on different compiler and hardware features. With the exception of the CADR-based machines, none of the machines described here has much in common other than perhaps tagged memory, and thus they are relatively immune to comparison by instruction mixes.

To overcome this, the LISP community has over the last 10 years or so established a set of relatively short programs which can be translated without much distortion into any of these LISP dialects and still tend to test the same things, usually things that any decent compiler will have good optimizations for. Figure 10-24 lists the names of these programs and a brief description of what they do. Figure 10-25 goes further and gives the LISP code for one of them, ***TAK.***

We will not attempt to give actual comparisons here and instead simply refer the reader to Gabriel (1986), where all these are documented in full, and actual measurements are given for many of the machines discussed here.

10.8.1 The Debate about Tag Support Hardware

(Steenkiste and Hennessy, 1987; Appel, 1989)

All the machines described in this chapter have direct hardware support for tags. When LISP-like language have been implemented on conventional machines, conventional wisdom has been that the cost of implementing tags and the associated shifting, masking, and compare and branches have been a significant performance penalty, and have been the "justification" for studying and building new architectures.

Within the last few years, however, a spirited debate has risen in the architecture community about the validity of this assumption. Much of the data for this counterattack has in fact come from analysis of small benchmarks from the set of Figure 10-24. The FACOM ALPHA data, for example, indicated that perhaps as little as 10 percent of the performance gain came from hardware tag support.

Program	Computational Feature Exercised
TAK	Deep, complex recursion with simple arithmetic at each call
STAK	Variant of TAK that uses special variables
CTAK	Variant of TAK that uses Catch and Throw
TAKL	Like TAK, but list processing instead of arithmetic
TAKR	TAK-like, but designed to thwart cache memories
DIV2	Divide a list of nils into two halves. Tests list builtins, function calls, and iteration.
FFT	1024 point Fast Fourier Transform using heavy floating point arithmetic.
FRPOLY	Compute powers of polynomials. Tests BIGNUMS.
FPRINT	Print a file.
FREAD	Read a file.
TPRINT	Read and write terminal.

(*a*) Benchmarks that test one aspect of a machine.

BOYER	A simple theorem prover about boolean and simple arithmetic expressions.
DESTRUCTIVE	Builds large lists and then modifies them.
TRAVERSE	Creates a 100 node directed graph and then traverses it, marking nodes as they are covered.
DERIV	Takes symbolic derivative of a formula expressed as an s-expression.
DDERIV	Like DERIV, but uses table of derivative formulas.
DER1	Like DERIV, but set up to test MacLisp compiler.
PUZZLE	Pack a variety of small pieces into 5x5x5 cube. Tests array constructs.
TRIANG	Given a triangle of 14 pegs in 15 holes, start with one and jump to remove all others.

(*b*) Composite operations.

FIGURE 10-24
The standard set of LISP Programs.

About this time compiler writers also began investigating how much type prediction and type checking could actually be done at compile time, thus eliminating altogether the need for either runtime hardware or generated code to check tags. A feeling began to grow that the potential was significant.

As an example of this, a study by Steenkiste and Hennessy (1986), looked in detail at a state-of-the-art RISC that did not have tag support (the Stanford University ***MIPS RISC*** in this case) and a highly optimizing LISP compiler. By switching compile-time type checking on and off, they determined that the actual runtime cost when the runtime code for built-ins had to do all the tag processing was about 22 percent by time, versus 25 percent for function call and return. While this is significant, it is not an overwhelming reason for building a nonstandard machine, especially for a commercial market.

```
(DEFINE tak (x y z)
            (IF (NOT (LESS y x))
                z
                (tak (tak (1 −  x) y z)
                     (tak (1 −  y) z x)
                     (tak (1 −  z) x y))))
```

Example: (tak 18 12 6) → 7

FIGURE 10-25
The TAK benchmark.

A later study (Steenkiste and Hennessy, 1987) then looked at a variety of tag implementation techniques for the same machine and the same set of benchmarks. These techniques included:

1. Keeping the tag in address bits that are ignored by the load and store instructions (namely, the bottom two)
2. Adding an instruction that tests the tag without requiring its extraction via shifts and masks
3. Both of the above
4. Hardware tests that trap when both arithmetic operations are not integers (such as the SPUR design)
5. Similar hardware traps for lists
6. Similar hardware traps for lists, vectors, and structures
7. A combination of techniques 1, 2, 4, and 6 above

The net result was that the simpler and most software-intensive of these, namely, the first three, by themselves gave up to a 14 percent speedup if full runtime type checking was employed. Adding support for the list operations and arithmetic (techniques 4 and 5) added another 3 percent, and the full set of techniques bought back over 22 percent—virtually all of the original slowdown.

In contrast, if compiler type checking was employed, the simple approach 3 above gave virtually the same speedup as the full hardware complement of approach 7. This should be treated as a real indication of the power of compile-time analysis, and a strong indication that architects of computers should first study carefully what can be done by compilers before adding relatively complex hardware features.

Further work (Appel, 1989) on functional languages such as ML (see Chapter 12) that give even more clues to the compiler about the expected runtime type of an object provide similar results—a good compiler can often predict the type of an object at runtime with enough accuracy that the code generated need never worry above tags and can simply execute as it would for a conventional language.

10.9 PROBLEMS

1. Assume that *X* is a LISP variable whose value is the list (1 (2 3) 4). Draw what *X* looks like (as a collection of memory cells) before and after evaluating (NCONC (RPLACA (CDR *X*) (CDADR *X*)) (CADR *X*)). Label which cells are new, which cells had values changed, and where the result of the NCONC points.
2. Expand the s-expression interpreter of Figure 7-9 to permit the body of a lambda expression to look like

   (lambda ((⟨*id*⟩*) ⟨**expression**⟩$^+$)

   where the list of expressions in the body is evaluated from left to right, and the value of the last one is returned as the value when the lambda is used as a function.
3. Expand the answer to Problem 2 to include labels and GO expressions.
4. What kind of modifications to the SECD machine would be necessary to support the answers to Problems 2 and 3 when code is compiled?
5. Write (in original LISP) programs which define MAP, MAPLIST, MAPCAR, and MAPC.
6. Define in SCHEME an extended syntax that takes products of an arbitrary number of arguments, assuming that the only built-in multiplies exactly two values.
7. What is the second set of bindings that EXPAND makes in the example of Figure 10-10?
8. Develop an abstract program interpreter for the core subset of SCHEME (Figure 10-9) plus the DEFINE form.
9. Repeat Problem 8, except generate an SECD compiler. (You may have to invent a new instruction to help handle SET!).
10. How would you extend either of the interpreters in Problems 8 and 9 to handle FLUID and FLUID-LET?
11. Devise how you might support a fully lazy SCHEME-like language on top of MultiLISP.
    a. First assume that execution will be on a single computer.
    b. Then assume a multiple-processor computer that can share memory access.
12. Rewrite the TAK benchmark of Figure 10-25 in a variety of LISP dialects. What special features does each buy you (if any)?
13. Develop an SECD program for TAK. Then sketch what the code might look like for a SCHEME-79-like or a SPUR-like architecture. (Feel free to invent any instructions you feel are necessary, but give definitions of how they operate.)
14. Define a method of passing arguments to, and returning results from, a LISP function compiled into the SPUR architecture. Comment on how closures might be built in such an architecture, and how the funarg problem might be handled.
15. Compare the different mechanisms described in this chapter for invoking user-defined functions. How much hardware and compiler support does each need, and what are the performance advantages and bottlenecks?

CHAPTER 11

COMBINATORS AND GRAPH REDUCTION

As described here, lambda calculus is a complete theory that describes much of mathematics with a single simple semantic model. Its only major difficulties are that care must be taken with identifier names (the renaming rule) and that a relatively large number of different functions are needed for even simple expressions.

Combinator theory is an interesting variation of lambda calculus that has neither of these difficulties. It consists of a minimal set of functions from which equivalents to all lambda expressions can be constructed by a very minimal translation algorithm. Further, the resulting expressions are simple compositions of combinators, meaning that we need no other lambda functions or identifiers (bound or otherwise).

The origins of this theory come from the observation that all identifiers in a lambda expression really "steer" copies of actual arguments to appropriate spots in a function's body. If we permit multiple argument functions, then it is possible to conceive of doing the same thing by creating functions which simply reorder their arguments in ways that end up duplicating the effects of this steering. These reordering functions are combinators, and they can be fully described without need of identifiers or the substitution rule.

The following subsections discuss combinator theory in more detail. First will come a description of a minimal set, followed by the process by which arbitrary lambda expressions can be converted to combinator form. This involves the ***bracket abstraction*** process, by which the identifiers are "abstracted" out of an arbitrary lambda function. Following this will come a discussion of other useful combinators beyond

the basic set, and optimization algorithms that can take one combinator expression and simplify it by using these additional combinators.

Next will come a model of computation that represents combinator-based expressions as graphs, and performs computations on them by ***graph reduction*** of subtrees. Such reduction strategies seem to violate the long-held principle of avoiding self-modifying code, but they have the advantages of extreme simplicity and near-automatic lazy evaluation.

Yet another optimization technique of growing importance spends effort at compile time to develop ***supercombinators,*** which are highly optimized for particular subexpressions in the program being compiled. This approach forms the basis of much of modern research into implementing functional languages using graph reduction. Section 11.6 describes the process of identifying such supercombinators and how to optimize their construction and use.

The final sections describe several interesting machines that rely on combinator theory for their basic operation. In particular, the ***G-Machine*** represents yet another abstract machine of growing importance in the compilation and execution of functional programs.

Perhaps the key ideas a reader should take away from this chapter are the concept of a combinator as an alternative to substitution, and graph reduction as a different way of structuring and managing a functional evaluation. For alternative references, the reader is referred to Curry and Feys (1958), Seldin and Hindley (1980), Turner (1979a), Burge (1975), Jones and Muchnick (1982), and Diller (1988).

11.1 BASIC COMBINATORS

One of our first notational simplifications for lambda calculus was to give names to standard lambda functions that are used over and over in real expressions (such as integers, "+," "×," " ",...). Whenever such labels show up in an expression they always mean the equivalent function. Since their meaning is "constant," it is not unreasonable to formally call such functions ***constants.***

When confronted with such an obviously useful notation, one of the first reactions of a good mathematician is to see what is the computational power of a sublanguage consisting only of such constants, and whether one set of constants is more useful than another. This was done with lambda calculus, with amazing results. All of lambda calculus can be developed from a set of only three constants (one of which is actually redundant). Further, it can be done with no lambda function definitions, just applications of one constant to another.

11.1.1 Constant Applicative Forms

The particular sublanguage of interest is called the language of ***constant applicative forms,*** with a single syntatic class called a ***caf.*** The basic syntax for cafs is the essence of simplicity: A caf is either a constant or the application of two cafs:

$$\langle \mathit{caf} \rangle := \langle \mathit{constant} \rangle \mid (\langle \mathit{caf} \rangle\ \langle \mathit{caf} \rangle)$$

Not present in this definition are ***free variables*** and lambda functions ($(\lambda x| \ldots)$).

As with lambda calculus, when multiple arguments to a function are involved, we will delete the innermost "( )," permitting cafs of the form:

$$\langle \mathit{caf} \rangle := \langle \mathit{constant} \rangle \mid (\langle \mathit{caf} \rangle\ \langle \mathit{caf} \rangle^{+})$$

Also, as with lambda calculus applications, the leftmost caf in an application is the function, those to its right are its actual arguments. And the fundamental meaning of such multiple-argument forms is the same as for lambda calculus. The innermost function is ***curried*** by accepting one argument at a time and producing a function of one fewer arguments.

This is lambda calculus without the lambda function definition (and as such looks a lot like s-expressions). We cannot define new functions as lambda expressions; we can only build functions from whatever constant functions are provided.

The constants for this sublanguage will be chosen carefully. What we want are functions that, when given cafs as arguments, always give cafs as a result. No free variables or embedded lambda functions that are not constants will appear. Further, we want to be able to define their operation without concern about substitutions and possible identifier clashes. In fact, there will be no need even to introduce the idea of an identifier, binding variable, or the like, in the semantics.

There are two categories of such constants. The first is the common set of arithmetic functions, integers, booleans, etc., which we know full well how to translate to pure lambda expressions, but would rather not. These are to be ***built-in*** to our language and may vary from one language to another. Functions in this class have the property that both their domains and their ranges are created from objects which we also call constants. For example, "+" maps from pairs of integers to integers (or in curried form from integers into functions of integers to integers). The result is that a caf created from such constants will, when evaluated, return another expression which is still a caf.

11.1.2 S, K, and I

The second set of constants, called ***combinators,*** are more general and, in a sense, more basic. These constants represent some particularly well-behaved functions out of which all others can be built, including the first set of constants. By definition, a combinator is any lambda function which:

- Has no ***free variables*** in it (no identifiers not surrounded by a controlling $(\lambda \ldots)$)
- Has a body in a caf form (or convertible to caf form)
- Or, more simply, when given arbitrary cafs as arguments, always returns a caf result

To make a combinator a constant, we will give each one a unique label and use that in place of the combinator's full lambda equivalent.

Although there are an infinite number of such combinators, some are more important than others. In particular, the three combinators **S**, **K**, and **I** defined in Figure 11-1 are usually considered most key. If **A**, **B**, and **C** are arbitrary cafs, then applying **S**, **K**, or **I** to them results in new cafs that are just rearrangements of the input. Computing such applications are thus much simpler than for more general lambda functions; we need no internal detail on identifiers or on the possible complexities resulting from substitutions and renamings.

In contrast, consider the function $(\lambda x|x(\lambda y|xyx)z)$. Applying this to an arbitrary expression **A** cannot be carried out and written down without knowing what free variables exist in **A**, and we still end up with an unremovable lambda expression in the middle of the result. In this form it is definitely not a candidate for inclusion as a basic combinator constant.

Each of the combinators in Figure 11-1 has a relatively easily understood interpretation. The ***I combinator*** is basically the ***identity function,*** which returns its single argument totally unmodified. The ***K combinator*** is a function that takes two arguments and returns a copy of the first. The second is discarded. In its curried form, it takes an arbitrary object **A** and creates from it a function which takes any other object as an argument and always returns **A**. For this reason it is known as the ***constant combinator*** or ***elemental cancellator.*** It is equivalent to to our previous definition of T.

The ***S combinator*** is a function that takes three arguments $\mathbf{A}_1$, $\mathbf{A}_2$,

Combinator Name	Lambda Equivalent	Semantics
S—Distributed Application	(λxyz|xz(yz))	S A B C → A C (B C) or (((S A)B)C) → ((A C)(B C))
K—Constant	(λxy|x)	K A B → A or ((K A)B) → A
I—Identity	(λx|x)	(I A) → A

Note: I = S K K = ((S K) K)
A, B, C arbitrary expressions

FIGURE 11-1
Basic combinator constants.

and A_3, and returns an application where A_1 and A_2 are both treated as functions, and are given as arguments a copy of A_3. The result of applying A_1 to A_3 is then used as a function over (A_2 A_3). Since **S** distributes its third argument over two other applications, it is known as the ***distributed application combinator*** or sometimes as the ***elemental compositor.***

The typical use of the **S** combinator is in its curried form, when it is given two functions and creates as a result a new function that accepts a single argument into an application involving the first two. As an example of this, consider the function **S K K.** The lambda equivalent is ($\lambda z|$**K** z(**K** z)). Using the definition of **K,** the body of this is the first argument z, that is, ($\lambda z|z$). Thus **S K K** is the same as the function **I.**

Other simple examples include the caf (**S** × **I**), which is equivalent to a function which squares its argument, or (**S** + **sqrt**) which is equivalent to the function ($\lambda x|+$ x (**sqrt** x)).

The next section will expand upon the generality of this set of combinators.

11.2 BRACKET ABSTRACTION

(Schoenfinkel, 1929; Turner, 1979b; Hughes, 1982)

In terms of computation, the most relevant piece of lambda calculus syntax is the application, where the function is in prefix form to the left of its argument. Conceptually, this is simple to evaluate; we give the function its argument value, and replace the pair by the result.

The actual complexity (such as it is) comes into play when we look at the definition of a typical function in the form ($\lambda x|$**M**). The x is a placeholder for the actual argument, and all free occurrences of it will receive copies of that argument when the application is reduced. Further, there are no constraints on how many and where the free x's can be in **M.** They can be intermixed freely with other arbitrary lambda expressions. The result is that intermediate steps in the function's evaluation must constantly involve taking strings of characters, looking them over for certain characters in special situations, dismantling the strings at these points, substituting new strings, and reassembling.

Now consider what would happen if we could always rewrite the arbitrary body expression **M** of a function ($\lambda x|$**M**) in a form where there is at most one free x, it is always the last character, and the rest of the expression is built out of well-behaved constants such as combinators. In other words, we find some caf **N** with no free x's such that ($\lambda x|$**M**) and ($\lambda x|$**N** x) are the same function, i.e.,

$$\text{for all } \mathbf{A},\ (\lambda x|\mathbf{M})\mathbf{A}=(\lambda x|\mathbf{N}\ x)\mathbf{A}=\mathbf{N}\ \mathbf{A}$$

The existence of such an **N** has some profound consequences. Using the eta conversion rule discussed in Chapter 4, the entire expression ($\lambda x|$**M**)=($\lambda x|$**N**x) can be replaced by simply **N.** We have "abstracted out" the need for the identifier x, and "compiled" the original function into a string of simple constants without identifiers or "λ's."

The processing of abstracting x out of **M** is called ***bracket abstraction*** and is written [x]**M.** This name and notation come from the analogy that we are "reversing" the basic substitution mechanism of applications; we are pulling something out of an expression rather that putting something in.

The interested reader is also referred to the work by Berkling (1986) called ***head order reduction,*** where something like bracket abstraction in reverse is used to convert a lambda expression to one where all the functions have the form ($\lambda x_1 \ldots x_n|x_j$). These functions are called ***selectors,*** and they can be implemented easily as indices into the environment.

11.2.1 Basic Translation

Figure 11-2 lists the major syntax rules for forming a lambda calculus expression **M**, and for each one shows how it would be converted into an equivalent caf form [x]**M** when some variable x is to be abstracted out. Thus, in all cases, we are free to replace an expression **M** wherever it appears by the form ([x]**M** x).

The cases covered by the second rule of this figure are worth closer consideration. This rule governs the case when the expression **M** is a single symbol that is not the same as the identifier being abstracted out. Thus **M** could be any identifier other that x, or any constant, combinator, built-in function, integer, etc. Thus [x]1 ⇒ (**K** 1), just as [x]**S** ⇒ (**K S**).

To repeat what this table means, each entry in the [x]**M** column is an expression which, when applied as a function to the variable being abstracted (x in this case), will yield an expression which is totally equivalent to the original **M.** This permits us to take lambda functions of the form ($\lambda x|$**M**) and replace them by ($\lambda x|$([x]**M**)x) or, even more simply, by [x]**M.**

The proofs that each [x]**M** has the property that ([x]**M**)x ⇒ **M** are direct. For example, ([x]x)x=**I**x ⇒ x by the definition of **I.** Also, ([x]y)x=((**K** y) x)=(**K** y x) ⇒ y by the definition of **K.** Again note that this holds for constants as well as identifiers other that x.

Syntax of M	Example	Equivalent [x]M
<id>	x	I—M the identifier x
<id>	y	(Ky)—where y a constant or identifier ≠x
<application>	(FA)	(S [x]F [x]A)
<function>	(λx|E)	K ([x]E) Binding variable is x
<function>	(λy|E)	[x]([y]E) where y≠x

FIGURE 11-2
Bracket abstraction process.

Second, to show that $([x](\mathbf{PQ}))=(\mathbf{S}\ [x]\mathbf{P}\ [x]\mathbf{Q})$, we apply x to both sides and reduce:

$$([x](\mathbf{PQ}))x=(\mathbf{S}\ [x]\mathbf{P}\ [x]\mathbf{Q})x \Rightarrow [x]\mathbf{P}\ x\ ([x]\mathbf{Q}\ x)$$

which is equivalent to **P Q**. (We assume recursively that the process worked correctly for $[x]\mathbf{P}$ and $[x]\mathbf{Q}$; that is, $[x]\mathbf{P}\ x=\mathbf{P}$ and $[x]\mathbf{Q}\ x=\mathbf{Q}$.)

The final two rules of Figure 11-2 handle the case where we are attempting to abstract an identifier out of a lambda function. We first abstract the function's identifier out of its body, and then abstract the first identifier out of the result. As a special case, if the binding variable is the same as x, then the process reduces to the first of these rules. The second case, however, is always valid.

Proof of this last rule is also recursive. If the bracket abstraction process works, then $(\lambda y|\mathbf{E})=([y]\mathbf{E})$, which is an expression without lambda functions. The expression may, however, have embedded x's in it. Thus, abstracting the x from it yields a caf.

These final rules also indicate how to get the abstraction process started in the first place. If our goal is to translate a lambda function to its equivalent caf form, we start by abstracting out of the function's body the function's first binding variable. To convert a multiple argument function such as $(\lambda x_1 \ldots x_n|\mathbf{M})$ to combinator form, we abstract out the first identifier x_1, which in turn requires abstracting out x_2 first. This requires abstracting out x_3, and so on until all the argument identifiers are abstracted out, as follows:

$$\begin{aligned}(\lambda x_1 \ldots x_n|\mathbf{M}) &= (\lambda x_1(\lambda x_2(\ldots(\lambda x_n|\mathbf{M})\ldots)\\ &= [x_1](\lambda x_2(\ldots(\lambda x_n|\mathbf{M})\ldots)\\ &= [x_1]([x_2](\lambda\ldots(\lambda x_n|\mathbf{M})\ldots)\\ &= [x_1]([x_2](\ldots([x_n]\mathbf{M})\ldots)\end{aligned}$$

11.2.2 A Simple Compiler

To complete our picture of this whole compilation process, Figure 11-3 gives an abstract program that accepts an arbitrary lambda function and converts it into a pure caf. The special case for function definitions having the same binding variable as x is not handled here—the general case is applicable, albeit with more work. Abstract syntax predicates and functions are used to take the input apart, test it, and put the result back together. Their meaning should be obvious.

The process of creating a compiler for s-expressions is direct replacement of the abstract functions by the appropriate list processing functions, and is left as a problem for the reader.

Figure 11-4 gives several simple examples of this translation process plus a check for each one that the resulting caf is correct.

```
compile(e) = "compiles lambda definition e into combinators" =
abstract(get-id(e),get-body(e))

abstract(x,e) = "abstract out identifier x from e" =
  if is-a-constant(e)
  then if e=x
       then create-constant(I)
       else create-application(K,e)
  elseif is-a-function(e)
  then abstract(x,compile(e))
  else "e is an application"
       create-application(
         create-application(S, abstract(x,get-function(e)),
         abstract(x,get-argument(e)))
```

FIGURE 11-3
Lambda to combinator abstract compiler.

11.2.3 A Larger Translation

For larger problems the process is the same, only more tedious, particularly when we encounter an expression of the form $(\mathbf{E}_1\mathbf{E}_2\ldots\mathbf{E}_n)$ [remember that this is equivalent to $((\ldots(\mathbf{E}_1\mathbf{E}_2)\mathbf{E}_3)\ldots.\mathbf{E}_n)$].

The third rule of Figure 11-2 gets used over and over again on essentially the same expressions. Each time it is applied, another **S** combinator is added, and bracket abstraction of one more $\mathbf{E}_i$ is initiated. The process is so common and so mechanical that it is worth working out once and using thereafter as a standard expansion. Figure 11-5 gives this rule and its proof.

As a larger example, Figure 11-6 gives a detailed example of the conversion of the successor function $(\lambda xyz|y(xyz))$ to a caf form using only **S**, **K**, and **I** as constants.

Finally, as an example of a bigger operation, Figure 11-7 takes the main body of the factorial function and converts it to combinators, although in this case we do assume built-in functions for basic math, integers, and the conditional if. A later section will complete the conversion of the full factorial to combinator form, including the necessary recursion control.

11.3 ADDITIONAL COMBINATORS

(Turner, 1979b; Burge, 1975; pp.41–43)

Two things are obvious from Figure 11-6. First, the compilation process seems to explode all out of proportion to the complexity of the original function. In fact, expressions of n symbols often balloon up something approaching 2^n symbols when put in basic combinator form. Second, in such expansions there are many repetitions of the same subexpressions [e.g., **(S(K K)(KS))**]. Both need to be reduced to make combinator theory a practical basis for practical computers.

Example: (λx|+ 1 x) where + and 1 are builtin constants
→ [x](+ 1 x)
→ S([x](+ 1)) ([x]x)
→ S (S ([x]+) ([x]1)) I
→ S (S (K +) (K 1)) I

Check: evaluate (S {S (K +) (K 1)} I) 3
→ {S (K +) (K 1)} 3 (I 3)
→ (K +) 3 {(K 1) 3} (I 3)
→ + {(K 1) 3} (I 3)
→ + {1} (I 3) = + 1 3 = 4

(*a*) A simple function with constants.

Example: TRUE = (λx|(λy|x))
Note: this is also definition of K.
→ [x]([y]x) ⇒ [x](K x) → S ([x]K) ([x]x) → S (K K) I

Check: Evaluate S (K K) I A B
→ (K K) A (I A) B
→ K K A (I A) B
→ K (I A) B → (I A) → A

(*b*) A nested function.

Example: (λx|+ (× x x) x)
Note: if fully parenthesized: (λx|((+ ((× x) x)) x))
→ (S [x](+ (× x x)) [x]x)
→ (S (S [x]+ [x](× x x)) I)
→ (S (S (K +) (S [x](× x) [x]x)) I)
→ (S (S (K +) (S (S [x]× [x]x) I)) I)
→ (S (S (K +) (S (S (K ×) I) I)) I)

Check: Evaluate (S (S (K +) (S (S (K ×) I) I)) I) 3
→ (S (K +) (S (S (K ×) I) I)) 3 (I 3)
→ (K +) 3 ((S (S (K ×) I) I) 3) (I 3)
→ + ((S (K ×) I) 3 (I 3)) (I 3)
→ + ((K ×) 3 (I 3) (I 3)) (I 3)
→ + (× (I 3) (I 3)) (I 3)
→ + (× 3 3) 3 → 27

(*c*) Nested abstractions.

FIGURE 11-4
Some simple combinator compilations.

The solution to this lies in two related techniques. First is the definition of some auxiliary combinators that are useful in certain circumstances. Second is the development of some optimization techniques that can replace one combinator caf by an equivalent but shorter one. An example of the former is the ***Y combinator*** introduced earlier to handle recursion in lambda expressions. Such techniques are discussed more fully in the next section.

[x](E_1 E_2 ... E_n)
= [x]({...(E_1 E_2) E_3)...} E_n)
= (S [x]{...(E_1 E_2) E_3)...E_{n-1}} [x]E_n)
= (S { S [x](...(E_1 E_2) E_3)....E_{n-2}) [x]E_{n-1}} [x]E_n)
...
= S(S(S...(S [x]E_1 [x]E_2) [x]E_3) [x]E_4)...[x]E_n)
n − 1 S's

FIGURE 11-5
Abstracting out of a chain of applications.

Figure 11-8 gives a table of useful combinators. As before, each one simply rearranges its arguments, guaranteeing that if the arguments are cafs, then the result is a caf. Also note that many of these combinators have shown up before as equivalents for well-known objects such as integers.

It is worth the reader's time to prove that the operation of each combinator (its semantics) matches the lambda definition. It is also instructive to convert each of these lambda definitions into strings of pure **S, K,** and **I** combinators. This both reinforces the compilation process and indicates the power of these new combinators.

Several combinators from this set are worth discussing further. First, **B** (the ***basic composition combinator***) accepts three arguments—the first two of which are treated as functions. The result is the first applied to the result of applying the second to the third. This is exactly ***composition*** by previous definitions.

Next, **C**, the ***reversal combinator,*** takes three arguments and returns three, but with the last two reversed. **W**, the ***repeat combinator,*** accepts two arguments and returns three, where the last is the second repeated.

The ***primitive recursion combinator*** **R** implements the equivalent of a ***do-loop*** in conventional languages. It accepts three arguments: a termination value, a function, and an index. The index must be a nonnegative integer, and if it is 0, the value returned by the expression is the termination value. If it is nonzero, the function is applied to two arguments: the integer, and a copy of the original **R** expression with the integer decremented by 1. The result is the function composed with itself multiple times, with the index provided as an argument in each iteration. As an example, consider:

R a f 3 ⇒ f 3 (R a f 2)
⇒ f 3 (f 2 (R a f 1))
⇒ f 3 (f 2 (f 1 (R a f 0)))
⇒ f 3 (f 2 (f 1 a))

```
(λxyz|y(xyz)) - "Successor Function"
—> [x](λyz|y(xyz)) —>[x]{[y](λz|y(xyz))} --> [x]{[y]([z]{y(xyz)})}
--> [x]{[y](S [z]y [z](xyz))}

—> [x]{[y](S (Ky) (S{S  [z]x  [z]y}  [z]z))}

—> [x]{[y](S (Ky) (S{S (Kx) (Ky)}    I))}

—> [x]{S{S  [y]S   [y](Ky)}          [y](S{S(Kx)(Ky)}I)}

—> [x]{S{S  (KS)  (S [y]K [y]y)} (S(S [y]S [y]{S(Kx)(Ky)}) [y]I)}

—> [x]{S{S(KS)   (S(KK)   I)}  (S(S (KS) {S(S [x]S [y](Kx)) [y](Ky)}) (KI))}

--> [x]{S{S(KS)(S(KK)I)}(S(S (KS) {S(S  (KS) (S [y]K [y]x)) (S [y]K [y]y)}) (KI))}

—> [x]{S{S(KS)(S(KK)I)} (S(S(KS) {S(S (KS) (S (KK) (Kx))) (S (KK) I)}) (KI))}

                                              E
—> S(S [x]S [x]{S(KS) (S(KK)I)}) [x](S(S(KS{S(S(KS)(S(KK)(Kx)))(S(KK)I)}) (KI))

—> S(S (KS) {S(S [x]S [x](KS)) [x](S(KK)I)})   (S{S [x]S [x]E}     [x](KI))

--> S(S(KS){S(S(KS) (S(KK)(KS))) (S{S(KS)(S(KK)(KK))}(KI))})(S{S(KS) [x]E}(S(KK)(KI)))

where [x]E = [x](S(KS) {S(S(KS)(S(KK)(Kx))) (S(KK)I)})
= S (S [x]S [x](KS))  [x]{S(S(KS)(S(KK)(Kx))) (S(KK)I)}
= S {S (KS) (S(KK)(KS))} {S(S [x]S [x](S(KS)(S(KK)(Kx)))) [x](S(KK)I)}

= S {S (KS) (S(KK)(KS))} {S(S (KS) [x](S(KS) (S(KK)(Kx)))) (S{S [x]S [x](KK)} [x]I)}
= S{S(KS)(S(KK)(KS))}{S(S(KS)(S{S[x]S [x](KS)} [x](S(KK)(Kx))))(S{S(KS)(S(KK)(KK))}(KI))}
= S{S(KS)(S(KK)(KS))}{S(S(KS)(S{S(KS)(S(KK)(KS))}{S(S [x]S [x](KK)) [x](Kx)}))
                                                              (S{S(KS)(S(KK)(KK))}(KI))}
= S{S(KS)(S(KK)(KS))}{S(S(KS)(S{S(KS)(S(KK)(KS))}{S(S(KS)(S(KK)(KK)))(S(KK)I)}))
                                                              (S{S(KS)(S(KK)(KK))}(KI))}

Thus total solution is:
S(S(KS){S(S(KS)(S(KK)(KS)))(S{S(KS)(S(KK)(KK))}(KI))})(S{S(KS)S{S(KS)(S(KK)(KS))}
{S(S(KS)(S{S(KS)(S(KK)(KS))}{S(S(KS)(S(KK)(KK)))(S(KK)I)}))(S{S(KS)(S(KK)(KK))}(KI))}}
                                                                              (S(KK)(KI)))
```

FIGURE 11-6
Successor function without built-ins.

```
letrec f = (λn|(if (= n 0) 1 (× n (f (− n 1))))))
   → [n](if (= n 0) 1 (× n (f (− n 1)))))
   → (S(S(S [n]if [n](= n 0)) [n]1) [n](× n (f (− n 1))))
   → (S(S(S (K if) (S(S [n]= [n]n) [n]0)) (K 1)) (S(S [n]× [n]n) [n](f (− n 1)))))
   → (S(S(S (K if) (S(S (K =) I) (K 0))) (K 1)) (S(S (K ×) I) (S [n]f [n](− n 1)))))
   → (S(S(S (K if) (S(S (K =) I) (K 0))) (K 1)) (S(S (K ×) I) (S (K f) (S(S [n]− [n]n) [n]1)))))
   → (S(S(S (K if) (S(S (K =) I) (K 0))) (K 1)) (S(S (K ×) I) (S (K f) (S(S (K −) I) (K 1)))))
```

FIGURE 11-7
Bracket abstraction on factorial.

Combinator Name	Lambda Equivalent	Semantics
S′	(λwxyz|w(xz)(yz))	S′ M N O P → M (N P)(O P)
B—Composition Combinator	(λxyz|x(yz))	B M N O → M (N O)
B′	(λwxyz|wx(yz))	B′ M N O P → M N (O P)
C—Reversal Combinator	(λxyz|xzy)	C M N O → M O N
C′	(λwxyz|w(xz)y)	C′ M N O P →M (N P) O
W—Repeat Combinator	(λxy|xyy)	W M N → M N N
Y—Fixed Point Combinator	((λy|(λxy|y(xx)) (λxy|y(xx)))	Y M → M (Y M)
P	(λxyz|zxy)	P M N O → O M N
T	(λxy|yx)	T M N → N M
J	(λx|I)	J M → I
Z—Zero or True	(λxy|x)	Z M N → M Also Z = K I
1,..Integers	(λnz|n(n(n...(nz)...) as above	
R—Primitive Recursion	—complex—	R M f Z → M R M f i → f i (R M N(i−1))

FIGURE 11-8
Some additional combinators.

If **f**$(x,y)=x \times y$ and **a** is 1, the result of this is 3!. Thus, treating "×" as a prefix operator, we get that **factorial**(n)=**R** 1 × n, which gives an elegant caf form for **factorial** of simply (**R** 1 ×).

11.3.1 Some Interesting Combinator Expressions

(Curry et al., 1972; Diller, 1988)

As an interesting insight into the power of these additional combinators, Figure 11-9 gives combinator-based expressions which function as we

Basic Function Equivalents:
cons = B C (C I)
car = C I K
cdr = = C I (K I)

Proof that car works: car(cons(A_1,A_2))
= C I K (B C (C I) A_1 A_2)
→ I (B C (C I) A_1 A_2) K
→ B C (C I) A_1 A_2 K
→ C (C I A_1) A_2 K
→ C I A_1 K A_2
→ I K A_1 A_2
→ K A_1 A_2
→ A_1

Proof for cdr: cdr(cons(A_1,A_2))
= C I (K I) (B C (C I) A_1 A_2)
→ I (B C (C I) A_1 A_2) (K I)
→ (B C (C I) A_1 A_2) (K I)
→ C (C I A_1) A_2 (K I)
→ C I A_1 (K I) A_2
→ I (K I) A_1 A_2
→ K I A_1 A_2
→ I A_2
→ A_2

FIGURE 11-9
Combinator expressions for list functions.

would like the list functions ***cons, car,*** and **cdr** to function. What they actually do is immaterial to how they act in expressions based on them. For example, one could define the semantics of these three functions as a set of functions that obey the following properties:

- **cons** takes two arbitrary arguments.
- **car** and **cdr** take only one argument, and the result from that argument is undefined for anything other than an expression that is the **cons** of two things.
- For all expressions $\mathbf{A_1}$ and $\mathbf{A_2}$, **car(cons($\mathbf{A_1}$,$\mathbf{A_2}$))=$\mathbf{A_1}$**.
- For all expressions $\mathbf{A_1}$ and $\mathbf{A_2}$, **cdr(cons($\mathbf{A_1}$,$\mathbf{A_2}$))=$\mathbf{A_2}$**.

Figure 11-9 proves that these combinator expressions have the desired properties.

As another example, the following combinator expression is the ***predecessor function,*** which returns one less than the single number presented as an argument, except for 0 which always returns 0:

predecessor=C (C (C I (S (B C (C I)) (S B) (C I) (K I)))
(C (C I (K I)) (K I))) K

where **predecessor** n=if n=0 then 0 else $n-1$. In this expression the leftmost **C** is the function, and it receives two of its three arguments from the long expression in ( ) and the trailing **K,** respectively.

11.4 OPTIMIZATIONS

As can be seen from Figure 11-6, it is absolutely necessary to find ways to shrink a caf derived from an arbitrary lambda expression. That this might be possible should be obvious from the figure. There are multiple copies of identical subexpressions all over, such as **(S(K K)(K S))**. Either we find shorter equivalents to such expressions, or we invent new combinators to take their place. While the latter is certainly feasible (cf. the ***supercombinators*** discussed later in this chapter), the former should be employed to the maximum extent first.

As an example of the power of such optimizations, consider an expression of the form **(S(K $\mathbf{E_1}$)I)**, where $\mathbf{E_1}$ is an arbitrary caf. By simple reductions, we find that this expression is equivalent to the caf $\mathbf{E_1}$ alone (i.e., for any caf $\mathbf{E_2}$, $\mathbf{(S(K\ E_1)I)E_2 \Rightarrow (K\ E_1)\ E_2\ (I\ E_2) \Rightarrow E_1\ E_2}$). Thus, whenever the compilation process finds that it has generated an expression of the form **(S(K E)I)**, it can substitute **E** in its place.

Figure 11-10 gives a table of similar easily proven optimizations. If used extensively during the compilation of a lambda function into a caf, such optimizations can very dramatically reduce the size of the resulting caf. Figure 11-11 gives some examples based on Figure 11-4; Figure 11-12 gives a more powerful one—the translation of the successor function that blew up before. The result from this latter example is the simple caf **S B,** something much more in tune with our notion of successor as a primitive function.

for any cafs x,y

1.	S (K x) (K y)	= K (x y)
2.	S (K x) I	= x
3.	S (K x) y	= B x y
4.	S x (K y)	= C x y
5.	S (B K x) y	= S′ K x y
6.	S x I y	= W x y
7.	S K x y	= I x
8.	B (K x) y	= B′ K x y
9.	B x I y	= x y
10.	C (B K x) y	= C′ K x y
11.	C K x y	= I x

FIGURE 11-10
General combinator optimization rules.

As a proof that **S B** does in fact implement the addition of 1 to an integer, consider that

$$\mathbf{Z_{i+1}\ a\ b \Rightarrow a(a(\ldots a(a\ b)\ldots)}\qquad \text{with i+1 a's}$$

and that

$$\begin{aligned}\mathbf{(S\ B\ Z_i)\ a\ b} &\mathbf{\Rightarrow S\ B\ Z_i\ a\ b \Rightarrow B\ a\ (Z_i\ a)\ b}\\ &\mathbf{\Rightarrow a(Z_i\ a\ b)}\\ &\mathbf{\Rightarrow a(a(a\ldots a(a\ b)\ldots)}\qquad \text{with i+1 a's}\end{aligned}$$

```
Example: (λx|+ 1 x) = S(S(K +)(K 1))I
  → S(K(+ 1))I by Rule 1
  → + 1 by Rule 2

Example: (λxy|x) = S(K K)I
  → K by Rule 2

Example: (λx|+ (× x x) x) = (S (S (K +) (S (S (K ×) I) I)) I)
  → (S (S (K +) (S × I)) I) by Rule 2
  → (S (S (K +) (W × )) I) by Rule 6
  → (S (B + (W × )) I) by Rule 3
  → (W (B + (W × )) ) by Rule 6

Check: (W (B + (W × )) ) 3
  → (B + (W × )) 3 3
  → + ((W × ) 3) 3
  → + (× 3 3) 3 → 27
```

Note: Rule numbers refer to Figure 11–10. Examples should be compared with Figure 11–4.

FIGURE 11-11
Optimization of simple compilations.

```
(λxyz|y(xyz)) = "Successor Function"
  → [x]{[y](S (Ky) (S {S(Kx)(Ky)} I))}
  → [x]{[y](S (Ky) (S {K(xy)} I))} by Rule 1
  → [x]{[y](S (Ky) (xy))} by Rule 2
  → [x]{[y](B y (xy))} by Rule 3
  → [x]{ S(S [y]B [y]y) [y](xy))}
  → [x]{ S{S (K B) I} {S (Kx) I}}
  → [x]{ S B x} by Rule 2 (twice)
  → S (S [x]S [x]B) [x]x
  → S (S (K S) (K B)) I
  → S (K (S B)) I by Rule 1
  → S B by Rule 2

Check: S B x y z → B y (x y) z →y((x y) z) = y(xyz)
```

Thus: (λxzy|y(xyz)) = (S B) (*compare with Figure 11-6*)

FIGURE 11-12
A more complex example.

11.4.1 Recursion

Our first introduction to a lambda calculus implementation of recursion involved the **Y** combinator. To make a function **(λ*n*|E)** recursive, we rewrote it as

Y (λ*f*|(λ*n*|E))

where inside **E** we use *f* for recursive calls.

Expressing a recursive function in combinators follows directly; we convert the body of the function to combinator form, then abstract out the function name and precede the whole expression by **Y**, namely:

Y [*f*]([*n*]E)

Figure 11-7 diagrammed the conversion of the body of **factorial** tocombinatory form. Obviously, one could extract *f* from this as required above. Needless to say, this is painful. To again demonstrate the use of optimizations and shrink the expression, Figure 11-13 optimizes just Figure 11-7 using the **B** and **C** combinators. This figure also shows by example that applying the resulting expression to 3 delivers the expected results.

The next step is to abstract *f* from this expression. Figure 11-14 diagrams the major steps of the process (we used some optimizations to be

```
letrec f = (λn|(if (= n 0) 1 (× n (f (− n 1))))))
  → (S(S(S (K if) (S(S (K =) I) (K 0))) (K 1)) (S(S (K ×) I) (S (K f)
     (S(S (K −) I) (K 1)))))
  → (S(S(S(K if)(S = (K 0)))(K 1))(S × (S(K f)(S − (K 1))))))—rule 2
  → (S(S(S(K if)(C = 0)) (K 1)) (S × (S (K f) (C − 1))))—rule 4
  → (S(S(B if (C = 0)) (K 1)) (S × (B f (C − 1))))—rule 3
  → (S(C (B if (C = 0)) 1) (S × (B f (C − 1))))—rule 3

Consider (S(C (B if (C = 0)) 1) (S × (B f (C − 1)))) 3
  → (C (B if (C = 0)) 1) 3 (S × (B f (C − 1)) 3)
  → (B if (C = 0)) 3 1 (S × (B f (C − 1)) 3)
  → if (C = 0 3) 1 (S × (B f (C − 1)) 3)
  → if (= 3 0) 1 (S × (B f (C − 1)) 3)
  → if F 1 (S × (B f (C − 1)) 3)
  → (S × (B f (C − 1)) 3)
  → × 3 (B f (C − 1) 3)
  → × 3 (f (C − 1 3))
  → × 3 (f (− 3 1))
  → × 3 (f 2)
```

FIGURE 11-13
Optimizing factorial.

Y [f](S (C (B if (C = 0)) 1) (S × (B f (C − 1))))
→ Y (B (S (C (B if (C = 0)) 1)) [f](S × (B f (C − 1))))
→ Y (B (S (C (B if (C = 0)) 1)) (B (S ×) [f](B f (C − 1))))
→ Y (B (S (C (B if (C = 0)) 1)) (B (S ×) (S B [f](C − 1))))
→ Y (B (S (C (B if (C = 0)) 1)) (B (S ×) (S B (K (C − 1)))))
→ Y (B (S (C (B if (C = 0)) 1)) (B (S ×) (C B (C − 1))))

FIGURE 11-14
Completing a fully combinatory form of factorial.

proved as problems at the end of the chapter). The resulting combination expression is about as long as the original lambda description.

11.5 GRAPH REDUCTION

A combinator expression consists of expressions of constants (combinators, built-in functions, numbers,...). Reducing it involves finding something in the function position of an application and applying it. For functions that are combinators of arity n, this involves simply taking the next n subexpressions and rearranging them. When written as a linear string of characters (as has been done in all examples to this point), each reduction simply shuffles the character string form. For that reason it is often called ***string reduction.***

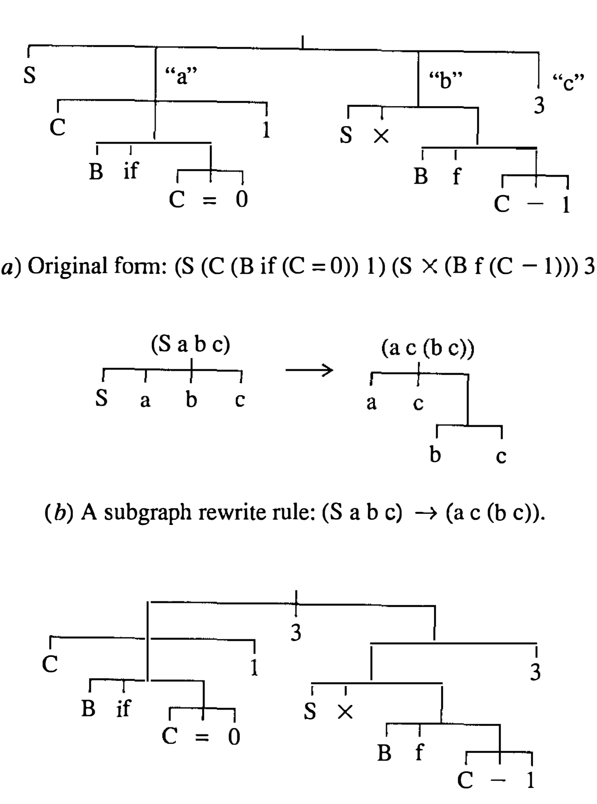
Consider what might happen if we arrange such expressions as graphs where each subexpression is the root of a subgraph, with the leftmost arc of the subgraph pointing to the function subexpression, and the arcs to the right representing the argument expressions, in order. Then the leaves of the graph are all the constants in the expression (see Figure 11-15).

A ***graph reduction*** of such a graph involves finding any subgraph where the leftmost subexpression (the function) is a constant, and the right branches going back up the subgraph are its arguments, and replacing the entire subgraph by the result of the application. For function leaves that are combinators, this involves ***rewriting*** the subgraph with a new one where the arguments have been rearranged.

The following subsections discuss graph reduction in more detail.

11.5.1 Curried Graphs

Although the concepts behind Figure 11-15 seem simple enough, trying to develop a general algorithm that performs this automatically has a basic problem. All the arguments for any particular combinator may not be in the subgraph containing the combinator as its function. For example, Figure 11-15 was written so that all the arguments for the uppermost **S** (the ones labeled "a," "b," and "c") are at the same level of the same subgraph.



(*a*) Original form: (S (C (B if (C = 0)) 1) (S × (B f (C − 1))) 3).

(*b*) A subgraph rewrite rule: (S a b c) → (a c (b c)).

(*c*) After rewrite: ((C(B if (C = 0))1)3((S × (B f (C − 1)))3).

FIGURE 11-15
Combinator expression in graph form.

In most circumstances, however, it is possible for two, or even all three, arguments of **S** to be in subgraphs other than the one containing **S**. To see this, consider Figure 11-15(*c*). Here only two of the three arguments that **C** needs are in the same subgraph. The third, 3, is in the next subgraph up.

Needless to say, to devise an algorithm for automating such graph reductions would be somewhat messy. We would have to know the arity of the function and count the number of arguments in the same subgraph.

One way to avoid this complexity is to standardize our representation of applications back to that for the original lambda calculus, namely, every application has exactly one function expression and one argument expression. If the function can accept n arguments, the result of the first application is a ***curried function*** of n−1 arguments. This function expression can in turn accept exactly one more argument and produce a curried function of n−2 arguments. The process repeats until all the arguments are satisfied.

In terms of a graph, the original function (the leftmost constant) is paired with the first argument in a binary graph. Depending on the complexity of this argument, the rightmost branch of this graph can point to either a constant itself or to a subgraph representing an expression.

If there is a second argument, it forms the right subgraph of a new binary graph, where the left subgraph is that constructed from the graph representing the curried function. The process repeats for each argument. Thus, for a function of n arguments, the corresponding graph is a binary graph of n levels, where each sublevel representing the function part is spawned from the left branch of its parent. Each right branch represents an argument.

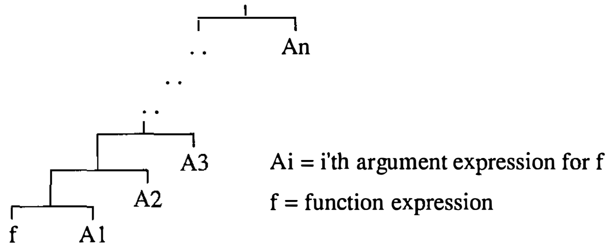
Of course, if an argument of a subgraph is itself an expression, the equivalent right branch in the graph equivalent is a binary subgraph constructed according to identical rules.

Figure 11-16 diagrams the general case for an expression with n arguments. Figure 11-17 handles the specific case of the expression from Figure 11-15. Figure 11-18 diagrams yet another case, this one for a lazy evaluation.

To repeat, the key characteristic of graphs constructed from cafs in this fashion is that if one travels down the left branches of a subgraph until a constant is reached, then that constant represents a function. If the arity of that function is n, then the rightmost branch is its first argument, and the rightmost branches of the n−1 parent nodes represent the other n−1 arguments.

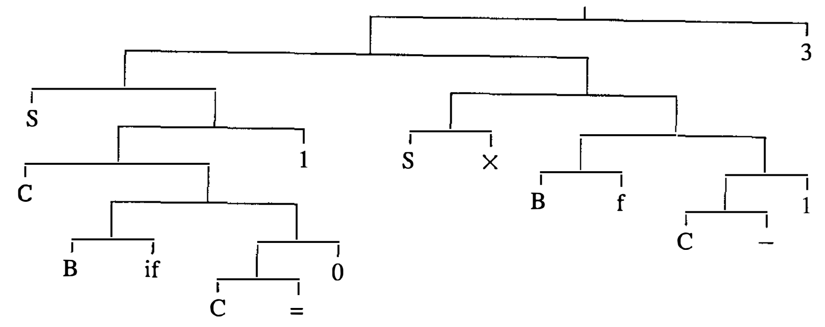
11.5.2 Subgraph Rewrite Rules

Any constant found as a left branch leaf can be treated as a function in some subexpression and as such is a candidate for reduction. We can actually perform it if there are enough arguments in the graph above it arranged as pictured in Figure 11-16. Reducing the resulting subgraph expression involves replacing the entire subgraph with a new subgraph representing the results.



(f A1 A2 A3 ... An)

FIGURE 11-16
General n-argument curried graph.



((S ((C ((B if) ((C =)0)) 1) ((S ×) (((B f) ((C –) 1)))) 3)

FIGURE 11-17
Graph in curried form.

letrec integers(n) = cons(n, integers(1 +n))

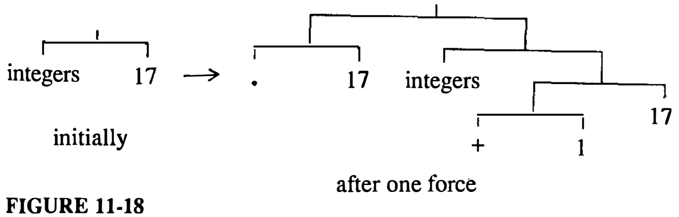


FIGURE 11-18
Lazy graph evaluation.

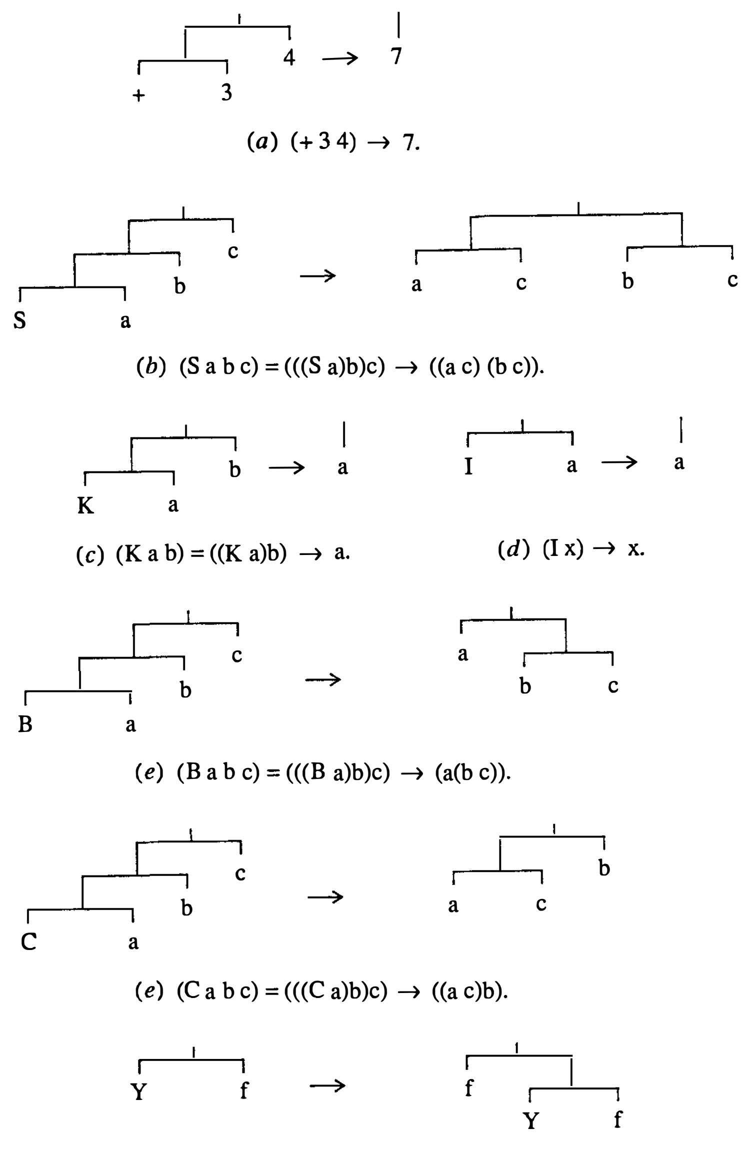
For built-in functions this replacement is the value of the application. For combinators the replacement is often a new graph constructed from a reordering of the original argument subgraphs. Figure 11-19 diagrams some samples of these.

After either of these replacements we are free to look at the graph again for another function leaf to apply. This again involves diving leftmost until a constant is found.

11.5.3 Graphical Forms for Recursion

Recursion can be handled by the **Y** combinator as pictured in Figure 11-19(*f*). This is clean and straightforward, but suffers from the problem that one must abstract out of the body of the recursive function the identifier being used for the function's name. Again, this is not difficult, but it often leads to increased combinator code to reshuffle expressions and get the identifier out and its replacement actual argument back in.

An alternative form fits very naturally into our graphical representation (Figure 11-20). Instead of abstracting out the function name, we leave it in, but handle it differently in the graph. Basically, we connect



(a) (+ 3 4) → 7.

(b) (S a b c) = (((S a)b)c) → ((a c) (b c)).

(c) (K a b) = ((K a)b) → a.

(d) (I x) → x.

(e) (B a b c) = (((B a)b)c) → (a(b c)).

(e) (C a b c) = (((C a)b)c) → ((a c)b).

(f) (Y f) → f (Y f).

FIGURE 11-19
Rewrite rule for basic combinators.

what was the "leaf" for each usage of the function name via an arc back to the root of the graph that represents the function's body in caf form. Now each time we wish to use an identifier as a function, we replace it by a copy of the entire caf form of the body (which still contains internal pointers back to its root). This copying can be repeated as often as necessary.

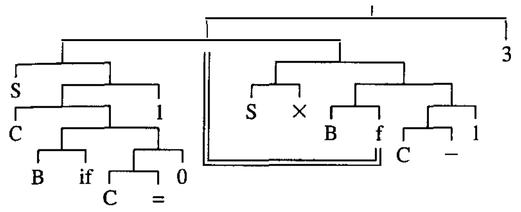


FIGURE 11-20
Graph in curried form.

11.5.4 Evaluation Orders and a Reduction Algorithm

As with pure lambda calculus, there are a variety of ways to decide which reduction to do next in a binary combinator graph. In ***normal-order reduction*** we start at the root and go left at each internal node until we arrive at a leaf node. The constant at the leaf is the function to be applied. Picking off the right branches of the n parent nodes of this leaf provides the n arguments needed for its complete evaluate. After replacing this entire n−1 level subgraph, the process is repeated.

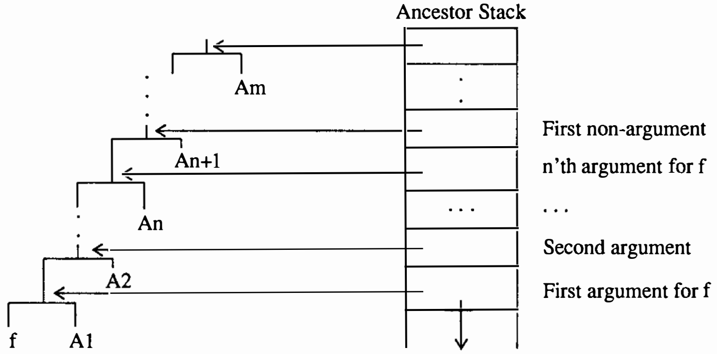
In ***applicative-order reduction*** any leaf on the left of some internal node may be used for a function in an application. In particular, we would like to take those spinning off of the right branches for the arguments of a function chosen as above. Reduction is similar; the argument subgraph is replaced by the result of the evaluation.

The problem with the latter technique is that very often not all the arguments to a function are available yet, and if we were to replace that subgraph with its reduction, that replacement would have to be a curried application, which in combinator theory is the original expression itself! For example, the second **C** from the left of Figure 11-17 is in a left branch position of a subgraph, but if we go up the graph we find only two arguments identified ("=" and "0") before the subgraph dies. Replacing this subgraph is possible, but without expanding **C** into its lambda equivalent, doing what partial substitutions are possible, and returning a specialized lambda function [(**λ**z|= z 1) in this case], there is nothing to do other than return the original subgraph.

Besides these, the standard concerns about infinite loops and duplicate evaluations also hold.

Based on these comments, the standard algorithm for graph reduction is normal-order evaluation using a ***left ancestors stack*** (cf. Figure 11-21). We start at the root and go left. If the node there is not a leaf, we push onto the top of a stack a reference to that node, and go down and left one more level. The process is repeated until a constant leaf is reached. For reasons to be described later, what gets pushed on the stack is the node just investigated, not the argument found in its cdr.

At the point where the leftmost node is a constant, the top n references on this stack are to graph nodes whose right-hand branches refer-



Assume f has arity n.

FIGURE 11-21
A left ancestor stack.

ence the n arguments needed by the function. Further, the (n+1)-st entry on the stack is to the node whose right branch contains the first argument not needed by this application.

Figure 11-22 gives a set of abstract programs to perform such reductions on graphs built as s-expressions. The main function is **look-left,** which accepts two arguments, the current subgraph left to be traversed, and a list of parents of this subgraph. As long as the node is not a function constant, then the current node is pushed to the list, and **look-left** is called on the node's left subgraph. When a constant is encountered, the function **apply** gathers up the proper arguments from the *stack,* reorders them appropriately into a new subgraph, and recalls **look-left** on the new subgraph.

Two auxiliary functions used to access the argument stack are **get-arg,** which gets the n-th argument from the stack, and **drop-args,** which drops n arguments from the stack. The **get-arg** function includes a **cdar** of the stack to remove an argument. This reflects what was **cons**ed onto the graph. It corresponds to normal evaluation.

The stack generated inside of **apply** is built from the original stack of arguments with the n arguments consumed by the selected function deleted. In most cases, this new stack is given to another call to **look-left** for processing, along with the subgraph resulting from the application. **look-left** will look down through this subgraph until a new constant left leaf is found. During this pass, nodes holding arguments are pushed onto the stack as before. Thus, when it comes time to process the new function, there is no distinction as to where the arguments came from; they are all simply in order on the stack.

Notice also that for combinators no evaluation of arguments is performed; argument subgraphs are simply shuffled around as needed. This is not true for many built-ins, such as the "+" shown here. Both arguments must be reduced using the same **graph-reduce** function before the replacement value can be computed.

```
graph-reduce(graph) = look-left(graph, nil)

look-left(root, stack) = "cons nodes to stack until leftmost
an atom, and return resulting stack" =
  if atom(root)
  then if is-a-function-constant(root)
       then apply(root, stack)
       else root
  else look-left(left-branch(root), cons(root, stack))

apply(function, stack) =
  if function = I
  then look-left(get-arg(1,stack), drop-args(1,stack))
  elseif function = K
  then look-left(get-arg(1,stack), drop-args(2,stack))
  elseif function = S
  then look-left(cons(cons(get-arg(1,stack), get-arg(3,stack)),
                      cons(get-arg(2,stack), get-arg(3,stack))),
                 drop-args(3,stack))
  elseif ...
  elseif function = "+"
  then let left-arg = graph-reduce(get-arg(1,stack))
       and right-arg = graph-reduce(get-arg(2,stack)) in
           (left-arg + right-arg)
       elseif function = "if"
       then if graph-reduce(get-arg(1,stack))
            then look-left(get-arg(2,stack), drop-args(3,stack))
            else look-left(get-arg(3,stack), drop-args(3,stack))
  elseif ... "handle other combinators and builtins"

get-arg(n,stack) = "get n'th argument from top of stack"
  if n = 1
  then right-branch(car(stack))
  else get-arg(n - 1,cdr(stack))

drop-args(n,stack) = "drop n arguments from stack"
  if n = 0
  then stack
  else drop-args(n - 1,cdr(stack))
```

FIGURE 11-22
A left ancestor reduction algorithm.

11.5.5 Avoiding Argument Reevaluation

An interesting observation about the algorithm of Figure 11-22 is that it does not modify the actual data structures, whatever they are, that represent the graph. (Of course, as implied by the construction of the programs, an s-expression format where the left-branch is the car of a cons cell and the right-branch is the cdr is a natural fit.) The variable *stack* that is passed around is bound to a list of subgraph nodes representing intermediate points, but no attempt is made to go back and actually modify the branch fields of parent nodes when a subgraph is modified. In fact, even though new subgraphs are created every time a function is performed, no prior graph is ever actually modified.

This approach exhibits all the advantages and disadvantages of normal-order evaluation. No argument is ever evaluated before it is needed, and the algorithm itself is easily verified as giving correct operation.

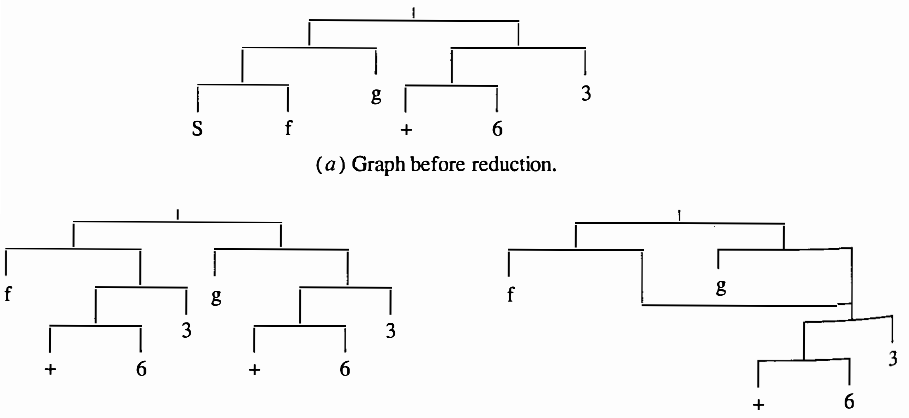
However, it is possible to evaluate the same subgraph more than once. This is because when we copy arguments, we are copying pointers to graphs that never change after they are built. Then, each time the value of an argument is needed, the corresponding graph is reevaluated afresh. This is almost equivalent to simply copying the argument graph as required. As an approach to sharing objects it is often called ***structure copying***.

As an example, consider Figure 11-23. After the **S** reduction there are essentially two copies of the subexpression (+ 6 3). As reduction goes forward, each copy is evaluated separately.

Now consider what happens if, when subgraphs are first evaluated, we destructively modify them so that *all* future references to the same subgraph see the same evaluated result. This is exactly what we earlier termed ***lazy evaluation***.

To implement this we must keep exactly one copy of each subgraph that will be modified as evaluation goes on and "share" this copy among all references to it. This is termed ***structure sharing***.

Figure 11-24 diagrams as an example a potential representation of the expression (**S E1 E2 E3**). The cell marked with a "*" is the uppermost cell associated with the expression. After reduction, the new subgraph should be rooted in this cell's parent car field.

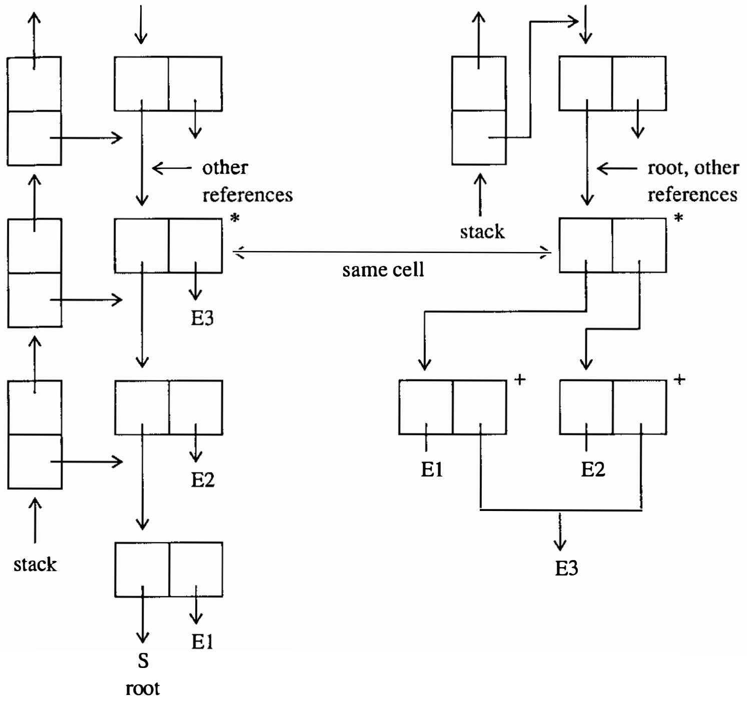


(*a*) Graph before reduction.

(*b*) Structure copying.

(*c*) Structure sharing.

FIGURE 11-23
Structure copying versus structure sharing.



(*a*) Before reducing S.

(*b*) After reducing S

* This cell is modified.
+ These cells are new.
All other cells are unchanged.

FIGURE 11-24
Structure sharing reduction of S combinator.

There are two ways of attacking the problem of substituting a new subgraph in there. First is to allocate a new cell and modify the pointer in the parent of the "*"-ed cell to point to the new graph. This does not work unless we also modify all other pointers to the old data structure. The second, and proper, choice is to leave the topmost cell of the subgraph in place, and modify its contents appropriately. All current references to the subgraph will thus in the future see the new reduced version. For the **S** combinator this involves allocating two new cells to hold pointers to the appropriate subexpressions (**E1 E3**) and (**E2 E3**), and rewriting the car and cdr of the *-ed cell to point to them.

Now, looking back at Figure 11-22, we see why the parent of an argument node, rather than the argument, is stacked. This allows access to node cells whose car should be modified.

For built-ins like "+," a similar approach works. The equivalent of the "*"-ed cell in Figure 11-24 is modified to contain the final value of the operation. All future references will see it without any need for flags, recipes, etc.

There is one problem, however, with combinators like **I** and **K**, which simply return one of their arguments. On the surface this sounds like "+," but the problem arises when the argument being returned is a pointer to another subgraph. A single pointer does not fit directly into a cell that was designed for a pair.

There are two solutions. First is to return **(I E)**, where **E** is the argument being returned. This works, but it requires all sorts of special tests to handle the **I** whenever it appears.

The alternative is to change the "*"-ed cell to have a tag of ***invisible pointer*** with a pointer to **E** as its value. This tag is essentially the same as mentioned earlier for Baker's garbage collection algorithm, and was used in several of the LISP machines we discussed. Although it uses up some extra storage, it is a much cleaner solution.

11.6 SUPERCOMBINATORS

(Hughes, 1982; Peyton-Jones, 1987; Diller, 1988)

The combinators discussed so far are relatively primitive, low-level functions that require a compiler to laboriously build up some combination that matches a particular expression. In contrast, a ***supercombinator*** is a single combinator that is specially built by a compiler to match optimally the outer level of some function being compiled. When executed, the result is a greatly reduced number of data shuffles. Arguments are moved only when necessary, and then exactly into the positions desired.

Such a function represents a potentially huge increase in performance. Of course it does complicate the basic reduction algorithm. Code (sometimes microcode) must be built dynamically to handle each new supercombinator, and then loaded and invoked as needed.

The following subsections address a more formal definition for supercombinators, a description of the key objects looked for by the compiler—namely, ***maximally free expressions***—optimal orderings of arguments for these supercombinators, and some other optimizations.

11.6.1 Defining and Finding a Supercombinator

Formally, the expression $(\mathbf{\lambda} x_1 \ldots x_n | \mathbf{M})$ is a supercombinator of arity n if:

- **M** is not a lambda function;
- And any subexpression of **M** which is a lambda function is itself expressible as a supercombinator.

This means that **M** is a caf involving only built-ins, combinators, other supercombinators, and the identifiers $x_1 \ldots x_n$. Applying the function $(\mathbf{\lambda} x_1 \ldots x_n | \mathbf{M})$ to a series of n argument expressions will then result in a shuffling of those expressions, with additional constants thrown in. This is again a caf.

There are some important differences between a supercombinator and a regular combinator. First, these supercombinators are built from user-provided functions, not defined beforehand. Next, unlike the prior combinators, the result of applying a supercombinator can explicitly insert new constants into the expression (in fact, most real supercombinators introduce several). Finally, optimized supercombinators order the arguments they process to delay as long as possible their evaluation and maximize their potential reuse. This will permit a very useful form of lazy evaluation.

A very simple example of a supercombinator is:

$$\boldsymbol{\alpha} = (\mathbf{\lambda} z_1 z_2 | \text{if } (= z_1\ z_2))$$

This caf takes four arguments, compares the first two, and then selects one of the remaining for continued evaluation. Its rewrite rule is:

$$\boldsymbol{\alpha}\ \mathbf{M}_1\ \mathbf{M}_2 \Rightarrow \text{if } (= \mathbf{M}_1\ \mathbf{M}_2)$$

Its typical use in a caf for some specific function might look like:

$$(\boldsymbol{\alpha}\ 0\ n\ 1\ (f\,(-\ n\ 1)))$$

As another example, we might build a supercombinator of three arguments with the following rewrite rule:

$$\boldsymbol{\alpha}\ \mathbf{M}_1\ \mathbf{M}_2\ \mathbf{M}_3 \Rightarrow (\mathbf{M}_1\ (\mathbf{car}\ \mathbf{M}_3)\ (\mathbf{M}_2\ (\mathbf{cdr}\ \mathbf{M}_3)))$$

The basic algorithm for finding such supercombinators starts with the function being compiled, such as

$$(\mathbf{\lambda} x_1 \ldots x_n | \mathbf{M})$$

and works inside out, starting with x_n and working backward to x_1. At each step, the innermost expression $(\mathbf{\lambda} x_k | \mathbf{M}_k)$ has a new supercombinator built for it, and the result is used to construct an $\mathbf{M}_{k-1}$ caf for the next iteration. For each such supercombinator, the equivalent rewrite rule must be constructed and saved for use when the function is evaluated.

The actual steps in converting $(\mathbf{\lambda} x_k | \mathbf{M}_k)$ to a caf called $\mathbf{M}_{k-1}$ involving a new supercombinator $\boldsymbol{\alpha}_{k-1}$ are as follows:

1. Find all the subexpressions of $\mathbf{M}_k$ that are independent of x_k. These are the ones that have no free occurrences of x_k in them. They are called ***free expressions.***
2. Pick the most encompassing of these expressions (those not part of other free expressions) as the ***maximal free expressions.*** Assume that they are $\mathbf{N}_1, \ldots, \mathbf{N}_m$.
3. Invent m new identifiers $z_1, \ldots, z_m$ not in use anywhere.

4. Define supercombinator $\boldsymbol{\alpha}_k=(\boldsymbol{\lambda} z_1\ldots z_m x_k|\ [z_1/\mathbf{N}_1,\ldots,z_m/\mathbf{N}_m]\mathbf{M}_k)$.
5. Until x_1 has been handled, repeat the process with $(\boldsymbol{\lambda} x_{k-1}|\mathbf{M}_{k-1})$ where $\mathbf{M}_{k-1}=\boldsymbol{\alpha}_k\mathbf{N}_1\ldots\mathbf{N}_m$.

Figure 11-25 diagrams the standard example of this process used by the chief inventor of supercombinators, John Hughes.

11.6.2 Free Expressions

The key step in the above algorithm is the location of the ***free expressions*** in $\mathbf{M}_k$. These expressions are independent of x_k, and thus can be "factored out" of the expression that will be converted into a supercombinator $\boldsymbol{\alpha}_k$ and made into an argument for it.

Formally, $\mathbf{N}_i$ is a free expression of $\mathbf{M}_k$ if both

- $\mathbf{N}_i$ is a subexpression of $\mathbf{M}_k$.
- The value of $\mathbf{N}_i$ is guaranteed to be independent of x_n, that is, a caf involving only constants, built-ins, of the identifiers x_1 through x_{k-1}.

For example, the free expressions in

$(\boldsymbol{\lambda} s|$if (= n 1) (**car** s) (**f** (− n 1) (**cdr** s)))

are (in rough order of size)

if, −, n, 1, **car**, **f**, −, n, **cdr**
(= n), (− n)
(= n 1), (− n 1)
(if (= n 1)), (**f** (− n 1))

Note that (f (− n 1) (**cdr** s)) is *not* a free expression because it is not independent of **s.** Also, the last of these free expressions are not subexpressions of any bigger free expressions, and are thus ***maximally free expressions*** (we do not count simple constants as maximally free).

```
index(n,s) =
  if n=1
  then car(s)
  else index(n − 1,cdr(s))
```

Lambda Form: Y (λfns|if (= n 1) (car s) (f (− n 1) (cdr s)))

Supercombinator form: Y α_2 WHERE

α_2 = (λz_3n| α_3 (if (= n 1)) (z_3 (− n 1))

α_3 = (λz_1z_2s| z_1 (car s) (z_2 (cdr s)))

FIGURE 11-25
Hughes' original supercombinator example.

Factoring these expressions out of the body of the function and replacing them by new identifiers yields an expression identical to the first:

$(\boldsymbol{\lambda} z_1z_2s|$ (z_1 (**car** s) (z_2 (**cdr** s)))) (if (= n 1)) (**f** (− n 1))

The first term here is the supercombinator. It is a pure function with no free variables, which, if given cafs as arguments, produces a caf result. The caf arguments in this case are the Ni's that were factored out and replaced by variables. Note also that the new identifiers are added ***in front of*** the identifier s which is being processed. This places the actual argument to be bound to s ***at the right end*** of the argument list, exactly where the eta rule wants it.

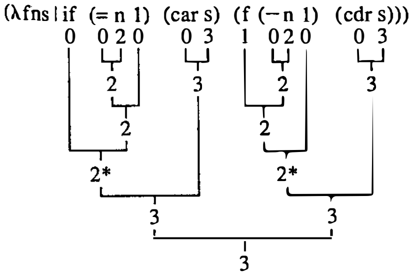
11.6.3 Identifying Maximally Free Expressions

There are several algorithms for identifying maximally free subexpressions from an arbitrary expression. One of them in particular is both simple and will help in some later optimizations. In this algorithm we assume that the identifiers in the overall function being compiled are read from left to right, and are each given a number (called its ***label***) to be used in further processing. Thus, in $(\boldsymbol{\lambda} x_1\ldots x_n|\mathbf{M})$, x_i is assigned a label of i.

Basically, the algorithm uses these labels to assign a number (called its ***level***) to each subexpression **N** of $\mathbf{M}_k$ as follows:

- If **N** is a constant or built-in, assign it a level of 0.
- If **N** is an identifier x_i with label i, assign it a level of i.
- If **N** is an application, assign it a level equaling the greater of the levels of the function and argument subexpressions.

Figure 11-26 diagrams a sample assignment. In general, any subexpression with number less than k is independent of the identifiers labeled k or higher. Thus, when processing $(\boldsymbol{\lambda} x_k|\mathbf{M}_k)$, any subexpression whose level is less than k and whose next bigger expression has level k is a maximally free expression.



* = maximally free for s.

FIGURE 11-26
Sample numbering.

11.6.4 Ordering the Maximum Free Expressions

In Figure 11-25 there were two subexpressions factored out and made arguments of the supercombinator α_3. Notice that to this point the order of these arguments has not been governed by any particular functional concerns. We could have derived α_3 from

$$(\lambda z_1 z_2 s|\ (z_2\ (\textbf{car}\ s)\ (z_1\ (\textbf{cdr}\ s))))\ (\text{f}\ (-\ n\ 1))\ (\text{if}\ (=\ n\ 1))$$

What difference is there in these two orderings? In the first one both arguments are functions of n, while only the first argument is still dependent on f. This is reversed in the second formulation.

In general, assume that we have

$$(\lambda x_1 \ldots x_{k-1}|\ \alpha_k N_1 \ldots N_i\ N_{i+1} \ldots N_m)$$

For any j we want $N_1 \ldots N_i$ to be all, and only, maximally free expressions for x_j, in particular for k−2. This will lump all, and only, those expressions that are dependent on x_{k-1} to the far right, where they can be most easily dealt with. More precisely, we want to order the N_j's so that for each x_i, all N_j's that depend on it occur before any that depend on any x_{i+1} through x_{k-1}.

If the labeling algorithm described above for maximally free variables has been used, we can use these same numbers to order them. Basically, we sort the maximally free subexpressions and arrange them as arguments to the eventual supercombinator by level, with the lowest to the left. Once this ordering has been imposed, the use of the z_i's inside the body of the supercombinator is specified.

One further optimization will occur if, within this set of maximally free subexpressions, there is exactly one at some level, and that one is a simple identifier. This will simplify some combinator later on by the application of a ***gamma optimization,*** namely,

$$((\lambda x_1 \ldots x_i|\ \alpha_k N_1 \ldots N_{m-1}\ x_i)$$

can be replaced by

$$((\lambda x_1 \ldots x_{i-1}|\ \alpha_k N_1 \ldots N_{m-1})$$

11.6.5 Completing the Example

Figure 11-27 summarizes all these steps as applied to Hughes' demonstration.

11.7 COMBINATOR MACHINES

(Clarke et al., 1980; Treleaven et al., 1982)

Combinators are for the most part simple enough to correspond to basic instructions in a more or less conventional computer. This section describes several attempts to implement such machines. Although their impact has been primarily on the advanced research community, there is significant promise, particularly for the G-Machine, and it would not be surprising to see offshoots show up in future high-performance machines.

1. Original Function: $(\lambda fns|\text{if}\ (=\ n\ 1)\ (\text{car}\ s)\ (f\ (-\ n\ 1)\ (\text{cdr}\ s)))$
2. First Step: $(\lambda s|\text{if}\ (=\ n\ 1)\ (\text{car}\ s)\ (f\ (-\ n\ 1)\ (\text{cdr}\ s)))$
    - Two mfes of level 2: $N_1 = (\text{if}\ (=\ n\ 1))$ and $N_2 = (f\ (-\ n\ 1))$
    - Select order N_1, N_2
    - $[z_1/(\text{if}\ (=\ n\ 1)),\ z_2/(f\ (-\ n\ 1))](\text{if}\ (=\ n\ 1)\ (\text{car}\ s)\ (f\ (-\ n\ 1)\ (\text{cdr}\ s)))$
      $= \alpha_3 = (\lambda z_1 z_2 s|\ z_1\ (\text{car}\ s)\ (z_2\ (\text{cdr}\ s)))$
3. Result $= (\lambda fn|\alpha_3\ (\text{if}\ (=\ n\ 1))\ (f\ (-\ n\ 1))$
4. Only free expression f − result is
    - $(\lambda f|(\lambda z_3|\alpha_3\ (\text{if}\ (=\ n\ 1))\ (z_3\ (-\ n\ 1)))\ f)$
    - Or: $\alpha_2 = (\lambda z_3|\alpha_3\ (\text{if}\ (=\ n\ 1))\ (z_3\ (-\ n\ 1)))$
5. Result: $(\lambda f|\alpha_2\ f)$
6. Simplify further to: α_2

FIGURE 11-27
The full example.

11.7.1 The SKIM Machines

(Turner, 1979b; Stoye et al., 1984)

The first attempt to use combinators as the basic instruction set architecture of real hardware began in the late 1970s with the remicrocoding of a conventional computer to perform graph reduction using the ***left ancestor stack*** reduction algorithm. The basic instructions were **S, K,** and **I,** plus the other major combinators of Figure 11-8. This implementation was followed by another that used list pointer reversal instead of a separate stack of pointers for the arguments.

For obvious reasons these machines were called the ***SKIM machines.***

SKIM I The original SKIM machine was an existing conventional computer that had writable microcode control store. A left ancestor stack graph reduction interpreter was written in this microcode, along with direct microcode support for the basic combinators **S, K, I, Y, C, B, U,** and the standard integer functions for arithmetic, comparison, etc. A compiler translated programs written in SASL (a functional language similar to our abstract language) into optimized combinator code suitable for loading into this machine.

The memory for the machine looked much like our standard list-oriented memory. Memory was a pool of cells that could be either a single integer or a pair of pointers. A program consisted of a graph built from such cons cells, where in each cell the car points to the function subexpression and the cdr points to its first argument subexpression. A function corresponding to a combinator or built-in was represented by an integer which specified an appropriate microcode routine.

Execution involved dynamically modifying the memory representing the program structure. The basic microcode kept a stack (in memory) of pointers back into the current program. When a combinator was found as the leftmost object, the matching microcode was executed, leaving the program graph modified exactly as described earlier.

Copies of arguments were made by ***structure sharing,*** i.e., copying just a pointer to the original subgraph for an argument. This permitted an easy lazy evaluation, since once an argument was evaluated, its subtree was replaced by the reduced value. All other pointers to that subgraph would then see the reduced value without further work. As discussed previously, some of the combinators, such as **I** or **K,** required special attention to permit proper operation. This attention was by replacing roots of affected subgraphs by either the (**I.**"value") cell or by an invisible pointer.

Performance of this system was analyzed using several benchmarks (see Figure 11-28). Comparisons were to a conventional SECD Machine implementation. If we assume that a basic SKIM instruction is approximately as time-consuming as a basic SECD instruction (a rational first approximation), then, as shown, the SKIM machine seemed to take about twice as many instructions as the SECD implementation. If, however, we also look at the number of memory cells allocated dynamically by the programs during execution, a different result emerges. The SKIM machine often needs only half the cells needed by the normal SECD Machine, and up to one-tenth the number of cells needed by a fully lazy version. These latter numbers are of particular relevance because the SKIM implementation is inherently fully lazy. It simply does a better job of managing storage by dynamically modifying and reusing storage during execution. This translates into fewer garbage collections per program execution, and thus a significant performance advantage that is not obvious from the raw instruction counts.

SKIM II The success of the original SKIM experiment led to a second version that was also microcoded, but which was faster, with a bigger address space, more basic data types, etc. The major conceptual difference was in how the arguments were tracked. Instead of a stack of pointers to parent subgraphs, this machine simply reversed the car pointers in the graph as it went left and down. To handle this it kept two key pointers in machine registers. The *forw* register points to the cell containing the current function expression. The *back* register points to the cell whose car initially pointed to the *forw* cell (the parent of the *forw* cell).

	Instructions Executed		Memory Cells Allocated		
Program	SKIM	SECD	SKIM	SECD	Lazy SECD
Towers of Hanoi	3,067	1,488	3,131	1,488	10,428
Factorial	1,280	975	975	470	5,565
Twice	92	158	65	128	1,206

Source of Data:

FIGURE 11-28
SKIM versus SECD Machine performance.

When a function is found which is a subexpression (i.e., *forw* points to a cons cell), the following sequence of register transfers takes place:

1. *temp*←**car**(*forw*)
2. *back*←**rplaca**(*forw, back*)
3. *forw*←*temp*

This process reverses the pointer in the car of the function expression, making that cell the car of the argument stack. *forw* receives the original contents of this cell so that a continued left and down scan can be performed. The cells joined backward from the *back* pointer match exactly the cells on the standard left ancestor stack.

When a combinator or built-in is reached, the arguments for it can be found by going backward through the *back* register, much as with the basic stack algorithm, but without the extra memory overhead of the stack cells, and without the extra indirect reference through the stack cells to get to the actual program cells. Figure 11-29 diagrams a sample of this for the **S** combinator.

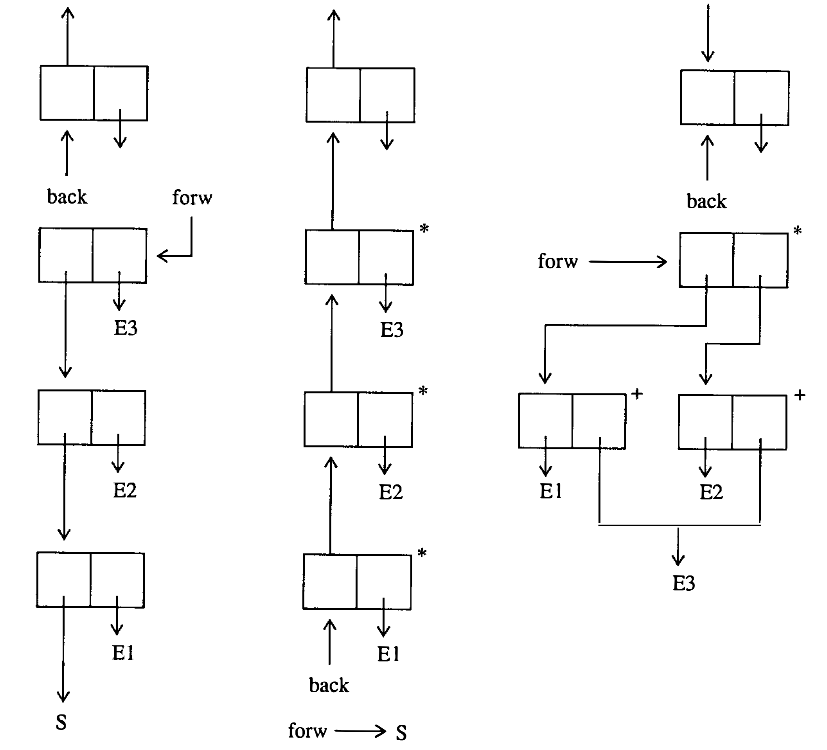
Another architectural innovation in SKIM II was the addition of a 1-bit ***reference count*** field to each cell. This count indicates whether or not the pointer stored in a field is the only pointer to the designated cell. It is set to 0 when the cell is initially built, left alone when a combinator simply rearranges its arguments with one copy of an argument, and set to 1 when a combinator copies an argument into more than one location.

The advantage of this is that the microcode for each combinator can quickly determine if a cell becomes free during a rewrite, and thus can be immediately reused without having to go through a storage reclaiming process. In actual programs this technique was highly successful, with 70 percent of the memory cells reclaimed directly by the combinator code without recourse to a garbage collector.

In overall performance these improvements gave something like a 4-to-1 increase over SKIM I, putting it in a very competitive position with conventional computers.

11.7.2 The CURRY Chip
(Ramsdell, 1986)

An example of the real simplicity that is possible when a machine is designed from the ground up to support combinators is the ***CURRY chip.*** This chip was part of a three-chip set (see Figure 11-30) built out of what



(*a*) At start.

(*b*) When S found.

(*c*) After reducing S.

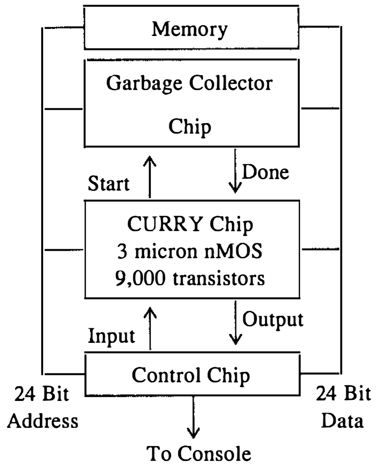
* These cell are modified.
+ These cells are new.
All other cells are unchanged.

FIGURE 11-29
Graph reduction with pointer reversal.

is today very-low-density VLSI. In approximately 9000 transistors, a complete CPU chip was built which performed as its instruction set basically the combinators accepted by the SKIM machines, but without microcode. A separate chip handled memory management and garbage collection in a fashion very similar to that for the ***SCHEME-79 chip*** described in Chapter 10. A third simple chip handled overall timing and interface with a test and control console.

At the slow (by current standards) clock rate of 0.4 MHz, the chip could perform about 18,000 reductions per second. This translates to about 13 machine cycles per instruction, which again is not bad at all for 9000 transistors.

The CURRY chip contained two registers, *fun* and *stack,* which correspond directly to the *forw* and *back* registers of SKIM II. Keeping track of arguments is done via pointer reversal in the program graph.



21	1	1	1
Value	X	Tag	GC

Basic Instructions:
S,K,I,B,C,S',C',Y
as before
plus:
P= (λxyz | zxy)
T= (λxy | yx)
J = (λx | I)
R= "Read"

FIGURE 11-30
CURRY chip set.

All functions (programs) for the CURRY chip have to obey the convention that they accept an input stream as their single arguments and deliver a single output stream as a result. Both of these streams are 1 bit wide, with 1-bit I/O paths connecting the CURRY CPU to the control chip, from which I/O to the outside world is actually performed.

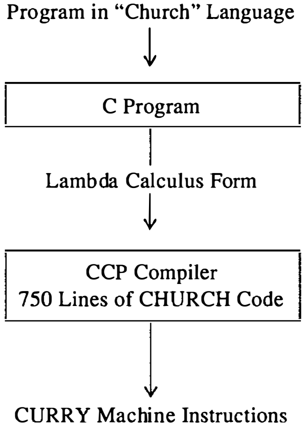
The **R** combinator performs the basic read of 1 bit. When expressed as a rewrite rule, this combinator would be used as follows (where **E** is the code for the function to receive the input):

R I E ⇒
P K (R I) E ⇒ **E K (R I)** ;if input is a 0 ⇒
P J (R I) E ⇒ **E J (R I)** ;if input is a 1

The code in **E** would accept as its first argument either **K** or **J** which would then internally select one of two expressions for evaluation. These expressions would match the case for an input of 0 or 1, respectively. The other argument to E, (RI), basically provides a stream function for the next bit.

Output is similar. The expression (**P K E**) outputs a 0 when **E** is a stream function, while (**P J E**) outputs a 1.

Finally, code for the CURRY chip was also produced with a real dash of simplicity (see Figure 11-31). Programs written in the ***CHURCH language*** were translated by a C program into something very akin to a pure lambda calculus form. This was then processed by a 750-line CHURCH program into a CURRY program, using the standard bracket abstraction algorithm plus a handful of optimizations. This program was



Compilation Rules:
Bracket Abstraction to S,K,I
+
Optimizations:
(I M)⟶M
(K M N)-->M
(S (K M))⟶(B M)
(S (M (KN)))⟶(C M N)
(S (B M N) O)-->(S' M N O)
(B I)-->I
(B M I)-->M
(B M (K N))-->(K (M N))
(C (B M N))-->(C' M N)
(J M)⟶M
(T M N)-->N M
(B C T)⟶P

FIGURE 11-31
Compiling code for CURRY.

then fed to a loader program which runs on the CURRY chip to load the graph into memory and start it.

Given that the basic compiler was itself a CHURCH program, the compiler could run on the CURRY chip itself.

As an experiment, Hughes' supercombinator algorithm was combined with a back-end translator which converted supercombinators into strings of conventional CURRY combinators. The resulting code was 2.5 percent smaller than code produced directly, but it required 11 percent more reductions.

11.8 THE G-MACHINE

(Auggustsson, 1984; Johnsson, 1983, 1984; Kieburtz, 1985; Peyton-Jones, 1987)

Just as with the SECD Machine (and with the WAM in Chapter 17), a highly successful method of increasing the performance of a graph reduction system is via "compilation" from a program to an optimized ***abstract machine.*** The architecture of such a machine is both an easy target from the input language and a simple translation away from a more conventional machine.

The ***G-machine*** is such an abstract machine. As with supercombinators, individual function definitions are compiled into unique code, although in this case it is a specialized sequence of G-machine instructions which perform the requisite graph reduction.

What is unique about the G-machine is that even though there is compiled code to execute function bodies, much of the code representing particular applications (i.e., the graphs and subgraphs) is built and executed dynamically. This permits lazy evaluation; individual applications can be constructed as graphs at one time and then executed at another.

The G-machine described here is an amalgam of several descriptions in the literature. It is so relatively new that each researcher in the area has added unique features to Johnsson's original ideas.

11.8.1 Major Data Structures

There are several major data structures for the G-machine:

1. A ***value stack*** contains the results of fully evaluated expressions that are to be processed by built-in functions. The value stack is a simple stack formed from consecutive locations, with no tags on cells.
2. A ***pointer stack*** (similar to the left ancestor stack) points backward into the graph at the arguments. This data structure is again a simple stack.
3. A ***heap*** holds the often dynamically built expression subgraphs, referenced by the pointer stack.
4. A ***dump*** holds return information for recursive calls to **eval,** namely, pairs of pointer stack and code addresses.
5. The ***program code area*** consists of a sequentially organized set of G-machine instructions.

Some implementations also define an ***environment*** to store operands, although for the most part the pointer stack takes its place.

The key data structure is the pointer stack. When a subgraph is being evaluated, its list of arguments is pushed on the stack as with graph reduction. The elements of this stack are pointers to tagged cells in the heap. Instructions exist to modify (as with **rplacd**) entries in this stack, permitting lazy evaluation techniques to change a pointer once and have many references to an object see an evaluated equivalent.

The value stack is a very conventional stack used to perform basic arithmetic and the like in a more efficient manner than for the SECD Machine. Objects here are untagged and are found in consecutive locations. Instructions exist to move them to and from the pointer stack, adding and deleting tags as necessary.

The heap is organized similarly to the list memory used throughout our previous machines. Cells are allocated from it as needed, with pointers to them embedded in the other major memory areas. Each cell contains a ***tag field*** and either one or two ***value fields.*** Unlike our previous organizations, however, the tag field often is more than a few bits. Instead it is an address to some native machine code that performs some specific function, such as a built-in. Specific examples include tags of ***INT, BOOL, CONS, AP, FUN,*** and ***HOLE.***

Cells of the first three types are like those in the SECD Machine. A cell with the fourth tag represents an APplication, with its first value subfield pointing to a subgraph for the function and the second pointing to the argument subgraph. A cell with the tag FUN has one value field which points to the code for some FUNction (built-in or supercombinator). The final tag, HOLE, has a value field representing a holder to be filled in later. It is used to construct cyclic data structures as is done with the SECD DUM instruction.

The G-machine's native instructions are stored in consecutive locations as with most conventional machines.

11.8.2 Typical Instructions

Figure 11-32 gives a brief description of some of the G-machine instructions that manipulate these data areas. The basic goal of these instructions is to manipulate a graph that is built on the heap. Instructions of the form MKxx dynamically build subgraphs consisting of cells with AP and FUN tags. These subgraphs are cons-like s-expressions whose car and cdr equivalents point to either other cells or compiled machine code for the body of a function.

The EVAL instruction is the most complex of the instructions, and does a leftmost dive through the topmost subgraph on the pointer stack, much as described earlier. Arguments are pushed onto the pointer stack until a basic function is found. The code for that function (as found in the cell with tag FUN) is then entered for execution. Return information is placed on the dump to handle nested calls and recursion.

Instruction	Execution
PUSHINT n	Push onto value stack a pointer to a cell with tag = INT and value = n.
ADD,SUB,MUL...	Replace top two elements of value stack by result.
EQ	Compare top two elements of value stack.
JTRUE,JFALSE	Conditional branch on basis of last compare.
GET	Transfer top of pointer stack to value stack, removing tag.
MKINT	Pop top element from value stack and push cell of tag INT and value equalling popped value onto pointer stack.
PUSH n	Push onto pointer stack a copy of the n'th pointer stack entry from the top (top is the 0'th)
PUSHFUN f	Push onto pointer stack a pointer to a cell with tag = FUN and value = pointer to code for f.
CONS	Pop top two objects from pointer stack, and replace with a pointer to cell with tag of CONS and car and cdr pointing to popped objects.
MKAP	Replace top two elements of pointer stack by a new cell of tag AP and value fields pointing to popped elements.
EVAL	Evaluate graph on top of pointer stack, and and push result to value stack. Save rest of pointer stack and return infomation on dump.
UPDATE n	Replace the n'th element on the pointer stack by a copy of the top element of value stack.
RET n	Return from a function, popping off the top n elements of the pointer stack.

FIGURE 11-32
Some typical G-Machine instructions.

Dynamically building code permits both curried functions and opens the door for lazy evaluation. Another instruction, UPDATE, completes this capability by replacing a prior argument subgraph with an evaluated result. All future references to this argument then see the computed value.

11.8.3 Example

As an example, Figure 11-33 lists the code for ***factorial.*** Initially we assume that the top item on the pointer stack (item 0) is to the *n* argument. This item is evaluated and compared to 0. If equal, that subgraph node is replaced by a "1." If not, a new subgraph is built, corresponding to $n \times \mathbf{fact}(n-1)$. This is then evaluated and the result rewritten as before.

In real systems a compiler converts a function into G-machine

```
fact(n) = if n=0 then 1 else n×fact(n-1)

G-MACHINE CODE: At entry, element 0 on pointer stack is n.
fact:      PUSH 0          ;Push 2nd copy of n to pointer stack
           EVAL            ;Evaluate top copy
           PUSHINT 0       ;Push a 0 to value stack
           GET             ;Transfer evaluated n to value stack
           EQ              ;Compare top pair of value stack
           JFALSE L1       ;Jump if not equal
; Continue here if n=0—pointer stack 0 = original n.
           PUSHINT 1       ;Push result 1 to value stack
L2:        UPDATE 1        ;Do graph rewrite of original n
           RET 1           ;Return, popping top element (1)
; Come here if n>0—pointer stack 0 = original n.
L1:        PUSH 0          ;Push a copy of n back on pointer stack
           EVAL            ;Evaluate it
           GET             ;Transfer it to value stack
           PUSHINT 1       ;Push a 1 to value stack
           PUSH 1          ;Push a copy of the n argument
           PUSHFUN sub     ;Push a pointer to subtract
           MKAP            ;Build subgraph code for (- n)
           MKAP            ;Build subgraph code for ((- n 1))
           PUSHFUN fact    ;Push a pointer back to fact
           MKAP            ;Build subgraph code for (fact (- n 1))
           EVAL            ;Evaluate the subgraph
           GET             ;Transfer result to value stack
           MUL             ;Multiply n by fact(n-1)
           MKINT           ;Move result back to pointer stack
           JUMP L2         ;Again modify the appropriate argument.
```

FIGURE 11-33
Sample code for G-Machine.

code, often with optimizations similar to those for the SECD machine. From here, each of the G-machine instructions is expanded into an appropriate sequence of the native machine instructions for the target machine the code should run on.

11.8.4 Performance Studies

Some early performance data is encouraging. An early software-based system that generated fully lazy, VAX machine code (Johnsson, 1984) was tested over several benchmarks (Fibonnachi, prime numbers, and insertion sort) and gave execution times within a factor of 2 (both slower and faster) of several other compiled versions, including a C-based one. Simulation of a specialized machine design which executes G-machine instructions directly (see Kieburtz, 1988) gave performance similar to that of a 3.6-mip computer with a relatively moderate machine cycle time and machine design.

Extensions of the G-machine to handle parallel graph reduction also have been demonstrated on an 18-way parallel processor (George, 1989). The major departures were in:

- The addition of a task pool to list runnable subgraphs
- No value stack—intermediate computations were kept on the equivalent of the pointer stack
- A flag used to determine if there are sufficient subgraphs to be worth spinning off parallel evaluation

On several benchmarks, performance was within a factor of 4 of that for a good compiler for a nonparallel, but largely equivalent, functional language running on a conventional sequential computer six times faster than the parallel one.

11.9 PROBLEMS

1. Apply the basic bracket abstraction process to the functions **not, and,** and **or** from Chapter 5. Assume that T and F are available as basic constants. Show by example that each translated sequence works as expected.
2. Compile (**λ***x*|+ *x* 1) into basic combinators, and check by applying to 3. Compare the code with Figure 11-4.
3. Repeat Problem 2 assuming that T and F are not available as built-ins.
4. Convert Figure 11-3 to a version that processes s-expressions of the form of Figure 6-5.
5. (Project) Upgrade the solution to Problem 4 to permit multiple arguments and assume built-in constants for numbers, booleans, standard arithmetic, and if.
6. Find pure lambda calculus equivalents to the combinator forms of **car, cdr,** and **cons.**
7. Show that the following are also valid definitions for **car, cdr,** and **cons**:

   cons = (C(B C(B(C I) K)))
   car = (C I(K I))
   cdr = (C I((B W(B B)) (K I)))
8. Express the primitive recursion combinator **R** (see Figure 11-8) as an abstract program.
9. Translate the following into combinator, optimized combinator, and supercombinator form:
   a. (**λ***m*|**λ***n*|**λ***fm*(*nf*))
   b. The lambda definition of addition
10. Show that the following equalities are true for any expressions **X** and **F**:
   a. **S F I=W F**
   b. **S F (K X)=C F X**
11. Using only the combinator **R,** the lambda definition of integers, and a "primitive" function **p***x*=*x*−1, write a lambda or combinator expression for the boolean function "=0."
12. Show all the steps involved in converting the first of the following definitions of factorial into the second. Then apply 3 to the second and derive the result.
   Y (λf|S [S {S(K if**)(S{S(K =)(K1)}I)} {K1}] [S {S(K×)I} {S[Kf][S(S(K−)I)(K1)]}])**
   Y (λf|S[C{B if (= 1)} 1] [S× {**B f** (**C** − 1)}])
13. Convert the definition of **member** to combinator form. Assume that primitives such as **eq, null, car, cdr,**...are available and do not need to be abstracted further. Try out your function on arguments 'a and '(b a c). List all steps.
14. Prove the following, assuming that **E** has no free occurrences of *n* in it, but that **A** might:
   a. [*n*]**E=(K E)**
   b. [*n*] **(S E A)=(B (S E)** [*n*]**A)**
15. Draw the binary graph form of Figure 11-12, and show the graph reductions until just before the start of the first recursion.
16. Compute an optimized combinator form for each of the following, and show the resulting binary graph:
   a. **Fib**(4) whererec **Fib**(*n*)=if *n* ≤ 1 then else **Fib**(*n*−1)+**Fib**(*n*−2)
   b. **Ack**(3,3), where **Ack** is ***Ackerman's function.*** (Chapter 1)
17. Write a version of **apply** from Figure 11-22 that performs lazy evaluation by list modification as pictured in Figure 11-24. Make sure you point out how you handled combinators like **K** and **I,** and built-ins like "+."
18. Convert the supercombinator α with the rewrite rule α **A B C** ⇒ **(A (car C) (B (cdr C)))** into a lambda function. Under what conditions might such a function be useful?
19. Convert the supercombinators α_2 and α_3 of Figure 11-25 into pure combinator form. Assume that the built-ins **car, cdr,** =, −, 1,...are constants.
20. Convert the following functions to supercombinator expressions:
   a. (**λ***xyz*|*y*(*xyz*))
   b. (**λ***fn*|if (2 *n*) 1 (+ (*f* (− 1 *n*) (− 2 *n*)))))
21. How would you compare the complexity of the G-machine to that of the SECD machine? Include consideration of memory management.

CHAPTER 12

OTHER FUNCTION-BASED COMPUTING SYSTEMS

Previous chapters have discussed the fundamentals of function-based computing (lambda calculus and combinators), the major language (LISP and its variants), and a spectrum of machine architectures used to implement it. This chapter includes short descriptions of a subset of other languages and architectures that have in some way represented a significant advance or introduced a unique set of features. Included are:

- Logo, a functional language taught to many school children
- FP, a mathematical system developed by John Backus whose emphasis is on higher-order functions and program-forming objects
- HOPE, a functional language that includes strong typing and pattern matching
- The FFP machine, a parallel processor where the nodes form a tree structure and string reduction of FP-like programs is the major execution mode
- ALICE, a parallel machine that spreads applications over multiple execution stations
- Rediflow, another parallel architecture that permits dynamic load balancing of runnable applications across arrays of processors
- Applicative caching, a technique for saving the results of a function application for later reuse.

Despite the number of such descriptions, the reader should not assume

that it is anywhere near exhaustive. There are many other examples for which good cases could have been built for inclusion here. In most cases the reason for their exclusion is that their major features are covered by one of the above. Space and availability of other references, however, also played a role.

For reference, a cross section of these alternative systems might include:

- ***ISWIM*** ("I See What I Mean"), a functional language which includes a combination of many abstract programming syntax features with FP and APL high-order functions and which has served as a starting point for many more recent languages (Burge, 1975).
- The ***GMD Machine,*** one of the first parallel functional machines actually built, which employed string reduction and a method of load spreading based on "tickets" (Berkling, 1975; Kluge, 1983).
- ***SASL,*** another language that looks much like abstract programming and was the foundation for many present-day languages (Turner, 1979b).
- The ***ZAPP*** ("Zero Assignment Parallel Processor"), a parallel graph reduction machine that includes dynamic load sharing (Burton and Sleep, 1981).
- ***KRC*** ("Kent Recursive Calculator"), a teaching language in which all statements are recursive definitions at the same level, where each expression being bound to an identifier may include a ***guard predicate*** which indicates under what conditions the definition holds (Henderson and Turner, 1982, pp. 1–28).
- The ***Bath Concurrent LISP Machine,*** a machine for a concurrent variant of LISP 1.5 in which several of the side-effect operators have been deleted (Marti and Fitch, 1982).
- ***FEL*** ("Function Equation Language") and the related ***FGL*** ("Function Graph Language"), in which programs are written as sets of definitions, the concept of lists has been extended to ***sequences*** which can be stored in consecutive memory locations for speed, streams and curried functions are supported directly along with many of Backus's FP functional forms, and for which several implementations have addressed questions of parallelism directly (Keller, 1983).
- ***SETL*** ("SET Language"), a language in which sets and tuples are the major data structures, explicit quantification over sets is permitted, functions may be defined in terms of sets of tuples, and new functions may be built by compositions of such sets (Schwartz et al., 1986).
- The ***FAIM-1*** (and its follow-on ***Mayfly***) parallel machine for generic Artificial Intelligence applications supporting both functional and logic-based processing (Davis and Robison, 1985; Davis, 1989).
- ***ML,*** a language which both predates and postdates HOPE, with some extensions such as exception handling, and which has become a popu-

lar language for driving compiler developments (Mitchell and Harper, 1988; Appel, 1987; Augustsson, 1984; Milner, 1984).

- ***HASKELL*** (Hudak and Wadler, 1988), a very large functional language incorporating nearly all the features described to date, plus a large collection of facilities that should make it usable for very large software projects. It represents an ongoing attempt by the computer science community to define a "standard" high-power functional language.
- Basically functional languages that have integrated in features from logic programming, such as ***LOGLISP*** (Robinson and Sibert, 1980), ***SUPER*** (Robinson and Greene, 1987), and architectures such as the ***Winter Abstract Machine,*** which blends both functional and logic-based computing and supports them in an architecture which is a mix of a supercombinatorlike graph reduction ISA and the ***WAM*** described later in Chapter 17 (Jamsek et al., 1989).

Survey articles by Treleaven et al. (1982), Kennaway and Sleep (1983), Vegdahl (1984), Kogge (1985), and Hudak (1989) include descriptions of many of these, with emphasis on general features. Other references, such as Peyton Jones (1987), survey compiler techniques. In addition, a variety of conferences, particularly the annual ACM Conference on LISP and Functional Programming, are good sources of continuing developments.

Finally, ***dataflow computing*** is another class of computing with strong connections to function-based computing that may be of interest [see Ackerman (1982) and Veen (1986) for a survey]. In dataflow systems, individual instructions are executed as soon as all their input arguments become available. This is similar in concept to, but essentially the opposite of, ***demand-driven evaluation.*** Most implementations use a variant of tagging to mark each datum with the instruction for which it is waiting. Matching hardware of some sort compares these tags to identify when all arguments for some instruction have been made available and the instruction can be executed. While this seems simple, major problems exist in handling function calls, particularly recursive ones, and in passing and modifying complex data structures. Samples of dataflow languages include ***ID*** (Arvind et al., 1980) and ***VAL*** (Dennis, 1982). Implementation techniques to solve some of these problems are discussed in Arvind and Inaucci (1981) and Arvind and Gostelow (1982) and include developing and using recursively defined tags, and ***I-structures*** (which handle partial updates of complex structures). A good example of an entire dataflow machine design can also be found in Gurd and Watson (1980a and 1980b) and Watson and Gurd (1979 and 1982).

12.1 LOGO

Although it is not advertised as such, perhaps the most widely known function-based language is ***Logo*** (designed by Seymour Papert of M.I.T).

Many school children are taught the language to help foster planning and reasoning skills. The standard package of routines so used are called ***turtle graphics*** and permit the user to develop programs that draw complex visual objects on a computer screen by giving a cursor called the "turtle" direction on where to go and what color line to leave behind.

What makes Logo such a good language for this purpose is its simplicity of expression and semantics—both of which come directly from its roots in lambda calculus and LISP. Its major features include the following:

- It was designed for an interpretative and highly interactive environment.
- Function application is the major computational mechanism.
- Prefix notation is standard except for arithmetic expressions which may be infix.
- It supports a variety of s-expressions and equivalents.
- It permits the bodies of functions to resemble LISP ***prog*** statements, which permit sequential execution of multiple expressions.
- It includes many direct analogs of special LISP forms, such as ***global variables, CATCH, THROW, property lists, packages of functions,*** etc.

Figure 12-1 gives an overview of Logo's syntax. There are two major syntatic forms operations and commands. An ⟨***operation***⟩ is equivalent to an expression; it is evaluated for its value. Its format is a direct analog of the major forms of lambda calculus. An operation may be a simple number, an identifier (preceded by a ":" to signify that the value bound to the identifier is desired), a symbol whose name and not binding is desired (an identifier preceded by a "—similar to LISP's ***quote*** form), an arithmetic expression in infix notation, or an application in prefix form (where the function name is first and is *not* preceded by any special character such as " or :). An ***if expression*** takes the value of an operation and selects one of two other Logo forms as its value. "( )" may be used to appropriately separate subexpressions within larger ones.

Lists in Logo are similar to those in LISP except that "[ ]" are used instead of "( )." Built-in functions in Logo that process such lists include ***FIRST*** for car, ***BUTFIRST*** for cdr, ***FPUT*** for cons, and ***SENTENCE*** for **append.** All of these are used as prefix operators in bigger expressions, as in:

```
FIRST (SENTENCE (BUTFIRST [A B]) [C D])
```

which returns B. The predicate ***EMPTYP*** returns T if its single argument is the empty list [ ].

A ⟨***command***⟩ in Logo is a statement that is executed for its side effects, and, unlike LISP, it does not even attempt to return a value. It may be used in two ways: on separate lines in the body of a function defined

```
<procedure> : = TO <proc-name> {:<argname>}*
                  <command> *
                  END
                | DEFINE "<proc-name> [[<argname>*] <command-list>]

<command> : = LOCAL "<name> | (LOCAL {"<name>}+)
              | REPEAT <operation> <command-list>
              | IF <operation> <command-list> {<command-list>}
              | TEST<operation>
              | IFTRUE <command-list>
              | IFFALSE <command-list>
              | OUTPUT {<name> | <number> | <list>}
              | LABEL<name>
              | GO<name>
              | CATCH "<name> <command-list>
              | THROW "<name>
              | MAKE "<name> <operation>

<command-list> : = [ <command>* ]

<operation> : = <number>
              | :<variable-name>
              | "<symbol>
              | <infix-arithmetic-expression>
              | <list>
              | <procedure-name> <operation>*
              | <built-in> <operation>*
              | IF <operation> <command-list> <command-list>
              | (<operation>)

<list> : = [ {<identifier> | <number> | <list>}*]
```

FIGURE 12-1
Basic Logo syntax.

by a ***TO*** keyword and terminated by an ***END,*** and in a ⟨***command-list***⟩ that can be interpreted at runtime.

As with applications, commands are in prefix notation with the leftmost symbol a keyword indicating the command. Many of these commands are similar to LISP forms. IF, for example, evaluates its first argument (an operation) for its value, and then if the value is T, interprets the commands packed in the second argument command-list. If the operation evaluates to F, and there is a third argument, that command-list is executed. If no third argument exists, the IF does nothing.

TEST permits a variation on this. It evaluates its single argument, and keeps the result internally to the Logo system. If multiple TESTs are performed, only the last one is kept. When an ***IFTRUE*** command is executed, this last TEST value is tested, and the command-list given as an argument to the IFTRUE is evaluated only if that TEST value was T. IFFALSE is handled similarly.

OUTPUT takes its single argument and returns it as a result from the procedure in which it is embedded, making that procedure act as a function. ***LABEL*** and ***GO*** work just as in LISP to permit arbitrary branches within the body of a Logo procedure. ***CATCH*** and ***THROW*** also work as in LISP to permit interprocedural branches.

A ***MAKE*** command works much like LISP's ***SET***. The first argument should evaluate to the name of a variable (a leading " is useful here), and the second argument evaluates to a value to replace the variable's current binding.

A procedure may be defined in one of two ways. First, a command line that starts with a TO starts an input mode in which each succeeding line is taken as another command for the procedure's body. A command line of ***END*** terminates this mode. The first argument of the TO defines the name of the procedure. An arbitrary number of arguments following it defines the names of the arguments (each of these should be a symbol name preceded by a ":").

Second, a command line with the keyword ***DEFINE*** can define a procedure dynamically and may be executed anywhere, even from within the body of some other procedure. The first argument to DEFINE should evaluate to a symbol name. Again, a "⟨***name***⟩ is common here. The second argument should evaluate to a list that can be interpreted as a list of argument names (without a leading ":") appended to a command-list. In execution, the second argument is bound to the property list for the first argument as its procedure value. When the procedure is invoked, the first element of the list bound to its procedure value is a list of argument names. Each of these is then bound to one of the actual arguments, and the command-list is executed.

Variables come in three forms and exhibit ***dynamic binding.*** Arguments to a procedure are bound to actual argument values at the call of the procedure and exist only as long as the procedure is executing. Variables that are local to a procedure and again exist only as long as the procedure is executing are created by a ***LOCAL*** command. Any procedure which is called during the execution of a procedure that has arguments or local variables will see those bindings unless they are ***shadowed*** by arguments or local variables to those procedures.

A MAKE command will modify the most current binding for the specified variable. If there is no argument or local variable with that name, a ***global variable*** with that name is created.

Figure 12-2 diagrams some sample code. Sequences (*a*) and (*b*) use some of the turtle graphics commands to draw various kinds of squares. Sequence (*c*) shows some simple expressions, I/O, arithmetic, and list processing.

12.2 FP

(Backus, 1978, 1981, 1985; Harrison and Khoshnevisan, 1985)

Many experts believe that the principal problems with conventional von Neumann programming languages lie in our inability to simply combine small programs into larger programs. Issues such as storage maps and

```
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
```

(*a*) Turtle graphics commands to draw a square.

```
TO DRAW-SQUARE :SIZE
   REPEAT 4 [FORWARD :SIZE RIGHT 90]
   END
DEFINE "SPINSQUARE [[SIZE ANGLE]
                    [DRAW-SQUARE :SIZE RIGHT :ANGLE
                     SPINSQUARE :SIZE + 3 :ANGLE]]
```

(*b*) Sample procedures to draw squares.

```
TO ADDLOOP
   LOCAL NUMBERS
   PRINT [TYPE TWO NUMBERS TO ADD]
   MAKE "NUMBERS READLIST
   IF FIRST :NUMBER = "DONE STOP
   PRINT SENTENCE [THE SUM IS ] (FIRST :NUMBERS)+(LAST :NUMBERS)
   ADDLOOP
   END
```

(*c*) A loop to read two numbers and print sum.

FIGURE 12-2
Some Logo code.

“housekeeping forms” such as do loops (which scatter parts of their operations over multiple statements) make it impossible to argue simply and convincingly about the mathematical correctness of anything larger than toy programs. John Backus (one of the inventors of FORTRAN) has argued for more than a decade that this is a major cause of the crisis in software development and that alternatives, particularly function-based ones, are needed. Further, he argues that what is needed is something higher than a simple lambda calculus-based language. What is needed are languages in which the key program-building operations are ***functional forms:*** functions which map other functions into new functions in clear and unambiguous fashions. This is in contrast to most of today's functional languages, which he terms ***object-level functional programming*** because of their concentration on building functions which map objects into objects.

The following subsections summarizes briefly a system that Backus introduced in his ACM Turing Award lecture (Backus, 1978) called simply ***FP*** and developed further since then. Although meant to be a mathematical language about which strong statements can be proved, it should be clear after reading that the concepts can be (and have been) embedded in real functional languages. A simple example that runs on personal computers can be found in Robison (1987). Extensions include ***strong typing*** (Guttag et al., 1981) and specialized machine architectures (Huynh et al., 1986).

12.2.1 FP Systems

An FP system is a mathematical system that includes mechanisms for defining:

- A set of objects consisting of ***atoms*** (***atomic constants***), ***tuples*** of objects, and $\perp$ (the ***bottom element***)
- A set of basic functions that map single objects into other objects
- A set of ***functional forms*** or ***program-forming objects*** (***PFO***s) that map functions into other functions
- A single infix operation called ***application*** (denoted by “:”), which actually indicates a computation to be performed

Figure 12-3 gives Backus's syntax for expressions in such a system. The notation here differs from that used elsewhere in this book, particularly for tuples (what Backus calls ***sequences*** and for which he uses ⟨ ⟩), ***conditionals*** (which Backus designates as “$\mathbf{p \rightarrow q;r}$” rather than if p then q else r), and in definitions, where def is used in place of let or letrec. We use Backus's notation here because of its widespread use in the literature on FP.

As with any functional language, application of a function to a single argument is the sole method of actually performing computations. A function that needs more than one object for its evaluation assumes that they are combined into a single tuple. Also, a method is included for assigning a name to an expression that delivers a function result.

Note that neither the concept nor the usage of a ***variable*** appears here. As with combinator expressions (***cafs***), all expressions are built by applying either functions to objects or functional forms to functions.

The object $\perp$ (***bottom*** or ***undefined***) is a special object assumed to be in the domain of all functions. All functions in an FP system are said to be ***bottom preserving***; that is, whenever any argument is $\perp$, the result is always $\perp$. A function may also produce $\perp$ from non-$\perp$ arguments. This permits normally partial functions, such as division, to become total functions (by mapping x divided by 0 into $\perp$). The object $\perp$ is thus a mathematically sound, and computationally handy, mechanism for signaling exception conditions and special cases where the function definition is not applicable.

In FP certain basic functions called ***constructors*** take one or more objects and build a tuple which can then be used as a single argument to a function. Notationally, surrounding one or more objects by “⟨ ⟩” with separation by “,” is a syntatic way of indicating that the basic construc-

```
<object> := ⊥ | <atom> | <tuple>

<tuple> := ≠ | <<object-expr> {,<object-expr>}*>

<object-expr> := <object> | <function-expr>:<object-expr>

<function> := <primitive> | <constant> | <selector> | <definition>

<primitive> := + | − | × | ...
               | id | trans
               | append-left | append-right
               | dist-left | dist-right | ...

<constant> := <number>

<selector> := <number>

<definition> := def <identifier> = <function-expr>

<function-expr> := <function>
                 |  <PFO>(<function-expr> {,<function-expr>}*)
                 |  <function-expr>∘<function-expr>
                 |  <function-expr> → <function-expr>
                    ; <function-expr>
                 |  [<function-expr> {,<function-expr>}*]
                 |  /<function-expr>
                 |  \<function-expr>
                 |  α<function-expr>

<PFO> := ∘ | →; | [ ] | / | \ | α
```

FIGURE 12-3
Basic FP syntax.

tor functions are to be applied to those objects to form a tuple. Although this is like **cons** or **list,** there is no sense of the result being a linked data structure; it is simply an object like any other that results from a function application.

Typical basic functions include arithmetic, boolean, and equality tests, along with standard tuple-manipulating functions such as **reverse, append,** and **rotate.** Other functions, such as **id** (equivalent to the combinator **I**) and **transpose** (take a matrix expressed as a tuple of tuples, and reverse the order), are simple but useful functions. Generalizations of **car** which permit access to the n-th top-level element are expressed as simply the integer n in the function position of an expression. Thus 3:*x* corresponds to the third element of the tuple *x*. Backus also includes variations that start with the back of the tuple rather than the front (an integer followed by an **r,** as in "2r"). Other variations of functions such as **appendr** work similarly.

Any simple object, such as an integer, can be made into a function by simply placing it in the function position of an expression and underlining it. The effect is that, regardless of the actual argument, the value returned is the value of the basic object. Thus $\underline{17}:x$ returns 17 regardless of what *x* is. This is virtually identical to a combinator expression such as **K** 17. (Remember that 17:*x*, without the underline, is a different expression, the 17th component of the tuple *x*.)

12.2.2 Program-Forming Operations

Perhaps the most unique and compelling part of FP is the way new functions (programs) are built. Rather than describing new functions in terms of application of functions to objects (or variables that will be later bound to objects), new FP functions are built by applying ***program-forming objects*** (***PFO***s) to other functions. Figure 12-3 defines the syntax for several of these; Figure 12-4 gives a simple definition of each in terms of how the functions they produce operate on arbitrary objects. Figure 12-5 then gives several examples.

It is critical to realize that these PFOs are higher-order functions similar to those described at the end of Chapter 5. They take functions as inputs and produce new functions, not objects, as results. In turn, these functions can either be combined via other PFOs into yet bigger functions or applied to some object to compute a new one. For example, the ***conditional*** PFO takes three functions as its arguments and returns a single function which, when that is applied to an object, results in applying the first function to the object, and if that is true, returning the result of applying the second function to the object. The third function is applied to the object if the result was false.

Similarly, the PFO ***construction*** "[ ]" brackets a series of functions in what looks like a tuple. It is not a tuple, however; it is a function. Only when applied to an object does it return a tuple of the same length as the construction, where the n-th component of the tuple is the result of applying the n-th function within the "[ ]" to the argument object.

Figure 12-5 defines several simple programs and a sample evaluation of each. The reader should trace these and make sure that he or she

PFO	Equivalent Operation When Applied
composition ∘	(f∘g):x → f:(g:x)
Conditional	(p → f; g):x → (p:x → (f:x) ; (g:x))
Construction [ ]	[f_1, ... f_n]:x → <f_1:x, ... f_n:x>
Right Insert /	/f:<x,y> → null:x → y ; f:<1:x, /f:<cdr:x, y>>
Left Insert \	\f:<x,y> → null:x → y ; f:<\f:<cdr:x, y>, 1r:x>
Map α	αf:<x_1, ... x_n> → <f:x_1, ... f:x_n>

Assume f, g, and p arbitrary functions, and x and y arbitrary objects.

FIGURE 12-4
Basic set of program-forming objects.

```
def subtract1 = −∘[id, 1̲]
   Example: subtract1:3 ≡ −∘[id, 1̲]:3
   → −:([id, 1̲]:3)
   → −:<id:3, 1̲:3>
   → −:<3, 1> → 2

def factorial = 0= → 1̲ ; ×∘[id, factorial∘subtract1]
   Example: factorial:3 ≡ (0= → 1̲ ; ×∘[id, factorial∘subtract1]>:3
   → 0=:3 → 1̲:3 ; (×∘[id, factorial:subtract1]):3
   → (×∘[id, factorial:subtract1]):3
   → ×:<id:3, factorial∘subtract1:3>
   → ×:<3, factorial:2>
   → ...
   → ×:<3, ×:<2, ×:<1, 1̲:0>>>
   → ×:<3, ×:<2, ×:<1, 1>>> → ×:<3, ×:<2, 1>> → ×:<3, 2> → 6

def inner-product = \+∘α×∘transpose
   Example: inner-product:<<1, 2, 3>, <4, 5, 6>>
   → \+:(α×:(transpose:<<1, 2, 3>, <4, 5, 6>>))
   → \+:(α×:<<1, 4>, <2, 5>, <3, 6>>)
   → \+:<×:<1, 4>, ×:<2, 5>, ×:<3, 6>>
   → \+:<4, 10, 18>
   → +:<4, \+:<10, 18>>
   → +:<4, +:<10, 18>> → +:<4, 28> → 32
```

FIGURE 12-5
Sample FP programs.

follows the reductions, particularly when an expression is a function and when it is an object.

12.2.3 Some Provable Laws

Given their somewhat ad hoc nature, the program-forming constructs of most programmimg languages offer little in the way of general mathematical statements about the way in which programs using them can be built. In contrast, the PFOs of FP permit a variety of relatively strong algebraic statements to be made about them and their use. Figure 12-6 gives a small selection from literally dozens of these listed in Backus and other works. Each such statement indicates how some relatively large class of programs built from PFOs is related to some other class. (Note that these really are mathematical statements and not programs; the variables in them may be replaced by arbitrary real FP programs and the statements will hold.)

Proving the correctness of these laws is usually accomplished by applying an arbitrary object to each side of the equality, and using the definitions of the PFOs to demonstrate that they reduce to identical expressions. For example, for the third law, applying an arbitrary object x to the left side yields:

1. $\mathrm{id}\circ f \equiv f\circ \mathrm{id} \equiv f$
2. $f\circ(g\circ h) \equiv (f\circ g)\circ h$
3. $f\circ(p \rightarrow q\ ;\ r) \equiv p \rightarrow f\circ q\ ;\ f\circ r$
4. $(p \rightarrow q\ ;\ r)\circ f \equiv p\circ f \rightarrow q\circ f\ ;\ r\circ f$
5. $[f_1, \ldots f_k, (p \rightarrow q\ ;\ r), f_{k+2}, \ldots f_n]$
   $\equiv p \rightarrow [f_1, \ldots f_k, q, f_{k+2}, \ldots f_n]\ ;\ [f_1, \ldots f_k, r, f_{k+2}, \ldots f_n]$
6. $[f_1, \ldots f_n]\circ g \equiv [f_1\circ g, \ldots f_n\circ g]$
7. $\alpha f\circ[g_1,\ldots g_n] \equiv [f\circ g_1,\ldots f\circ g_n]$
8. $/f\circ[[g_1,\ldots g_n], h] \equiv f\circ[g_1, f\circ[g2, \ldots f\circ[g_n, h]]\ldots]$
9. $\backslash f\circ[[h, g_1,\ldots g_n]] \equiv f\circ[f\circ[\ \ldots [f\circ[h, g_1], g_2], \ldots g_n]$
10. $/h\circ[\mathrm{id}, i] \equiv \backslash h\circ[i, \mathrm{id}]$ if h is associative with identity i

Note: f, g, p, q, r are all arbitrary functions.

FIGURE 12-6
Some provable laws.

f ∘ (**p**→ **q**; **r**) : *x*→ **f**:(**p**:*x*→ **q**:*x*; **r**:*x*)

which yields **f**:**q**:*x* if **p**:*x* is true, and **f**:**r**:*x* otherwise. This is exactlywhat applying the right side to *x* yields. The two halves are thus identical, regardless of what the functions **f, g,** and **r** are.

The usefulness of these laws lies in aiding our formal understanding of how programs work, and in permitting the derivation of mathematically sound ***optimizations*** that can be applied at any time by rewriting programs, or pieces of programs, into other programs that are perhaps more efficient. A prime example of this is a relatively general approach to transforming certain recursive programs into others that are ***tail-recursive,*** and thus executable in a more efficient iterative form. Figure 12-7 shows an example of this for the factorial function given in Figure 12-5.

The key to such transformations comes from the ***Recursion Removal Theorem,*** which says that if we have a program of the form (for arbitrary smaller programs **g, h, p, q,** and **r**):

def **f**=**p**→ **q**; **r** ∘ [**g**, **f** ∘ **h**]

where **r** is an ***associative function*** on pairs (that is, **r**:<*x*, **r**:<*y*, *z*>>=**r**:<**r**<*x*, *y*>, *z*>), with an ***identity element*** **i** (**r**:<*x*, **i**>=**r**:<**i**, *x*>=*x*), then one can construct a function **f1** defined as:

def **f1**=**p** ∘ 1→**r** ∘ [2, **q** ∘ 1]; **f1** ∘ [**h** ∘ 1, **r** ∘ [2, **g** ∘ 1]]

This new function **f1** accepts an argument which consists of a pair of el-def factorial = 0= → $\underline{1}$; ×∘[id, factorial∘subtract1]

In terms of the Recursion Removal Theorem:
p = 0=
q = $\underline{1}$
r = × (associative with identity 1)
g = id
h = subtract1

Then f1 = 0=∘1 → ×∘[2, $\underline{1}$∘1] ; f1∘[subtract1∘1, ×∘[2, id∘1]]

and factorial = f1∘[id, $\underline{1}$].

Example: factorial:4
→ f1∘[id, $\underline{1}$]:4
→ f1:<id:4, $\underline{1}$:4>
→ f1:<4, 1>
→ 0=∘1 :<4, 1> → ×∘[2, $\underline{1}$∘1]:<4, 1>
; f1∘[subtract1∘1, ×∘[2, id∘1]]:<4, 1>
→ 0=:4 → ×:<2:<4, 1>, $\underline{1}$:1:<4, 1>>
; f1:<subtract1:1:<4, 1>, ×:<2:<4, 1>, id:1:<4, 1>>>
→ f1:<subtract1:4, ×:<1, id:4>>
→ f1:<3, ×:<1, 4>> → f1:<3, 4>
→ ... → f1:<1, 24>
→ ... → f1:<0, 24>
→ ×∘[2, $\underline{1}$∘1]:<0, 24>
→ ×:<2:<0, 24>, $\underline{1}$:1:<0, 24>>
→ ×:<24, $\underline{1}$:0, 24> → ×:<24, 1> → 24

FIGURE 12-7
Making factorial tail-recursive.

ements (both functions) and is fully tail-recursive (the "else" branch returns the result of applying **f1** to some other expression). This means that it can be implemented without using a stack for intermediate calls—it can be implemented as a loop. Further, and more important, it permits us to rewrite the non-tail-recursive function **f** in the fully tail-recursive form:

def **f**=**f1** ∘ [**id**, *i*]

This kind of transform should look suspiciously familiar. We construct an argument for **f1** from the original argument to **f** (delivered by **id**) and a constant (**i**). Each tail-recursive call to **f1** modifies both components, and returns some function of both of them when the predicate **p** is satisfied. In fact, what this theorem does is take nonrecursive definitions of functions and automate the discovery of equivalent functions which use ***accumulating parameters*** to avoid costly recursions.

In the sample derivation of Figure 12-7, not only do we get an iterative form of factorial using this technique, but by using other algebraic laws (such as **id** ∘ **g** ≡ **g**), even further simplifications can be made.

Although it is not complex, the proof of this theorem is too long to be repeated here. The interested reader is referred to Backus (1985) or Kieburtz and Shultis (1981) for details.

12.3 HOPE

(Eisenbach, 1987; Bailey, 1985; Burstall et al., 1980; Sannella, 1981)

Nearly all the functional languages discussed to this point have been ***untyped;*** that is, the programmer provides no information about the nature of the objects to be bound to variables (especially formal function arguments). It is assumed that the underlying compiler or interpreter will maintain runtime tags that determine the current data type of each object.

In contrast, most conventional languages have varying degrees of ***strong typing,*** which requires the programmer to specify explicitly the format of values bound to variables, in structure and in how the bits are to be interpreted. This both gives more bits for value fields and permits significant optimizations, since no runtime type checking is needed, and the code generated can be specialized to exactly the type of operands expected.

HOPE is a functional language with such strong typing (see the examples of Figure 12-8 and the simplified syntax of Figure 12-9). All function definitions must explicitly declare their domains and ranges. The types for these sets can be either built-in or programmer-developed. Further, such declarations are done in an expandable and rigorous fashion, permitting mathematical (and compile-time) analysis and reasoning about the nature of HOPE programs. This permits more correct and efficient code.

Another major characteristic of functional programs is that they often contain highly nested if-then-elseif trees (just look at our various abstract interpreters). These can become hard to read, especially when they are deeply nested. HOPE also supports a construct to simplify this, namely, a ***pattern matching*** facility that selects one statement for the body of the function from a set of them on the basis of values of the actual arguments. Such a facility represents a small step toward the more sophisticated pattern matching called ***unification,*** which will drive the logic programming systems discussed in the latter part of this book.

There are other features of HOPE that are not discussed here, namely, support for modules of programs, infix notation, operator overloading, and I/O.

Finally, HOPE was designed for lazy evaluation, with a machine like ALICE (described later) in mind. Eisenbach (1987, chap. 11) describes an abstract machine and a runtime system that matches.

12.3.1 HOPE Syntax

There are four basic statement types in HOPE (see Figure 12-9). All start with a keyword or special character string, and end with a ";". This sec-

```
dec fact num -> num;
--- fact(n)<= if n<2 then 1 else n×fact(n − 1);

dec ackerman : num # num -> num;
--- ackerman(0,j) <= j+1;
--- ackerman(i,0) <= ackerman(i − 1,1);
--- ackerman(i,j) <= ackerman(i − 1, ackerman(i,j − 1));

typevar anytype;
dec append : list(anytype) # list(anytype) -> list(anytype);
--- append(nil, y) <= y;
--- append(head::tail, y) <= head::append(tail,y);

typevar listtype, rtype;
dec map: (listtype->rtype) # list(listtype) -> list(rtype)
--- map(f, nil) <= nil;
--- map(f, head::tail) <= f(head)::map(tail,f);

dec build-scaling-fcn num -> (num -> num);
--- build-scaling-fcn(n) <= lambda(x) n×x

typevar leaftype;
data tree(leaftype) == nil ++ leaf(leaftype) ++
     node(tree(leaftype), tree(leaftype));

dec flatten : tree(leaftype) -> list(leaftype);
--- flatten(nil) <= nil;
--- flatten(leaf(n)) <= n::nil;
--- flatten(node(x, y)) <= append(flatten(x), flatten(y));
```

FIGURE 12-8
Some HOPE examples.

tion overviews these statements. Specific aspects of this syntax and their matching semantics is given in the following sections.

Of most importance is the ***function definition*** statement. This starts with the keyword dec and gives the name of a new function to be defined. Then, just as with the original mathematics of Chapter 2, set descriptions for both the domain and the range of the function are given. These descriptions are separated from each other by "→" and from the function name by a ":". Both of them are built from ***data type*** descriptions available from either the system or from programmer definition (discussed later). Although all HOPE functions formally accept exactly one argument (just like their mathematical counterparts), the descriptions permitted for these domain and range types cover structures such as tuples, which permit arbitrary argument groupings in practice. As an example, in Figure 12-8 the function **map** accepts a tuple of two arguments, one of which is a function itself, and the other a list. The domain of the function argument is the same as the type of the objects joined in the second argument's list. The result of **map** applied to such a pair is a list of objects whose types are the same as that of the range of the first argument.

```
<statement> := <function> | <type-var-decl> | <type-def> | <equation>

<function> := dec <fcn-name> <domain-type> -> <range-type>;

<equation> := "---" <fcn-name> <pattern> <= <expression>;
<expression> := <constant>
              | <identifier>
              |  <expression><builtin-infix><expression>
              |  <fcn-name>(<expression>{, <expression>}*)
              |  if <expression> then <expression> else <expression>
              |  let <identifier> == <expression> in <expression>
              |  <expression> where <identifier> == <expression>
              |  lambda(<identifier>{, <identifier>}*) => <expression>
              |  nil | [<expression>{, <expression>}*]
<pattern> := (<simple-pattern>{,<simple-pattern>}*)
<simple-pattern> := <identifier>|<constant>
                  | <simple-pattern>::<simple-pattern>
                  | <constructor>(<simple-pattern>,
                     {,<simple-pattern>}*)

<type-def> := data <typevar> == <constructor>{++ <constructor>}*
<constructor> := num | truval | char | <constant>
               |  list(<constructor>)
               |  set(<constructor>)
               |  map(<domain-type> → <range-type>)
               |  <constructor>#<constructor>
               |  <constructor>(<constructor>{,<constructor>}*)
<domain-type> := <constructor>
<range-type> := <constructor>

<type-var-decl> := typevar <identifier>{, <identifier>}*;
```

FIGURE 12-9
Simplified syntax for HOPE.

Following a function definition in a program is one or more ***equation statements*** that make up the body of the definition. Each such statement consists of a leading "---", followed by a ***pattern*** that defines when the statement is to be applied, and a companion expression on the right side of an " ⇐ " (read "is rewritten as"). The pattern on the left represents properties the actual arguments must have before that equation is usable. The expression on the right represents how the actual arguments should be transformed into a result. As shown in Figure 12-9, the permissible forms for an expression are direct analogs to those discussed earlier for abstract programming. The meanings of each are exactly as before.

Although there may be many such equations for a single function definition, only one of them should be applicable for a particular real argument. There should be no overlap in patterns.

The ***type definition*** statement in HOPE permits a programmer to define a new data type. Such a statement starts with the keyword data and consists of an identifier by which the data type will be known and a data type expression separated from this identifier by a "==" (read as "is defined as"). This right-side expression can describe something as simple as a set of constants, such as {1, 2, 3} or {**red, white, blue**}, or as complex as a multiply nested data structure where each component can be some of arbitrary type. Note that a symbol such as "**red**" can be treated as a constant much as in Pascal or other modern von Neumann languages.

The type expressions in both the function definitions and the type declarations can themselves be parametric. This means that the types themselves can be functions of other types. For data statements this is signified by following the left-hand side type identifier by "( )," with other identifiers (called ***type variables***) inside them. Using these identifiers on the right-hand side means that the type of the overall data structure will be a function of the type of the object provided when the type declaration is used. For example, in the following statements the type *twotuple* is defined as a pair of elements whose types can be anything as long as they are the same. Then the function **foo** is defined to have a domain consisting of a pair of numbers and a range of a pair of objects, both of which are a pair of booleans.

```
data twotuple(x) == (x#x);
dec foo: twotuple(num)→twotuple(twotuple(truval));
```

When such statements are used in domain and range type expressions in a function definition, confusion is possible between what should be a type variable and what is a symbolic constant. For this reason, the final HOPE ***type variable declarations*** statement is included. This statement begins with the keyword typevar and consists of a list of identifiers to be treated whenever they appear in the following function definitions as type variables, not symbolic constants. When the functions are used in actual applications, the type of their actual arguments will be substituted for these type variables, permitting the type of the result to be predicted.

12.3.2 Specific Data Types

All HOPE objects have associated with them a ***data type*** which indicates the structure of the object (how smaller pieces of it may be accessed), what such a structure should be called, and how the smallest pieces without further structure are to be interpreted. The type declaration statement defines new types; the function definition and equation statements use them. When a value is printed out after an application is complete, its data type is also printed out.

There are three unstructured types provided in the language: ***num*** for numbers, ***truval*** for boolean "truth values," and ***char*** for character strings.

As described above, the syntax of the type declaration statements

permit a new type to be defined in terms of old types. For example, in Figure 12-10(*a*), the type **one-bit-integers** corresponds to the set of integers 0 and 1, the type **days** contains the names of the days of the week as symbolic constants, and **one-bit** may be either a boolean or a 1-bit integer. Note the use of the "++" (read "or") to indicate that objects of either type are to be considered valid objects of the type on the left of the ==. This is somewhat equivalent to the union operator.

None of these types has any structure. HOPE permits structured types to be defined by ***constructor functions*** and used in type declarations by what look like applications. Figure 12-10(*b*) diagrams several examples. An object of type **boolean-pair,** for example, must be a tuple of two other objects, both of which must be truvals. An **integer-tree** object can actually have three different internal structures: It can be the simple unstructured constant **nil**; it can be a **leaf** with an integer value; or it can be a **node,** which itself has two subcomponents, both of which may be arbitrary trees.

The identifiers **nil, leaf,** and **node** are all constructor functions, and may be used within expressions to "construct" objects of the desired type. Note that recursively defined types are permitted. For example, the expression "**node(leaf(1), node(leaf(2), nil)),**" when used in an equation, would result in HOPE building an object of type **integer-tree.**

Finally, HOPE permits definition of ***higher-order types*** in which the data type is generic. This is indicated by putting ***type variables*** in parentheses after the type name in a data statement, and using those variables as a type name on the right-hand side. The resulting type can then be

```
data one-bit-integer == 0 ++ 1;

data days == Sunday ++ Monday ++ Tuesday ++ Wednesday ++
             Thursday ++ Friday ++ Saturday;

data one-bit == one-bit-integer ++ truval;
```

(*a*) Simple enumerated types.

```
data boolean-pair == truval#truval;

data integer-tree == nil ++ leaf(num) ++
                     node(integer-tree, integer-tree);
```

(*b*) Structured types.

```
typevar anytype;
data generic-tree(anytype) == nil ++ leaf(anytype) ++
  node(generic-tree(anytype), generic-tree(anytype));
data integer-tree == generic-tree(num);
```

(*c*) Higher order types.

FIGURE 12-10
Some sample HOPE data type definitions.

used to define more specific types in further type declarations [as in Figure 12-10(*c*)] or within expressions.

Finally, HOPE includes several generic data types as part of the language. These include tuples, lists of arbitrary objects of the same type, sets of objects of the same type, and functions. The symbol "#" is a built-in constructor for tuples. For lists, the constructor "::" is equivalent to a **cons,** while the notation "[a, b, c]" is shorthand for **a::(b::(c::nil)).**

12.3.3 HOPE Pattern Matching

Each equation statement in the body of a function corresponds to one branch of what would normally be a nested tree of if-then-elseifs. The left side of each equation is called a ***pattern,*** and it determines whether or not that equation is applicable to a particular argument. This pattern is matched against the actual argument, and if a match is possible, any variables in the pattern are bound to components of the actual argument before proceeding to the expression on the right.

The actual pattern-matching process starts with determining if the actual type of the argument matches that expected in the pattern. This is particularly relevant to enumerated types, such as the trees of Figure 12-10, where the actual argument is only one of these. This match proceeds by comparing the names on the constructor functions, if any, called out in the pattern to those actually embedded in the argument value. Thus, in the function **flatten** of Figure 12-8, each of the three equations covers one of the three outermost constructors possible for the type **tree.**

As an aside, HOPE permits notation of the form "(**a, b, c**)" in a pattern to stand for a tuple of the form **a#b#c.**

Given that the constructors are correct, the values found as subterms to those constructors in the actual arguments are compared to the corresponding elements of the pattern. If the pattern element is a constant such as "1" or "**nil,**" the actual element must match exactly or the equation does not apply. If the pattern element is a variable and there are no mismatches elsewhere, then that variable is bound to the actual argument element when the expression is executed.

The function **ackerman** in Figure 12-8 gives several examples of this. The first equation covers the case where the argument is a pair whose first element is a 0. The second equation covers the case where the second component is a 0; the third equation matches all other cases.

The final equation for **flatten** is a slightly more complex case that demonstrates the value of the notation. It applies only when the object is a **node,** and then it automatically performs the equivalent of a **car** and **cdr** on the object, all without any nested lets or the equivalent. In contrast, the abstract program code for such an equation would look something like:

```
flatten(n)=if null(n) then n
                elseif atom(n) then cons(n, nil)
                else let x=car(n)
                        and y=cdr(n) in
                        append(flatten(x), flatten(y))
```

In many cases the HOPE form is more readable, even if it places more work on the compiler or interpreter to parse it and generate the appropriate equivalent actions.

12.4 TREE MACHINES

(Mago, 1979, 1980, 1985; Mago et al., 1981)

If one thinks about evaluating an expression in parallel, the first thing that happens is the evaluation of all the lowest-level applications where the argument values are available directly. After that the next highest applications are done, and so on. The next result is a treelike pattern of reductions.

Such general observations have led several groups to investigate parallel machine designs for functional languages which are in fact arranged as a tree. Mago's ***Cellular Tree Machine*** or ***FFP Machine*** was one example (Figure 12-11), where the ultimate goal was something that would fit on a VLSI chip. Other similar projects included the ***AMPS Machine*** (Keller et al., 1979), the ***X-tree project*** (Despain and Patterson, 1978; Patterson et al., 1979), and the ***Caltech Tree Machine*** (Browning, 1980).

For the most part, such machines consist of treelike interconnections of two kinds of processors, one kind for the leaves (***L processors***)

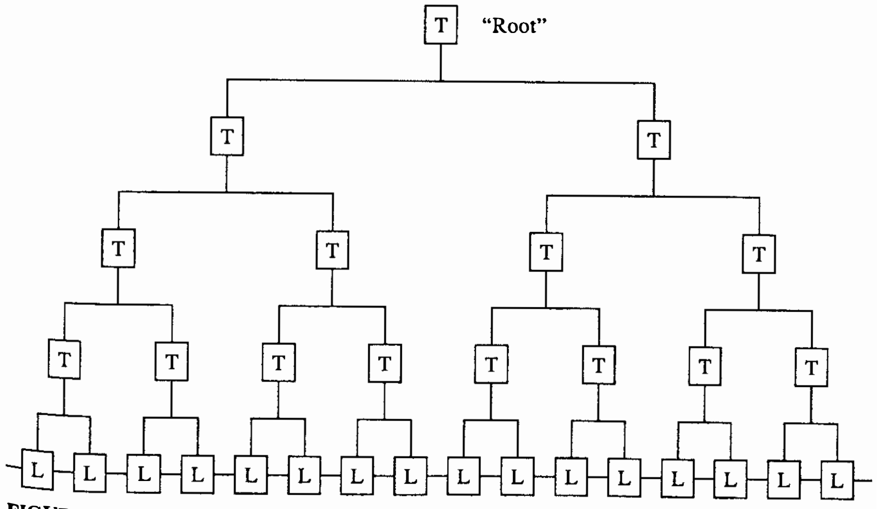


FIGURE 12-11
The FFP Tree Machine.

and one for the internal nodes (***T processors***). One of the advantages of such an organization is that if the processors can be made small enough, they can be placed in a recursively growing H pattern on the surface of a VLSI chip, as shown in Figure 12-12.

For Mago's design, the program (a variant of Backus's FFP language) is stored one symbol at a time in the leaves. All of these leaves are connected in a linear string to permit easy loading. A ***nesting level*** number is included with each symbol to indicate how many parentheses deep the symbol is in the original program. The L processors are actually small microprogrammable machines where all real work is performed. The T nodes provide communications between the L processors to permit arguments to be transmitted to functions. There is no central clock; each processor is a small finite state machine that asynchronously communicates with its neighbors as needed.

Computation proceeds in cycles, each of which resembles one or more waves that start at the leaves, proceed up the T cells, and then go back down. In the first such wave, called the "partitioning phase", the symbol and its nesting level is transmitted up the tree to its T cells. Each T cell looks at the nesting level and type of the two symbol streams it receives, and decides whether they represent a full application (function plus arguments) or part of an expression that must be passed farther up. In the former case, the arguments are passed back down the path on which the function came up, until it is made available to the L cell holding the function symbol. This all occurs in parallel; at the end of it all applications which have no nested applications have their arguments collected in with the functions. Microcode for each operation is then broadcast in parallel down from the root of the tree and stored in each node that needs that operation.

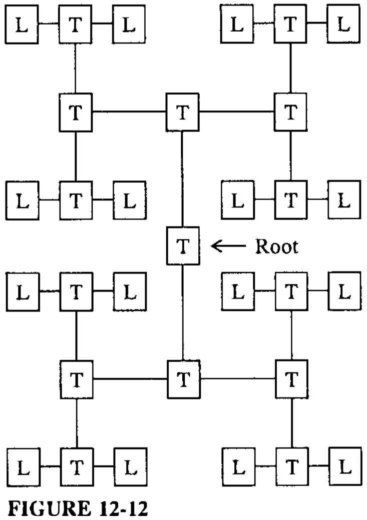


FIGURE 12-12
H-shaped VLSI layout of Tree Machine.

In the second "execution phase," all leaves marked as having a valid application proceed to perform them, again in parallel. In the final "storage management phase," the results of each evaluation are redistributed one symbol to a leaf. Leaves that no longer contain symbols are collapsed out. Both linear shifts of the L nodes and waves up and down the tree are possible.

These multiphase cycles are repeated until the entire program string has been reduced to a single token in a single leaf. The net effect is a maximally parallel ***string reduction*** of the original string into the final one.

12.5 ALICE

(Eisenbach, 1987, chap. 8; Pountain, 1985; Darlington and Reeve, 1981)

ALICE ("Applicative Language Idealized Computing Engine") is a flexible parallel processor designed for ***graph reduction*** of programs in which the applications can be stated in terms of ***rewrite rules.*** Each node in such graphs corresponds to an application and contains information about the name of the function that forms it and the arcs that leave it. In HOPE terminology, this also includes *constructor functions* from which data structures are built. The arcs from a node indicate which other nodes provide results needed as arguments for this application (or whose values are subterms for the data structure this node represents).

In ALICE, the information describing a node is kept in a ***packet,*** with all such packets kept in a ***packet pool.*** Execution proceeds by having many ***reduction agents*** survey the packets for those whose arguments are sufficiently computed for at least one step of the function in the node to be executed. When such a packet is found, a reduction agent rewrites it to either a simple value (if the function was a built-in) or to a new subgraph representing the expression that corresponds to the function's definition. Depending on the nature of the rewrite, the agent may signal the parent node of the node just rewritten, informing it that the rewrite has occurred and that it may now be ready to be evaluated. All the reduction agents work in parallel on different nodes in the graph. When the graph reduces to one irreducible node, program execution is complete.

This model of computation has turned out to be fairly generic, and the machine is a good target for a variety of languages, from HOPE, LISP, and FP, to combinators and PROLOG, and even including conventional languages like PASCAL. The following sections will address the machine as if HOPE were the original programming language. Further, we will address only ***eager evaluation*** techniques. ALICE also supports a variety of lazy and mixed-order evaluations; discussions of them will be left to the references.

12.5.1 Packets

Figure 12-13 diagrams the contents of an ALICE packet. There are six major fields, with the contents of one of them, the Status field, divided even further.

Packet name	Unique name of this packet in system.
Function name	Name of Builtin, Function, or Constructor.
Argument list	Packet names (or constant values) of all arguments needed for this node.
Signal list	Names of all packets which have signalled that they need the result of this packet.
Reference count	Number of packets referencing this one.
*Pending signal count	Number of arguments still unevaluated.
*Lazy bit	Indicator to propagate Not-Yet-Required.
*Not-yet-required bit	Indicator to prevent premature evaluation.

*Part of status field.

FIGURE 12-13
An ALICE packet.

First is the ***Packet Name*** field. The contents of this field represents a unique name by which the rest of the computer can reference it. This cannot be simply the name of the memory cell starting the packet, because packets are often copied, and after rewrite the same "node" may reside in a physically different area of memory.

Next is the name of the function associated with the node this packet represents. In the case of HOPE this function comes in three flavors: built-ins, program-defined functions (via a dec statement), and the data structure defining ***constructor functions*** mentioned in data statements. Only the first two have code associated with them. The third serves as a convenient place to join together multiple objects needed to build a bigger object. If we were implementing LISP on ALICE, for example, the only such constructor might be a **dot,** corresponding to the result of executing a **cons.**

The third field, the ***Argument List,*** contains a set of packet names for the arguments for this function. These thus correspond to the arcs needed to connect the nodes of a graph. In cases where the arguments are known to be simple constants, a value is substituted for a packet name in this list.

The ***Signal List*** is a list of packets that have indicated they are awaiting the result of reducing this packet before they can continue.

The ***Reference Count*** field keeps a count of the number of packets which have arcs to the current packet; that is, their argument lists include the name of this packet in them. Each time such a packet is done with this packet, the reference count is decremented. When the count becomes zero, the packet is ***free,*** and may be reclaimed for use as another node. The ALICE system is deliberately designed to guarantee that there is never a cyclic loop in a program graph, so that the problems mentioned with regard to reference-counting ***garbage collection*** in Chapter 8 never materialize.

The ***Pending Signal Count*** is a count of the number of packets in this packet's argument list that have not yet signaled that they have been reduced. As long as this count is nonzero, this packet is not reducible—there is still some argument whose value has not been evaluated to the point where a reduction agent can take over this packet and execute the function.

Also in a packet is a ***Not Yet Required Bit,*** which indicates that this packet should not be reduced, even if all its arguments are available, until its value really is needed by some other packet. In addition, the ***Lazy Bit*** signals the system that all packets spawned from this one should be marked "not yet required." Together, these two fields permit lazy evaluation techniques to be implemented, and to focus the attention of the available reduction agents on parts of the graph which directly affect computation.

12.5.2 Hardware Architecture

Figure 12-14 diagrams the architecture of the prototype ALICE machine built at Imperial College in England. There are two interconnection systems and two kinds of processing modules. Up to 64 modules can be accommodated.

Each of the modules is constructed out of INMOS, Inc., ***Transputer*** microprocessors. These chips contain on them a RISC-like CPU, four 10-Mbit/s serial links that can be interconnected to other Transputers in arbitrary configurations, and a small amount of fast RAM memory.

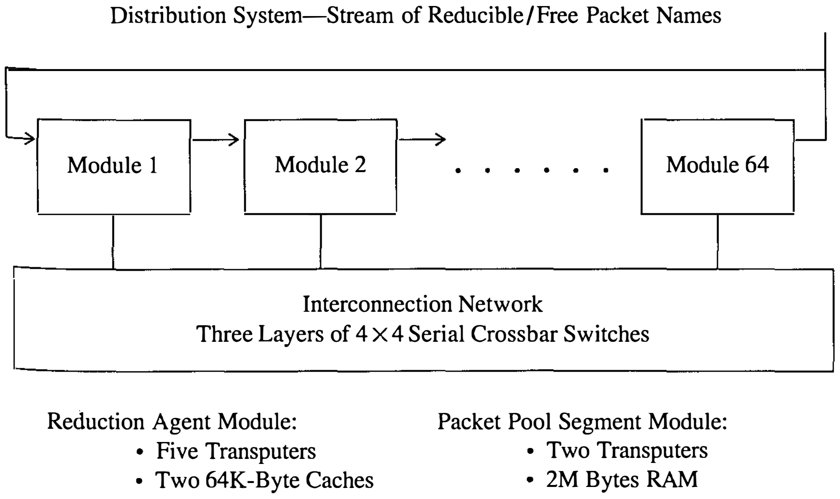


FIGURE 12-14
The prototype ALICE machine architecture.

The ***Reduction Agent*** module consists of five Transputers and two 64-Kbyte packet caches. Two of these Transputers are used as ***Rewrite Units,*** which take reducible packets, request the code for the functions, create the new graphs, and initiate signaling of parent nodes. For efficiency, each physical rewrite unit maintains eight sets of registers, permitting eight virtual node-rewriting processes to be active at the same time. One of these processes can be executing on the Transputer while the other seven are waiting for I/O (such as program code fetch) to complete.

The other module type, a ***Packet Pool Segment*** or ***PPS,*** represents a chunk of packet storage. It contains two Transputers and 2 Mbytes of RAM. It has two functions: to process messages from the reduction agents, and to keep a stream of reducible and free packet names on the Distribution System (described below). An example of a message from an agent is to inquire about whether or not some named packet has been reduced.

The ***Interconnection Network*** is a very-high-speed multilayer interconnection network built out of 4 by 4 crossbar switches in custom Emitter-Coupled Logic chips. These crossbars permit any arrangement of their four inputs to be connected to any arrangement of their four outputs at the same time, giving four simultaneous 150-Mbit/s serial communications per chip. Three layers of 16 such chips permit 64 such simultaneous connections between processing modules. The kinds of communication that occur over this network include queries from agents, their responses, and copies of packets in both directions.

The ***Distribution System*** is actually a ring serial connection of Transputer ports. Over this link is a steady stream of packet names, both those packets that are ready for reducing and those that are free and can be grabbed for use in the construction of new graph nodes. Each module monitors the stream as it passes through it, removes packet names as required for its own internal workload, and inserts new ones as generated.

12.5.3 Packet Evaluation

Execution of a program on ALICE starts with the code for all programs stored in the PPSs and an initial graph (set of packets) representing the initial expression to be evaluated. These are circulated on the Distribution System and picked up by Reduction Agents, where they are checked for reducibility. The rules for a reducible packet (for HOPE programs) are:

1. Its function field does not contain a constructor function.
2. The arguments are reduced enough so that a rewrite rule can be decided on and executed.

For packets with programmer-provided functions (those defined by dec statements), the code associated with that function must be fetched over the Interconnection Network before this test can be performed.

For HOPE programs this second rule usually means that each argument is evaluated far enough so that it is either a constant value or a data structure where the constructor function is known. This is to permit determination of which one of the patterns associated with the equations for that function has been satisfied. For example, for the **flatten** function of Figure 12-8, this means evaluating the argument until it is known whether the topmost constructor function for the argument is **nil, leaf,** or **tree.**

If some argument is not yet sufficiently evaluated, the Reduction Agent increments this packet's Pending Signals Count, and adds the packet's name to the argument packet's Signal List. Processing on this packet is then suspended until the Pending Signals Count goes to zero.

If all arguments are sufficiently reduced, the appropriate rewrite rule can be chosen (the appropriate equation in the HOPE sense) and a new graph created from it. If this graph is either a simple value or rooted in a constructor function, all the packets in the original packet's Signal List are signaled. This causes their Signal Count to be decremented, and if any of these reaches zero, those packets become reducible and are added to the stream on the Distribution System. The reference counts for the argument packets are also adjusted as necessary at this point. Arguments whose values are no longer needed have their reference counts decremented; those that are used more than once have theirs incremented appropriately. This is done via messages from the Reducing Agents to the PPSs.

When a new graph is created, packets to contain new nodes are chosen from those marked as free that are circulating on the Distribution System. The Reducing Agent selects such a packet name and then sends a copy of the new packet's contents over the Interconnection Network to the appropriate PPS module.

12.6 REDIFLOW

(Keller and Lin, 1984)

Another approach to a highly parallel collection of identical processors executing a function language is the ***Rediflow system*** (where Rediflow is a contraction of Reduction and Data Flow). Here each node is an entire computer with some local memory and interconnection ports to other nodes. The number of such ports and the actual pattern of interconnection is variable; the systems studied by Keller assumed four ports per processor, permitting interconnection patterns that range from simple two-dimensional meshes, through trees, to more general interconnections involving perfect shuffle patterns. Communication over these ports is via packets of one of three kinds:

- "Fetch packets," which contain the address on a memory location in some other node that is to be read, and the address of the location in the originating node that is to receive the value.

- "Forwarding packets," which contain an address and a value to be stored in that address. This is often sent as a response by a node receiving a fetch packet.
- "Apply packets," which consist of a closure and a pointer to a tuple of arguments to be given to the function defined in the closure.

The assumed architecture of each node is an abstract machine that is a good compiler target for ***FEL*** (***Function Equation Language***)—a language where a program consists of a set of mutually recursive function definitions with a set of built-ins that include many of Backus's functional forms. The major instruction of interest here is an INVOKE, which indicates that a function is to be applied to a tuple of arguments. This instruction generates an apply packet, which can be placed in one of two queues in the processor. The "Local Queue" contains packets which can only be executed on the node they were generated on. The "Apply Queue" contains packets from either this node or from neighbors that have decided to migrate extra work to this one. Normal operation of either queue is first-in, first-out (FIFO), with new packets added to the back of a queue and the one at the front the next candidate for execution. This tends to have nodes generate as many packets as possible, enhancing the possibility of spreading them around to other processing nodes and thus increasing parallelism.

What is particularly novel about the Rediflow machine is the conditions under which these apply packets are permitted to migrate around the system (Figure 12-15). Each node computes an estimate of its current load (termed its ***internal pressure***) as the sum of the lengths of its local and apply queues and a constant divided by the fraction of memory still available. With this, the pressure to migrate tasks out of a node increases as either the number of tasks queued in the node or the memory utilization increases. Whether or not this task migration actually occurs depends on the pressure signaled by the neighbors of the node (their ***propagated pressure***). If this signaled pressure is less than the internal pressure (and that pressure is greater than some minimal threshold), then a task will be moved from the apply queue of the current node to that of the neighbor with the lowest propagated pressure. If all the propagated pressures from neighboring nodes are greater than this one, then no tasks migrate.

The actual pressure propagated from one node to its neighbors (to help them govern their decisions) is a function of both the internal pressure and the smallest pressure of any neighbor. If the internal pressure is low enough (below some threshold), then this node will signal a pressure of zero to its neighbors. This indicates that it is willing to accept new tasks. If the pressure is above the threshold, the propagated pressure is a function of the minimum pressure of any neighbor (with the diameter of the network of processors as an upper bound). This heuristic helps tasks flow from one side of a node through it to lower-pressure neighbors on the other side, even if there is a fair amount of work already present at the intermediate node.

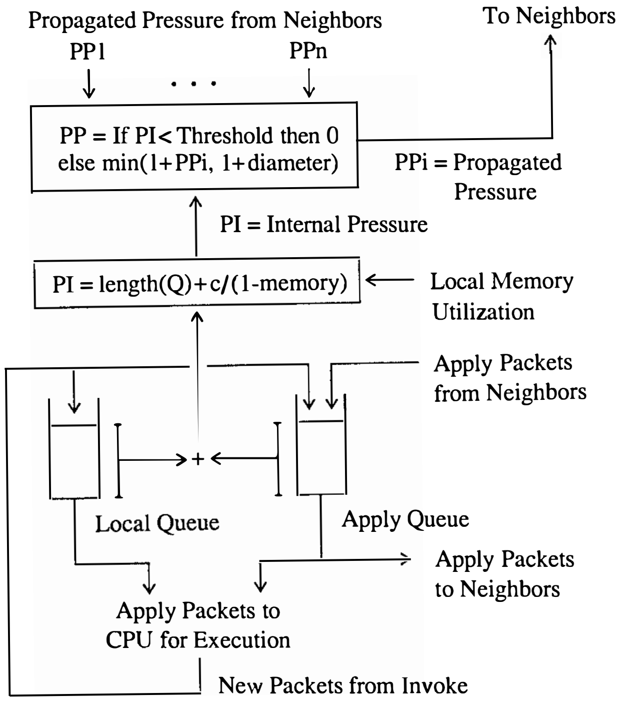

Propagated Pressure from Neighbors
PP1
. . .
PPn
To Neighbors
PP = If PI< Threshold then 0
else min(1+PPi, 1+diameter)
PPi = Propagated Pressure
PI = Internal Pressure
PI = length(Q)+c/(1-memory)
Local Memory Utilization
Apply Packets from Neighbors
+
Local Queue
Apply Queue
Apply Packets to Neighbors
Apply Packets to CPU for Execution
New Packets from Invoke

FIGURE 12-15
Application packet management.

When the propagated pressure from a node's neighbors is high enough, this propagation pressure equation will prevent any new tasks from migrating to them. The tasks stay local to the node. Under such conditions, the node switches the protocol used to select tasks from the Apply and Local Queues from FIFO to LIFO ("last-in, first-out"). This tends to generate a more storage-efficient mode of operation, in which applications are worked until they are completed before proceeding to some other applications.

Experiments with several benchmarks indicate that a speedup of roughly 60 percent of the number of processors is feasible for meshes of up to perhaps 50 node configurations.

12.7 FUNCTION CACHING

(Keller and Sleep, 1982)

The property of ***referential transparency*** means that no matter how often one recomputes the same pure functional expression, the result is always the same. This opens up the interesting implementation possibility of "remembering" the result of an application in a form such that the next time the same argument is presented to the function, this remembered value can be restored and an entire recomputation avoided. For example, in

letrec **fib**(*n*)=(if *n*<2 then 1 else **fib**(*n*−1)+**fib**(*n*−2)) in **fib**(100)

fib(100) expands to **fib**(99)+**fib**(98). Here **fib**(99) also expands to **fib**(98)+**fib**(97). If the result of the first occurrence of **fib**(98) is cached and then recalled, the considerable computation involved in the second occurrence is saved. The savings are even greater when one looks at all the intermediate points.

Because of its resemblance to a cache in the memory system of a conventional computer, such implementations are often called ***function caching***. The functions that implement them have also been called ***memo functions***, because they "take a note" of what a function produces as a result of an application.

The most obvious kind of such implementation is to create an initially empty data structure when a function is defined, and modify it every time the function is called. Figure 12-16 diagrams a simple example in which the identifier *fib-cache* is bound to an almost empty ***association list*** (a list of argument-value pairs) *before* the function **fib** is defined, and is modified *during* each execution of **fib** that has a new argument. Note that this predefinition of *fib-cache* is absolutely necessary for the scheme to work.

In the approach shown here, new argument-value pairs are added just after the first such pair in *fig-cache* by replacing the cdr of the first list cell by a pointer to a new cell (the result of the **cons**) whose cdr itself points to the rest of the old list. This approach was chosen for simplicity of exposition; more complex approaches that may minimize search time are also possible.

12.7.1 Applicative Caching

Although it is relatively simple and efficient, the caching approach of Figure 12-16 is definitely dependent on side effects. For that reason it is often called ***nonapplicative caching***. An alternative that relies on no-side-effect operations is possible, especially when the arguments of the function can be mapped onto the positive integers. The basic approach is to generate a ***stream*** of values from the function, and then, when a particular argument is applied, to use that argument value as an index into the stream. If the stream extends that far, the appropriate value is already available. If the argument is for a value that has not yet been completed, the closure at the end of the stream is repeatedly **force**d until it has been extended far enough.

```
let fib-cache = '((1 1)) in ....fib(10)...
   where fib(n) = let z = assoc(n,fib-cache) in
         if null(z)
         then rplacd(fib-cache, cons(cons(n,q), cdr(fib-cache))
             where q = if n<2 then 1 else fib(n-1) + fib(n-2)
         else cdr(z)
```

Note: the function assoc is defined in Chapter 6.

FIGURE 12-16
Nonapplicative caching.

```
letrec cache(f) = (λn|select(n,cache-stream))
   where cache-stream = process(f,integers(0))
and fib = cache((λn|if n<2 then 1 else fib(n-1) + fib(n-2))
in ... fib(10) ...
```

FIGURE 12-18
Caching functionals.

Figure 12-17 gives such an example. A stream is created by applying the **process** function from Chapter 9 to the stream of nonnegative integers. This stream is initially unforced and represents the cached function. A ***proxy function*** with the name of the desired function is defined to use its argument to index into the stream. The indexing function **select** forces the stream when necessary. A third auxiliary function defines the actual computation. Note that the recursion uses the cached values when possible.

As before, the stream must be defined separately from and before the other functions. Otherwise, each time the **fib** function is called, the entire stream will be regenerated from scratch.

12.7.2 Caching Functionals

Given that all of the support functions for Figure 12-17 are pure functions with no side effects, it is possible to go one step further and develop relatively generic higher-order functions called ***cache functionals,*** which accept descriptions of the functions to be cached and return self-caching versions of them. Figure 12-18 diagrams an example. When executed, it creates a stream that computes the function's value and inserts it in a function object that accepts one argument as an index into the stream. This object is the value bound to the function name.

Other caching functionals are also possible and in fact can be built into a system so that they are always invoked, or only on the basis of programmer directives.

```
letrec fib-cache = process(fib1,integers(0))
and fib(n) = select(n,fib-cache)
and fib1(n) = if n<2 then 1 else fib(n-1) + fib(n-2)
and select(n,cache-stream) = let x = force(cache-stream) in
    if n = 0 then car(x) else select(n-1,cdr(x))
in ... fib(10) ...
```

Note: the stream integers and functions process and force are defined in Chapter 9.

FIGURE 12-17
Applicative caching.

12.8 PROBLEMS

1. Develop an abstract program for an interpreter of an interesting subset of Logo. Use whatever abstract syntax functions might be appropriate. Develop a Logo program to interpret a command list.
2. Using Backus's FP system, find an equivalent functional form for $[\mathbf{f}_1,\ldots,\mathbf{f}_n] \circ [\mathbf{g}_1,\ldots,\mathbf{g}_n]$.
3. Prove the laws of Figure 12-6.
4. Convert the definition of the functions **member** and **length** from Figure 5-15 into FP functions. Assume that the functions **eq** and **null** are available. Apply the recursion removal theorem to your definition of **length.**
5. Write the **towers-of** and **toh** functions from Figure 5-18 into FP form. Does the recursion removal theorem apply to **toh?**
6. Write short HOPE functions for the following:
   a. **Towers-of** and **toh** functions
   b. A generic **reduce** function
   c. An interpreter for the simple lambda calculus of Figure 6-5
7. Write a short HOPE program that includes a datatype that might be either a single number, a tuple of two numbers, or a triple of numbers, and a function that accepts objects of such types and returns a single number representing the sum of all the numbers present in the argument.
8. Describe how you might augment the SECD Machine to handle HOPE programs, especially the pattern-matching part. Describe any assumptions you have to make about what the compiler can do.
9. Assuming the following sizes for tree processors on a VLSI chip, how big must a chip be to contain a 2-leaf tree, 4-leaf tree, 8-leaf tree, . . . ,2^n-leaf tree:
   a. Both leaf and node processors are n by n μm.
   b. The wires that connect node-to-leaf processors or leaf-to-node-to-node processors are n μm wide.
   c. A leaf processor can be put right next to a node processor without any additional connecting wire needed.
10. Draw out the packets that would be generated during an ALICE execution of the **flatten** function of Figure 12-8 when its argument is of the form:

    tree(tree(leaf(1), tree(leaf(2), leaf(3))), tree(nil, leaf(4)))

    Assume that only one reduction agent processor is present.
11. In general, how much parallelism do you think there is in ALICE execution of expressions involving **flatten** versus other functions such as **map** or **reverse?**
12. Describe how the ALICE machine described here might handle lazy evaluation.
13. Assuming that we cached the argument and result for each call to **fib** the first time such a call was made, how many calls would be avoided by functional caching during the computation of **fib**(6)?
14. Convert Figure 12-16(*a*) into SECD code that does in fact modify the binding to *fig-cache* each time a new argument to **fib** is encountered.

CHAPTER 13

A LOGIC OVERVIEW

Consider the typical image of a computer as a "black box" to which one submits inputs and gets results. Preparing programs for such systems involves developing detailed expressions which relate the inputs to the outputs. The computing hardware itself gets no overall view into what is desired; it simply does blindly whatever assignments and substitutions (or equivalent) are called for, and in the specified order. This is true even of the function-based computing models discussed in the first half of this book.

In contrast, consider how we humans prepare such programs (or do many other things in our daily lives). We "reason" about things and actions, and develop programs, plans, and other results based on the perceived consequences of their various interactions. Intuitively, this process is also some form of computation, albeit of an entirely different nature than what most programmers are accustomed to. The computations seem to bear on the overall "relationships" of objects rather than internal details of how they are put together.

The field of logic-based computing addresses such processing head-on (see Kowalski, 1979; Cohen and Feigenbaum, 1982). Computation here corresponds to methods of "formal reasoning," where the objects of such reasoning are statements about the world, such as "facts," or "if-then rules" about the interaction of other facts or if-then rules. In turn, "reasoning" or "logic computing" will use one set of such statements to predict or compute the truth or falsity of other sets of statements of more direct interest.

This chapter introduces logic computing. It starts with an overview of the basic logic computational model (an ***inference engine***) and proceeds to a brief but more formal introduction to systems of logic, with details of the simplest, namely, ***propositional logic***. This is followed by discussions on "classical" inferencing methods, where ***inferencing*** is the key computational step that derives the truth of one set of statements from another.

The material in this chapter is conceptual only; no attempt is made to describe exactly how to implement the techniques in real computing languages and machines. Such topics will be discussed in later chapters. At the other extreme, no attempt is made to flesh out fully the appropriate mathematics of logic. Most of that will be left to the standard mathematics references, with only what is really relevant to computation covered in the next chapter. The reader's major goal from this chapter should be to understand how the various concepts interrelate and how problems can be expressed and solved using them intellectually.

13.1 INFORMAL OVERVIEW

Logic computing deals with the processing of statements about the world (or some abstract version of it). This section informally introduces notation used to describe such statements, typical ways in which the statements can be processed, and several examples that demonstrate the kinds of problems that are well suited to this mode of computation. It is assumed that the reader is familiar with the relevant material from the introductory chapters of this book, particularly that concerning ***relations***.

13.1.1 Definitions

The primary kind of statement with which logic computing deals is a ***fact***, that is, an expression that some object or set of objects satisfies some specific relationship. Examples of this might be "The price of this book is $49.95," "At 5 p.m. on April 15 it is raining in Endicott," or even "The factorial of 3 is 6."

The classical mathematical approach to dealing with such facts, particularly when there are a lot of them constructed similarly, is via a ***relation***. From Chapter 2, a relation is a set of ***tuples***, where each tuple represents an ordered set of objects that together share some property or relationship. The set represents some subset of a Cartesian product of other sets. In a real sense, such sets of tuples represent organized ways of representing similar facts; i.e., if some tuple **t** is in some relation **R**, then "it is a fact" that the set of objects in **t** "have the property" **R**. For example, the relation **weather** may have tuples consisting of several components, such as time, location, temperature, pressure, humidity, wind, etc.

Another example might be the relation **hair-color**, with tuples consisting of a person's name and the color of his or her hair. The statement "Mike's hair is red" thus represents the fact that the tuple (Mike,red) is a member of the relation **hair-color.**

This concern about membership of objects in relations gives us an operative definition for facts and ***truth*** or ***falsity***. A simple statement like the above is a ***fact*** (equivalently, is ***true***) if the tuples mentioned in it are in the specified relations. If this is not the case, the statement is ***false***.

Simple relations are not the only kinds of statements that can be formed and processed. Other kinds can discuss under what conditions various potential facts may be true or false, such as "If **A** is a fact, then so is **B**," or "Either **A** is a fact, or **B** is not, or both." These also can be true or false, and logically combined or processed with other statements very similarly. Truth in such cases means that there is never a combination of objects that somehow causes a contradiction.

A simple example of this is the statement "**A** is a fact, and **A** is not a fact." A tuple cannot be both in a relation and not in it.

As a more common example, an ***if-then rule*** is a common name given to a statement that tells something about how specific sets of related tuples of objects can predict that other related tuples will satisfy other relations. The **if** part of the statement identifies the conditions which must be met, and the **then** part specifies what then can be assumed true without further work. Very often, there are ***logical variables*** that occur in both parts, which "generalize" the statement by making it hold for a vast number of objects that might be substituted for them. A simple example of this might be: "For all x, if x is a living person and x is going to Endicott today, then x should bring an umbrella." The variable x in this if-then rule applies to all living humans who may be going to Endicott today (potentially a large number).

Such statements can be true in two different ways. First, they can be true because all the referenced relations are completely defined, and one can exhaustively verify that all combinations of tuples that satisfy the **if** conditions also satisfy the **then** conclusions. Second, and more interesting from a computational viewpoint, they can represent a description of relations which are not described elsewhere. If we assume that such rules are true by themselves, then they automatically define parts of the relations they are discussing.

For example, if we assume that the relation **has-hair** includes anything that has a fuzzy surface, such as a rug, and if we make the standard definition of an **animal,** then a statement of the form "If x has hair, then it is an animal" must be false, because the object **rug** satisfies the former but not the latter. On the contrary, if we have no prior explicit definition of the **animal** relation, and if we are stating that the above statement is true, then the statement itself automatically identifies part of the objects in the relation **animal,** and this includes things like rugs.

Note that our definition of ***truth*** and ***falsity*** is somewhat different from a conventional one. Our definition is essentially recursive and is based on the definitions we wish to impose on the underlying relations. A complex statement is true if and only if its substatements are appropriately true, and the simplest kind of statement is true only if it represents some collection of objects that represents one or more tuples in some relation. A complex statement is false if its negation can be proved.

This last comment about falsity needs some further amplification. Proving that the ***negation*** of a statement is true requires careful definition of what negation means. This will be done later. However, whatever it is, it is different from saying that a statement is false simply because we fail to find a proof that it is true. For example, consider a logic program that states there are two possible conditions of weather, sun or rain, and that it is sunny in Boulder. From this alone it is impossible to show that either "It is raining in Endicott" or "It is not raining in Endicott." Thus we cannot say that either statement is true or false.

With these definitions in mind, ***reasoning*** is thus the process of deciding or ***inferring*** the truth or falsity of new statements or facts based on assuming that some other set of statements or facts is true or false. In a real sense it deals with the computation and testing of relations, and tuples that might belong to them, ideally without exhaustive enumeration and searching of the relations.

A statement that is shown to be true as the result of such a process is said to be an ***inference*** from the original set.

As an example of this process, given the statements:

- If x is the father of some child, then x is a parent.
- Larry is the father of David.

we could "infer" that "Larry is a parent" without having to look at all the tuples of the relation **is-a-parent** to see if the object "Larry" is there.

Note that there may be several kinds of reasoning, such as "common sense," "legal," and "scientific," all of which we humans consider valid at different times, and all of which may operate somewhat differently, with the variations dealing with the conditions under which inferences can be made and the rigorousness and order in which logical arguments are constructed. These conditions are termed ***inference rules,*** and the procedures that guide the construction of inference chains are called the ***decision procedure.***

The term ***inference rule*** should not be confused with "if-then rule." The latter is simply a common kind of logic statement; the former is a description of how to derive the veracity of one statement given that others are true.

Now, the term ***logic*** as used by mathematicians and philosophers refers to the formal study of inference rules and decision procedures. It deals with the "form" of an argument about statements rather than the content or meaning of any particular statement, relation, or object.

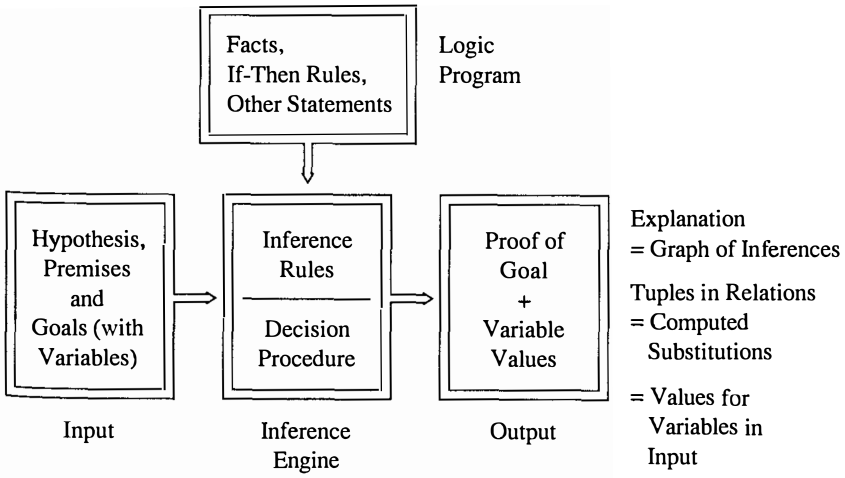
13.1.2 Inference Engines

Finally, ***logic computing*** deals with automating some set of inference methods. Our vision or model of the computer is thus a "reasoning agent" to which we present ***logic programs*** consisting of sets of statements to be assumed true, and ask questions about what can be inferred from them. These logic programs represent the relevant relations which in turn define the world of interest to the programmer. The types of input questions may include ***goals,*** which are statements that we would like proved true; and/or ***hypotheses*** or ***premisses*** (spelled with two s's), which are additional statements to be assumed true just for this run of the program.

The output from the computer can include not only a true/false indication about the goal, but also the sequence of inferences that led to it (an ***explanation*** of the answer), and often specific tuples that were found in the process of doing the inferencing.

Inside the computer, activity consists of a series of ***logic computations*** which together find a sequence of inferences joining the program to the statements representing the question to be answered. The combination of hardware and software that actually performs such computation goes by the name ***inference engine.***

Figure 13-1 diagrams this overall model. What we will study in the rest of this book is the theory behind the apparatus making up the inference engine, and how such engines can be created in real computers.



An inference engine is:

- sound if it cannot infer a false result.
- complete if it can infer all true results.

FIGURE 13-1
Logic computing.

13.1.3 Computing Tuples

The concept of "computing" tuples that make various inferences possible is worth further discussion. Very often in real applications the goal presented as input to a logic computation consists of a tuple (for some relation) where only some of the components of the tuple have values. The rest are identified by ***free variables*** (also called ***logic variables*** or ***identifiers***) with no initial values. Activating the inference engine causes it to try to prove via a series of inferences that some version of the given tuple is in fact in the specified relation, if the variables in the tuple are replaced by certain values. The inference engine tries not only to prove the given statement but to "fill in the blanks" as it does so. It tries to compute a ***satisfying substitution*** for the initially free variables.

Such a computation may be ***nondeterministic*** in that there may be many different substitutions for the variables that will make the goal true, each with its own proof sequence. In such cases, unless otherwise controlled, the inference engine may return as soon as it has any one of them. Very often it may be impossible for the inference engine to determine if, or how many, other solutions exist, or how they relate to the one found. It may be up to the user to request that other answers be sought, and that any postprocessing be done to find the "one" really desired.

This concept should be contrasted with function-based computing, where the *use* of substitutions of values for variables is the primary computational step. In logic programming, it is the *computation* of such substitutions, not their use, which plays a central role.

13.1.4 Examples

As an example of some of these ideas, consider the "logic program" of Figure 13-2.

If the initial input is "Given that Gary is going to Endicott, then what object z should he get?," then "executing" this program clearly involves a chain of inferences where the variable z is given either the value "umbrella" or the value "raincoat." The order in which these answers are found is a function of the precise internal details of the decision procedure, and often is not totally predictable by a simple model of the system's operation in the user's mind.

As another example, consider the set of statements in Figure 13-3.

- If x is going to y, and it is raining at y, then x should get an umbrella.
- If x is going to y, and it is raining at y, then x should get a raincoat.
- If x is going to y, and it is snowing at y, then x should get boots.
- It is raining in Endicott.

FIGURE 13-2
Sample logic program.

- Jim is-father-of Mary
- Mary is-mother-of Tim
- Mary is-mother-of Mike
- Peter is-father-of Marybeth
- Roy is-father-of Peter
- Martha is-mother-of Roy
- Stephen is-brother-of Peter
- if (y is-mother-of x) then (y is-parent-of x)
- if (y is-father-of x) then (y is-parent-of x)
- if (x is-parent-of y) and (y is-parent-of z) then (x is-grandparent-of z)
- if x is-parent-of y then age-of(x)>age-of(y)
- age-of(Tim) = 3

FIGURE 13-3
Some sample statements.

The first few statements are facts which give specific tuples of objects (people in this case) that are members of specific relations. The middle statements determine conditions under which certain tuples are members of other relations. In a real sense, they tell us how to "compute" parts of the relation **is-parent-of.**

Remembering that a function is nothing more than a special form of relation, the final statements tell something about how the function **age-of** behaves under certain inputs. It is a description of some of its properties, not how to compute it.

Note also that we have chosen here to express the statements in an ***infix*** notation and could just as easily expressed them in the more conventional, but totally equivalent, ***prefix*** form, such as:

implies((and(parent(x,y**), parent(**y,x**)), grandparent(**x,z**))**

or even in a mixed infix/prefix form, such as:

parent(x,y**)** $\wedge$ **parent(**y,x**)** $\Rightarrow$ **grandparent(**x,z**)**

When using prefix notation, most people feel more comfortable with relation names that drop the beginning **is-** and ending **-of.**

For logic programs like Figure 13-3, the kinds of question statements that could be asked include:

- Is Jim a grandparent of Tim?
- Who are all the grandparents of Tim?
- Who are the children of Mary?
- Who are the parents of anyone who is a grandparent of Mike?
- Why is Roy the grandparent of Tim?
- Compute all relationships of anyone to anyone.
- What can be said about the age of Roy?

Next, consider a more numerically oriented problem, namely, the computation of factorials: See Figure 13-4.

A typical statement to use as a question for this program is "**fac**(3)=z?," for which the system should draw a series of inferences that prove **fac**(3)=z is true if z=6. Further, these inferences will look remarkably like the sequence of substitutions needed to derive **fac**(3) in a functional language.

A not so typical, but perhaps more interesting, question might be "Is there a z such that **fac**(**fac**(z))=720?" An appropriate logic computing system could solve this problem by essentially "computing backward" without explicitly being told to, something none of the other methods of computing discussed in this book can handle with any kind of grace.

Finally, consider the question, "Find all numbers which are the factorials of others." The above set of statements could be logically beat against itself repeatedly to discover first that **fac**(0)=1, then **fac**(0+1)=(0+1)×**fac**(0)=1, then....There are logic-based computing systems available today which can handle these kinds of problems.

13.2 FORMAL LOGIC SYSTEMS

(Mendelson, 1964, 1979; Clocksin and Mellish, 1984, chap. 10; Chang and Lee, 1973, chaps. 2, 3)

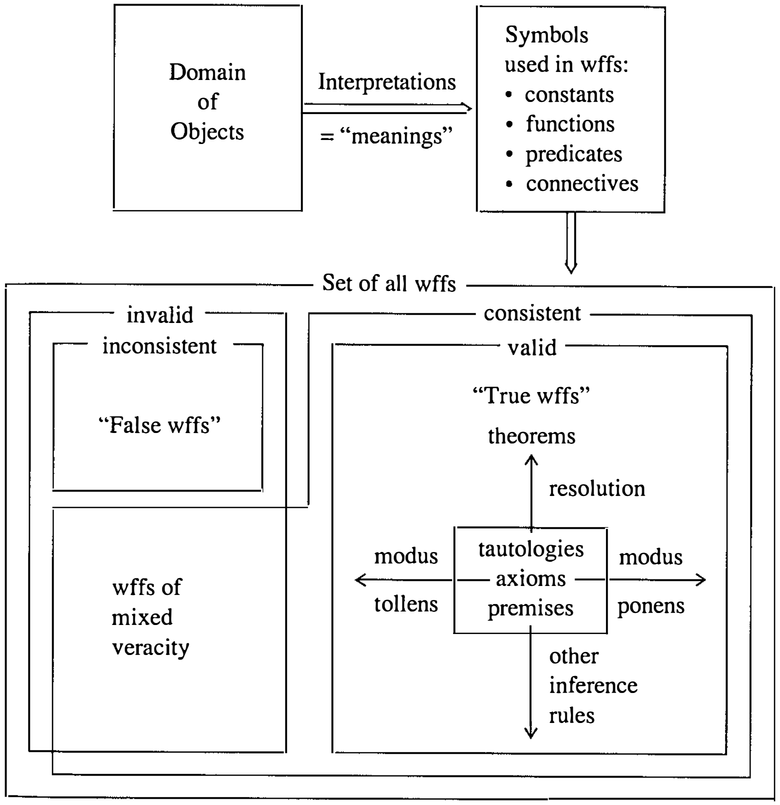
Any discussion of logic-based computing must begin with a good understanding of what logic is and how one talks about its concepts. As with functional and object-oriented computing, we start with a ***formal model of logic,*** often also called a ***system of logic, calculus of logic,*** or ***theory of logic.*** The following subsections introduce the general characteristics that such models must have. As a reference, Figure 13-5 diagrams pictorially the major terms to be defined gradually over the following subsections. The reader may want to flip back to it as he or she progresses through the rest of this section.

This material will lead in the next section to the simplest such model, the ***propositional logic.*** The next chapter describes a more commonly used model, namely, the first-order predicate logic.

As with any other model of computing, we start by addressing the two major components, syntax and semantics. Syntax deals with how to write valid expressions; semantics refers to what such expressions "mean."

- fac(0) = 1
- if n>0 then fac(n) = n×fac(n−1)
- if n<0 then fac(n) = "error"
- if z=n−1 then n=z+1

FIGURE 13-4
A logic program about factorials.



(see Figure 13-7 for definitions)

FIGURE 13-5
A logic system.

Generically, the ***syntax*** of a logic model revolves around ***logical expressions*** built up from a set of more basic ***symbols.*** In most logic texts the subset of symbol strings that represent valid expressions often go by the term ***well-formed formulas*** (or ***wff***s for short). Individually they may also be called a ***statement, proposition,*** or ***sentence.***

These logical expressions will often resemble in syntax the functional expressions of the last division. Further, we will often treat them in a similar fashion; namely, given the definition of certain relations, such expressions may be equivalent to (or reduce to) a value of either true or false. However, unlike functional expressions, we will often use logical expressions in a backward fashion; namely, if we state that some expression is true, this will put some constraints on definitions of internally named relations.

Usually, a relatively simple set of BNF-like rules are sufficient to describe the syntax of wffs. The statements, propositions, facts, if-then rules, etc., of a particular problem are then constructed as wffs according to these syntax rules.

Typical syntaxes include allowances for:

- ***Constants*** (specific objects)
- ***Functions*** applied to such objects to yield other objects
- ***Predicates*** to test for membership of tuples of objects in relations
- ***Identifiers*** (or ***logic variables***) as in lambda calculus to stand for "unknown" objects
- And other operations which combine these, plus place constraints on values that various variables may take on

As discussed in Chapter 1, a predicate and a relation are closely related. Very often, the name of a predicate is the same as the name of some relation, and its use will be to test if some tuple is in that relation.

In Figure 13-3, for example, "Jim," "Mary," "Tim," "2," etc., are constants, while **age-of** denotes a function, and **is-mother-of, is-father-of,** "=," " ≥ ," etc., represent predicates over certain relations. The "if-then," **and,** and **implies** terms represent ***logical connectives.***

The ultimate purpose of a system of logic (or logic program) is to divide the universe of wffs into pieces; those to be considered "true," those that are "false," and those about which no absolute statement can be made. In a very practical sense this is largely equivalent to specifying and computing under what conditions what objects might be members of what relations or combinations of relations. The user or developer of a logic program has this partitioning in mind when he or she designs the program, with the exact specification of the dividing line identified through the ***semantics*** of the logic program.

This identification can occur in two ways. First, some particular subset of wffs (called the ***axioms*** of the model) may be simply assumed true. These axioms themselves come in two flavors: those that are inherently true by their very construction (called ***tautologies***) and those whose truth does not flow from their construction but are to be assumed true anyway (***premisses*** or ***hypotheses***). An example of a tautology that occurs in most systems of logic is "For any *x*, *x*=*x*." On the other hand, "If *x* is human then Adam is an ancestor of *x*" may be assumed true in a particular logic system, but does not follow from its underlying construction as a wff. In a sense these latter axioms represent the "background knowledge" of the theory.

In most real logic programming languages the user provides these axioms (particularly the latter kind) as the logic "program."

13.2.1 Rules of Inference

Axiom specification by itself provides only part of the information needed to accurately specify the "dividing line" the programmer is after. In a properly written program, other parts of the language's semantics "extend" these statements to cover other statements which fully define the "true" or "false" sides of the wff set. This extension is via ***inferencing,*** which uses specific patterns of known sets of wffs to predict or deduce the veracity of other wffs. Each such generic pattern is termed an ***inference rule,*** and is roughly equivalent to a relation over ordered subsets of wffs. More than one inference rule may be present in a system, and inference rules may overlap each other.

The actual process of inferencing involves finding some inference rule **R,** some subset **X** of wffs already in the "true" set, and some other wff α such that $\mathbf{X} \cup \alpha$ is in **R.** Given our definition of **R** as a set of wffs, this means that the wff α should also be considered true in the system under discussion. When this occurs, we say that α is an ***inference, logical consequence, conclusion,*** or ***direct consequence*** from the set **X** using the inference rule **R.**

A common inference rule is ***modus ponens.*** This rule takes some wff **A** and some other wff of the form "if **A** then **B**" and infers that the wff **B** is also true. It represents the relation:

modus-ponens={(**A,** "if **A** then **B,**" **B**)|**A** and **B** any valid wffs}

Note in this case the use of the term "rule" in both "inference rule" and "if-then" rule. The former is a characteristic of the computational model; the latter is a specific kind of wff. Although the terms are different, this text will often drop the distinguishing adjective and just use "rule," with the appropriate meaning to be clear from context.

A *proof* is a sequence of wffs $\alpha_1, \ldots, \alpha_n$ such that each α_k is either an axiom or a direct consequence of some subset of the prior α_j's, j<k.

A ***theorem*** is a wff α which is a member of some proof sequence, usually as the last wff. The notation $\Gamma \vdash \alpha$ indicates that α is a theorem in the system of logic under discussion (with Γ standing for the input set of axioms specified by the logic program).

When we are dealing with several systems or theories of logic at the same time, we will use a subscript on the "⊢" symbol to indicate which is applicable.

13.2.2 Properties of Inference Rules

The concepts of a theorem and a proof sequence together place some criterion on the rules of inference to be chosen by the designer of a logic system. Obviously, it should be impossible to infer a theorem which is not in the overall "true" set of wffs. A set of inference rules which permits no such false theorems is termed ***sound.*** On the other hand, we want the inference rules (if used properly) to permit proofs for all true wffs that are derivable from the axioms. Such a set is logically ***complete.***

In contrast, an ***unsound*** system permits some wff that should be false to be improperly designated as true. An ***incomplete*** theory is one for which some inherently true wff (a tautology) cannot be derived from the provided axioms and tautologies.

Slight variations of the theorem concept will also be of interest. For example, we may wonder if some wff α would be true if we added some set of wffs **Y** to the axiom set of some system. These **Y** wffs are termed the ***hypothesis,*** or ***premisses,*** and if a proof sequence exists which includes members of **Y** as axioms, then α is a ***consequence*** of **Y** and this proof sequence a ***deduction.*** The notation $\Gamma, \mathbf{Y} \vdash \alpha$ denotes the existence of such a sequence. It is roughly equivalent to showing that $\Gamma \vdash (\text{if } \mathbf{Y} \text{ then } \alpha)$.

Just because the designer of a logic system provides a complete and sound inference mechanism does not guarantee that the set of axioms provided by the programmer as the input logic program are themselves logically perfect. It is possible to have a set of axioms that permits derivation of a ***contradiction***; i.e., inference sequences are possible which show that some wff is both true and false. Such an axiom set is said to be ***inconsistent.*** A simple example would be a set of axioms of the form {**A**, if **A** then **B**, if **A** then **not** (**B**)}. Both **B** and **not** (**B**) can be inferred from this set.

In contrast, a ***consistent*** set of wffs is one that does not permit derivation of a contradiction. In most cases this is an important property of an axiom set, and one the logic programmer must strive to achieve if the results of computation from it are to be believed.

13.2.3 Decision Procedures

At this point it should be obvious that finding inferences and proof sequences is going to be an important part of logic-based computing. We will be specifying a set of axioms as program input, and asking various questions about other wffs and their validity under the given inputs. The system will then try various combinations of inference rules in an attempt to find an appropriate proof sequence. Usually an infinite number of such combinations are possible, and at best a handful that actually work. The others either prove wffs that are not of direct interest, end up in "dead ends," or get caught in infinite loops deriving either the same or similar wffs. Obviously we would like some guarantees on the operation of our system such that the questions we ask are answerable in finite time using some mechanical or algorithmic method that is implementable in a digital computer, but is more efficient than the "try all possible combinations" approach (often called an ***exhaustive search*** or ***British Museum search***). We also would like to be informed if the question posed has no answer (i.e., has no provable relationship to the current logic program).

Unfortunately, this is not always possible. There are systems (including many of the most interesting ones) which are ***undecidable***; that is, there is no algorithmic approach for determining whether a particular wff is either true or false, other than an exhaustive search.

Systems for which we we do have some guarantees are ***decidable.*** In these cases we have algorithms which know how to choose the appropriate sequence of inference rules and wff subsets in some predictable period of time, and which are guaranteed to find a proof sequence if one exists. Typically no guarantees are given about what happens if the question asked is false or has no proof. The system may simply respond "I don't know" or perhaps loop forever.

Such selection algorithms are termed ***decision procedures,*** and when coupled with the designated set of inference rules, form the ***inference engine*** of the logic system. They are the focus of detailed discussions in later sections.

13.2.4 Interpretations

Another key part of the semantics of a logic expression is the specification of an ***interpretation*** that "maps" a "meaning" to each symbol (constant, function, or predicate) that makes up the expression. This starts with a ***domain*** or set of domains that defines the set of possible values or objects to be dealt with. An interpretation then associates one of these domain values with each unbound symbol used as a constant of any sort. Similarly, symbols used to represent functions are assigned function definitions (mappings) from some domains into others. Symbols used as predicates map into tests on relations over these domains. Variable symbols represent placeholders whose actual values may be taken from the domains.

In a real sense, specifying an interpretation is similar to specifying a substitution.

As an example, consider the symbol **Hal.** One interpretation of this symbol might be some particular human being who is very good at imitating Mark Twain. Another interpretation might associate the symbol with an advanced computer used on some fictional spaceship.

Likewise the function "+" might receive one interpretation which defines it as a relation from $\mathbf{Z}\times\mathbf{Z}$ to **Z,** with the elements of the tuples corresponding to the rules of integer addition. However, an equally valid interpretation might assign it the meaning as a function from {T,F}×{T,F} to {T,F} which obeys the truth table for ***inclusive or.***

Finally, the relation **is-parent-of** might be interpreted as the obvious one over cross-products of living beings. However, it might also refer to root cells in an s-expression.

The key point here is that there is often an infinite number of interpretations and combinations of interpretations which can be given the symbols used in a wff or set of wffs. In general, we will be interested in how these different interpretations affect the truthfulness of the wff. The prime definition thus is that a wff is ***true under an interpretation*** if the result of ***evaluating*** the wff under the interpretation is true. This is equivalent to saying that the specified interpretation ***satisfies*** the wff.

The process of evaluating a wff starts with the substitution of all simple constants in the wff by their "meanings" as objects from the interpretation. It continues to substitute recursively any expression involving a function symbol for which all of its arguments are known (as objects). The value replacing this subexpression is the object defined by the function's mapping (again from the interpretation). The resulting expression is a logical combination of predicate symbols with known "objects" as their arguments. The initial interpretation again provides a mapping from these arguments to the true/false set, and the logical connectives then combine them to give a final value.

Figure 13-6 gives an example of a wff and a particular interpretation that one might use. Although this is the most common one, it is not the only such interpretation; there is literally an infinite number of other ones (e.g., assume that all the odd numbers are equivalent to 1 and all the even ones are equivalent to 0).

Interpretations involving variables, quantifiers, and other items will be discussed later.

The concept of evaluating a wff under an interpretation gives a method of comparing two wffs. We say that two wffs are ***equivalent*** if they evaluate to the same true/false values for every possible interpretations.

Obviously, different interpretations may affect whether or not we want wffs to be considered true. Given a set of axioms and inference rules, a particular wff is ***satisfiable*** or ***consistent*** if it is true under at least one interpretation. Such an interpretation is said to be a ***model interpretation*** which ***satisfies*** the wff.

If the wff is true under all interpretations, we say that it is ***valid*** or, equivalently, a ***tautology***. In contrast, it is ***invalid*** if it is not valid, that is, there is at least one interpretation which makes it false. Such an interpretation is said to ***falsify*** the wff. Finally, it is ***unsatisfiable, inconsistent,*** or a ***contradiction*** if it is false under all interpretations. Figure 13-7 summarizes these definitions. Note that it is possible for the same wff to be both satisfiable and invalid; at least one interpretation causes the wff to evaluate to true, and at least one interpretation causes it to evaluate to false.

Wff:

$(\mathrm{fac}(0) = 1)$
$\wedge\ ((n>0) \Rightarrow (\mathrm{fac}(n) = n \times \mathrm{fac}(n-1)))$
$\wedge\ (\mathrm{fac}(3) = 6)$

Symbol	Interpretation	
0	number zero	Constants
1	number one	
3	number three	
×	{((0,0),0),...((3,2),6),...}	Functions
−	{((1,1),0), ((2,1),1), ((3,1),2),...}	
fac	{(0,1), (1,1), (2,2), (3,6),...}	
=	{(x,x)}	Predicates
>	{(1,0), (2,1), (3,2), (3,1),...}	
∧	{(T,T)}	Connectives
⇒	{(T,T), (F,T), (F,F)}	

FIGURE 13-6
A satisfying interpretation.

As examples of these definitions, the wff (**A** or not **A**) is valid (any interpretation will make either **A** or not **A** true), (**A** and not **A**) is inconsistent for similar reasons, and **A** by itself is both consistent and invalid (interpretations exist which can make **A** either true or false).

The interpretation of Figure 13-6 is a satisfying interpretation for the specified wff. Notice what happens if we change the interpretation so that, for example, the definition of ">" is changed to include equality, if "×" is changed to be modulo 8, or if "∧"'s interpretation includes ((T,F),T) as a tuple. The interpretation is no longer satisfying. Thus this wff is consistent but not valid, invalid but not inconsistent.

The concept of an interpretation also gives an alternate and critically important definition of an ***inference*** or ***logical consequence***. A wff **G** is an inference from a set of axioms $\{\mathbf{A}_1, \ldots, \mathbf{A}_n\}$ if any interpretation that simultaneously satisfies all of the axioms also satisfies **G**. Note that there is no guarantee that the interpretations we find this way are the only interpretations that satisfy **G**; there can be others, particularly ones that do not satisfy some $\mathbf{A}_k$.

Most real logic-based programming languages include at best only weak mechanisms for specifying interpretations. Instead, they usually assume that the validity of the statements input by the user (and thus the wff set to be assumed true) is independent of the interpretation used. Consequently, the validity of wffs is highly tautological; they are either assumed true under all interpretations, or they are false. Thus, in wffs dealing with "Peter," "Tim," "Roy," and "Pi" as constants, for example, the system might assume nothing about them other than that they are separate objects. In the real world, of course, these character strings might relate to the names of people, boats, or even coded names of spe-

		# of Interpretations Evaluating to:	
A wff is:	if:	True	False
consistent, or satisfiable		≥1	don't care
valid, or tautology		all	0
inconsistent, unsatisfiable, or contradiction		0	all
invalid		don't care	≥1

FIGURE 13-7
Wffs and interpretations.

cial numbers. A properly designed logic system should maintain its logical soundness and completeness regardless of which interpretation is used.

This emphasis in turn identifies a highly desirable property of inference rules for logic-based computing: They should permit decision procedures that can prove tautologies without exhaustive testing of all possible interpretations.

13.2.5 The Deduction Theorem

The above definitions lead to a very important result (called the ***Deduction Theorem***), which will drive many of the inference engines we develop in this text. This theorem can be stated in one of two equivalent forms. First, the wff **G** is a logical consequence of the wffs $\mathbf{A}_1,\ldots,\mathbf{A}_n$ if and only if the wff

$$\text{if } (\mathbf{A}_1 \wedge \mathbf{A}_2 \ldots \wedge \mathbf{A}_n) \text{ then } \mathbf{G}$$

is valid (i.e., a tautology). Here we use the symbol " ∧ " for the logical "and" connective.

The second form of the theorem states that the wff **G** is a logical consequence of the wffs $\mathbf{A}_1,\ldots,\mathbf{A}_n$ if and only if the wff

$$\mathbf{A}_1 \wedge \mathbf{A}_2 \ldots \wedge \mathbf{A}_n \wedge \neg \mathbf{G}$$

is unsatisfiable (where "¬" is the symbol for logical negation). This is equivalent to saying that if we assume all $\mathbf{A}_i$ are true, and that **G** is false, then we get a contradiction. There is no interpretation that will make it true.

13.3 PROPOSITIONAL LOGIC

(Mendelson, 1964, 1979, chap. 1; Chang and Lee, 1973, chap. 2)

Perhaps the simplest system of logic that demonstrates most of the syntatic and semantic features mentioned above is the ***propositional calculus***. This is the first system taught in formal logic courses, and matches well many problems in such fields as digital logic circuit design.

In the syntax for this logic, a ***proposition, propositional letter,*** or ***atom*** corresponds to a "declarative sentence" that may take on interpretations of true or false. By convention, different propositions are assigned different uppercase letters. Figure 13-8 gives some examples that we will use throughout this section. Propositions are the only constant symbols out of which propositional statements are created; there are no functions or nonboolean constants.

One way of viewing a propositional letter is as corresponding to a relation with exactly one tuple. If we wish to talk about several tuples in the same relation, we must give each one a unique name and deal with them individually (for example, **C** and **D** in Figure 13-8 for the relation **I-have**). The use of propositional letters simplifies our descriptions somewhat, but at the loss of visibility into internal detail of the tuples making up a relation.

A = The car has gas.
B = I can go to the store.
C = I have money.
D = I have food.
E = The sun is shining.
F = I have an umbrella.
G = Today I can go on a picnic.

FIGURE 13-8
Some sample propositions.

Note that within the propositional calculus, these letters can take on interpretations (in the sense of the prior section) involving true or false values only; the fact that they have an "English-language" interpretation is irrelevant.

Construction of wffs from these letters involves combinations using ***logical connectives*** which correspond to functions over {T,F}×{T,F}→{T,F}. Several such sets of connectives are possible, but no one has any more computational power than any other. This means that any wff that can be formed in one such system can be mechanically translated into another system, usually by simple wff rewriting.

Figure 13-9 gives a syntax for the most common form of propositional logic. As with all our other notations, we will feel free to drop pairs of "( )" where their location is obvious and the resulting expressions are still unambiguous.

The standard interpretation for the connectives is most easily described in a ***truth table*** format (see Figure 13-10), which lists for each possible value for a connective's arguments what the resulting value is. The meaning given to these connectives should agree with the reader's experience with their use elsewhere, such as in digital logic design.

```
<proposition> : =  A|B|C|...

<binary-connective> : =  ⇒ | ∧ | ∨ | ≡

<unary-connective> : =  ¬

<wff> : =  <proposition>
          | ( <unary-connective><wff>)
          | (<wff> <binary-connective> <wff>)
```

FIGURE 13-9
BNF for a propositional logic model.

A	B	$\neg$A	A$\Rightarrow$B	A$\vee$B	A$\wedge$B	A$\equiv$B	A$\otimes$B
T	T	F	T	T	T	T	F
T	F	F	F	T	F	F	T
F	T	T	T	T	F	F	T
F	F	T	T	F	F	T	F

$\neg$ is "not"
$\Rightarrow$ is "implies"
$\vee$ is "inclusive or"
$\wedge$ is "and"
$\equiv$ is "equivalence"
$\otimes$ is "exclusive or"

Let X and Y be arbitrary wffs, then:

X$\wedge$Y is equivalent to $\neg$(X$\Rightarrow\neg$Y)

X$\vee$Y is equivalent to ($\neg$X)$\Rightarrow$Y

X$\equiv$Y is equivalent to (X$\Rightarrow$Y)$\wedge$(Y$\Rightarrow$X)

X$\Rightarrow$Y is equivalent to ($\neg$X $\vee$ Y)

FIGURE 13-10
Common logical connectives.

Most of these connectives combine two wffs together to form a larger one. For all but $\Rightarrow$ (***implies***), there is no particular name given these two subwffs, and in fact they could be switched around the connective without any effect on the result (the connectives are ***commutative*** functions). However, for " $\Rightarrow$ " the two subwffs cannot be switched, and they have been given explicit names. In the wff "**A** $\Rightarrow$ **B**," the wff **A** is the ***antecedent*** or ***condition,*** and **B** is the ***consequent*** or ***result.*** The entire wff often goes by the term ***if-then rule,*** with the antecedent representing the "if" part and the consequent the "then" part. The informal meaning of such a rule is that if we can prove the antecedent, then the truth of the consequent follows directly.

Note that with the interpretation of Figure 13-10 the connectives $\wedge$, $\vee$, and $\equiv$ can be replaced as shown by combinations of $\neg$ and $\Rightarrow$. Thus the former are redundant, and could be deleted from a system supporting only a minimal set of connectives {$\neg$, $\Rightarrow$}.

The semantics of this propositional model require definition of possible interpretations, a minimal set of axioms, and a set of inference rules. Interpretations were discussed above. Basically, a particular interpretation of a wff involves specifying assignment of a truth value to each letter and a functional mapping (usually Figure 13-10) to each connective.

In this case axioms which are tautologies are particularly easy to identify; we simply do a ***truth table analysis*** on the wff in question. In this table there is one row for each possible interpretation (set of true/false assignments to the propositions) and one column listing the resulting veracity of the total wff. If there is at least one interpretation (one row) whose result is true (matching value in the result column), the wff is satisfiable. If all possible interpretations (the entire result column) yield true, the wff is a tautology. If all possible interpretations are false, the wff is inconsistent. Figure 13-11 gives a simple example of a truth table analysis of a wff which is satisfiable but not a tautology.

Interpretations				
A	B	C	(A$\Rightarrow$B)	((A$\Rightarrow$B)$\Rightarrow$C)
F	F	F	T	F
F	F	T	T	T
F	T	F	T	F
F	T	T	T	T
T	F	F	F	T
T	F	T	F	T
T	T	F	T	F
T	T	T	T	T

FIGURE 13-11
Truth table analysis of **((A $\Rightarrow$ B) $\Rightarrow$ C)**.

Next, even though propositional calculus represents a particularly simple system, there are still an infinite number of wffs possible, and, surprisingly enough, an infinite number of tautologies, many of which have similar forms. To account for such similar wffs and, thus, to reduce the complexity of discussions involving tautologies, mathematicians often introduce ***axiom schemas*** which cover an infinite number of possible tautologies. In these schemas any of the lowercase "variables" can be replaced by arbitrary wffs in propositional logic, and the resulting overall wff will still remain a tautology. As an example, Figure 13-12 lists three common axiom schemas used in many proposition-based systems.

13.4 A SIMPLE INFERENCE ENGINE

All real propositional logic programs have more than one axiom or axiom schema in their basic definition, all of which we assume are true. As mentioned before, usually these axioms are not all tautologies; those that are not are included because we the programmers say so. Conceptually, this is equivalent to "and"ing all the individual wffs together into a single large wff and constraining that wff to evaluate to true. In turn, this constraint means that of all possible interpretations that might be possible, the only ones that will be used when trying to prove problems are those that made the original axiom set all true (particularly the nontautologies).

13.4.1 A Brute-Force Inference Engine

The result of the above discussion is that we can now outline a simple ***truth table-driven inference engine*** for propositional logic. Assume for a start that the kind of logic problem we want to solve is a deductive one: given some set **X** of axioms and some new wff **G** (for goal), we want to determine that it is a theorem in our system: namely, does **X**$\vdash$**G?** Since this is propositional logic, **G** is an expression involving propositional let-

Let a,b,c be arbitrary wffs.

The following describe sets of tautologies.

1. (a⇒(b⇒a))
2. ((a⇒(b⇒c))⇒((a⇒b)⇒(a⇒c)))
3. ((¬b⇒¬a)⇒((¬b⇒a)⇒b))

Sample truth table analysis of Schema 2:

a	b	c	a⇒b	b⇒c	a⇒c	x	y	x⇒y = Schema 2
F	F	F	T	T	T	T	T	T
F	F	T	T	T	T	T	T	T
F	T	F	T	F	T	T	T	T
F	T	T	T	T	T	T	T	T
T	F	F	F	T	F	T	T	T
T	F	T	F	T	T	T	T	T
T	T	F	T	F	F	F	F	T
T	T	T	T	T	T	T	T	T

Where

x = (a⇒(b⇒c))

y = ((a⇒b)⇒(a⇒c))

Some tautologies that follow from Schema 2 include:

- ((A⇒(B⇒C))⇒((A⇒B)⇒(A⇒C))
- (((A∨C)⇒(B⇒C))⇒(((A∨C)⇒B)⇒((A∨C)⇒C)))
- (((A∨C)⇒((B∧C)⇒C))⇒(((A∨C)⇒(B∧C))⇒((A∨C)⇒C)))
- (((A∨C)⇒((B∧C)⇒(D⊗A)))⇒(((A∨C)⇒(B∧C))⇒((A∨C)⇒(D⊗A))))

FIGURE 13-12
Axiom schemas in the propositional calculus.

ters and connectives. The operation of the inference engine is then rather simple:

1. Pick an interpretation (assignment of T and F to letters) that has not been tried before.
2. If all interpretations have been tried, we are done and **G** is a theorem.
3. If the interpretation does not satisfy the original axiom set **X**, ignore it and go back to step 1.
4. Try the interpretation on **G**, using the standard definitions of all connectives.
5. If the result is F, quit—**G** is not a theorem.
6. If the result is T, go back to step 1.

13.4.2 A Modus Ponens-Based Inference Engine

As was shown by the proofs for the tautologies in Figure 13-12, the above procedure works well for small problems. However, for larger problems it rapidly gets out of hand. Consider, for example, a wff involving just 30 propositional letters (not even enough to describe a simple 16-bit adder). A full truth table analysis would require over 1 billion interpretations (rows).

The use of inference rules in the inference engine drastically decreases this combinatorial explosion by handling with one symbolic operation many common parts of a truth table. For the propositional calculus, only one such inference rule is needed, namely, ***modus ponens.*** This rule states that if one is given as valid the pair of wffs x and $x \Rightarrow y$ (where x and y are themselves arbitrary wffs), then the wff y is also valid. No further analysis is needed, regardless of the complexities of x and y.

This definition agrees with our intuitive notion of an "if-then" rule. Combining it with the above axiom set gives a still simple inference engine, but one that is more efficient than the one above. However, its use does require that the wffs be written in forms using only "not(¬)" and "implies(⇒)."

Solving problems in a fashion similar to before, but using modus ponens, would result in an inference engine of the form:

1. (Decision Procedure) Pick two wffs from the axiom set, where one of these wffs has ⇒ as its outermost connective.
2. (Application of Inference Rule) Verify that the antecedent part of the axiom with the outermost ⇒ exactly matches the other axiom.
3. If no match occurs, go back to step 1.
4. If a match occurs, compare the consequent with **G.**
5. If the same, quit with the answer yes.
6. If different, add the consequent to the axiom set, and go back to step 1.

Eventually, this process will generate all wffs which can be deduced from the original set. If one of them matches, the goal is a theorem.

13.4.3 Other Inference Rules

Just as there are many sets of equally expressive connectives, there are other inference rules with the same logical power. Two such are ***modus tollens*** and ***resolution,*** and a myrid of variations. Modus tollens is like running modus ponens backward; namely, given a wff "$\neg y$" and another wff "$x \Rightarrow y$," we can infer the wff "$\neg x$." This is the same as stating $\neg y,(x \Rightarrow y) \vdash \neg x$.

The rationale for this inference rule can be developed by looking at the truth table description for " ⇒ " (Figure 13-10). If y is false (**B** in the figure), the only conclusion to which one can come that is consistent is that x is also is false (**A** in the figure). Again, modus tollens requires wffs expressed only with " ⇒ " and "¬."

Resolution is a little different, since it is normally used when individual wffs are expressed in ***conjunctive normal form,*** which is the "and(∧)" of terms built only from "not(¬)" and "or(∨)." However, simplistically it can be thought of as a "chaining rule," which takes two wffs of the form "$x \Rightarrow y$" and "$y \Rightarrow z$" and infers the wff "$x \Rightarrow z$." In their conjunctive form, these two wffs look like $(\neg x \vee y)$ and $(\neg y \vee z)$, with resolution simply "combining" the two wffs and "cancelling out" the y from one and the matching $\neg y$ from the other. Again this is $(x \Rightarrow y),(y \Rightarrow z) \vdash (x \Rightarrow z)$.

All these inference rules will be discussed in greater detail in the next two chapters, along with matching decision procedures to govern their use.

13.5 A SIMPLE PROBLEM

As an example of a problem expressed in a propositional calculus framework, consider the propositions labeled **A, B,**...and **G** from Figure 13-8 and an initial set of axioms with "interpretations" as follows:

1. **A ⇒ B**—If the car has gas, then I can go to the store.
2. **(B ∧ C) ⇒ D**—If I can go to the store and I have money, then I can buy food.

   Note that when we translate this wff into a form using only " ⇒ ," there are several possibilities, including:
   - **B ⇒ (C ⇒ D)**
   - or **C ⇒ (B ⇒ D)**
3. **(D ∧ (E∨F)) ⇒ G**—If I have food and either the sun is shining or I have an umbrella, then today I can go on a picnic.

   Again, there are several equivalent wffs involving only " ⇒ " and "¬," including **(D ⇒ ((¬E ⇒ F) ⇒ G))**.
4. **A**—The car has gas.
5. **C**—I have money.
6. **¬E**—The sun is not shining.
7. All the axiom schemas from Figure 13-12.

Stating that these wffs are axioms is equivalent to assuming that they always evaluate to true. Since they are not all tautologies in themselves, this is equivalent to "and"ing them all together into one giant wff, and constraining the interpretations which we use to solve problems using them to ones that make this single giant wff evaluate to true. Thus, for example, the fourth axiom **A** constrains all interpretations to ones where the propositional letter **A** must be assigned the value true. Likewise, the first axiom excludes any interpretation in which **A** is true and **B** is false. Note that modus ponens would deduce from these two examples that the wff **B** is also true, but this agrees with the combination of the above interpretations that constrains assignments to ones where both **A** and **B** are true.

In structure, the first three of these program wffs are axioms that define *if-then rules,* while the next three correspond to *facts.* The referenced axiom schemas carry over from the basic construction of the logic system itself. It is important to emphasize one more time that only this latter set represents inherent tautologies; the truth of the others is due totally to the wishes of the programmer of the system.

Examples of the kind of wff questions that one might ask are:

- **D**—Is it true that "I can buy food"?
- **F ⇒ G**—Is it true that "if I have an umbrella, then I can go on a picnic"?

Again, asking a question in this system is equivalent to identifying a wff and asking if it is in the set of true wffs deducible from the axioms and inference rules. It is up to the decision procedure to mechanically find the proof sequence, if it exists.

For propositional logic, the simplest possible proof procedure is to simply do a truth table analysis, i.e., look at all possible assignments of true/false to all the propositions used in the axiom set and goal and see if there is at least one interpretation (one row) that makes all the original axioms and the goal true at the same time. Figure 13-13 diagrams one such analysis. Of the 128 possible interpretations, there are only 4 that satisfy all the axioms. Further, in all 4 cases, **D** is also true, meaning that we have proved **D.**

Another possible decision procedure in this system involves beating these wffs (using an inference rule) against each other and any new wffs as long as new wffs are inferred. As each is inferred, it can be compared to the desired question. A match ends the computation. If no match is found, the procedure will eventually infer every possible true wff. While it is more difficult to formulate such a procedure, we mention it here because most other logic systems are not amenable to simple truth table-like analysis, and must resort to more complex procedures.

Interpretation							Axioms						Goal	Axioms
A	B	C	D	E	F	G	A⇒B	(B∧C)⇒D	(D∧(E∨F)) ⇒G	A	C	¬E	D	Satisfied?
F	—	—	—	—	—	—				F				No
—	—	F	—	—	—	—					F			No
—	—	—	—	T	—	—						F		No
—	—	—	F	—	—	—							F	No
			....						...					... No ...
T	F	T	T	F	F	F	F	T	T	T	T	T	T	No
T	F	T	T	F	F	T	F	T	T	T	T	T	T	No
T	F	T	T	F	T	F	T	T	F	T	T	T	T	No
T	F	T	T	F	T	T	T	T	T	T	T	T	T	Yes
T	T	T	T	F	F	F	T	T	T	T	T	T	T	Yes
T	T	T	T	F	F	T	T	T	T	T	T	T	T	Yes
T	T	T	T	F	T	F	T	T	F	T	T	T	T	No
T	T	T	T	F	T	T	T	T	T	T	T	T	T	Yes

— means either T or F

FIGURE 13-13

A proof procedure using truth tables.

As an example, a proof sequence for **D** using only modus ponens is straightforward, namely:

- From **A** and $(\mathbf{A} \Rightarrow \mathbf{B})$ infer **B**
- From **B** and $(\mathbf{B} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{D}))$ infer $(\mathbf{C} \Rightarrow \mathbf{D})$
- From **C** and $(\mathbf{C} \Rightarrow \mathbf{D})$ infer **D**

Figure 13-14 diagrams this sequence pictorially.

The general term for this kind of inference is ***forward chaining,*** or ***antecedent reasoning,*** since the system is expanding "forward" from known wffs to new ones.

Note that there are several other sequences that also prove **D,** and that one of the reasons this one went so simply is that we "chose" the right equivalent representation of wff tautology axiom 2. In general, modus ponens-based decision procedures will "hunt around" for the right form basically by trying many variations of the three axiom schemas. This is particularly apparent if one tries to prove the second question above, i.e., $(\mathbf{F} \Rightarrow \mathbf{G})$. Although it is "obvious" once **D** is proved that this also follows, the amount of work involved without using some "auxiliary" inference rules (such as from x and $(x \vee y) \Rightarrow z$ infer $y \Rightarrow z$) is astonishing. This is one of the reasons why logic-based computing did not really become practical until other sets of inference rules (particularly resolution) were well understood.

There is an alternative decision procedure that involves picking a wff whose truth value is desired and "working backward." At each step one tries to find a wff which, if one used modus ponens between it and some other wff, would infer the desired wff. This "other wff" then becomes the new question, and the process is repeated. Needless to say, this is ***backward chaining*** or ***consequent reasoning.*** However, it is most frequently used with inference rules other than modus ponens. Again, we will discuss this in much more detail in the next two chapters.

Finally, the limitations of this system can be seen by looking at the kinds of questions that one might like to ask but for which there are simply no expressible wffs reflecting them. For example, "Do there exist any conditions under which I can go on a picnic?" Other logics, particularly the first-order predicate calculus, do permit such questions.

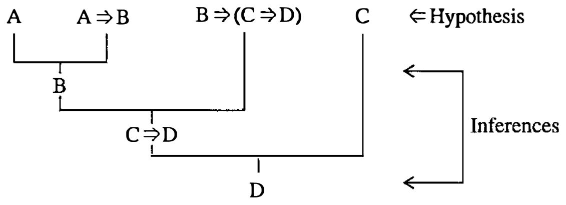


FIGURE 13-14
Sample proof sequence.

13.6 PROBLEMS

1. Rewrite Figure 13-3 in prefix notation.
2. Invent an appropriate set of relations, and express Figure 13-2 using prefix notation.
3. Augment Figure 13-3 to include statement(s) defining the relation **is-uncle-of.** What would be returned if the question was:
   a. Is **is-uncle-of**(x, Tim) true?
   b. Is **is-uncle-of**(Peter, x) true?
4. For Figure 13-4, show informally a chain of inferences which lead to the conclusion that **fac**(3)=6 is a theorem.
5. How might an inference engine answer the question, "Why is Martha the grandparent of Peter?"
6. Show via a truth table analysis that all three axiom schemas of Figure 13-12 given for the propositional calculus system are tautologies.
7. For the picnic problem of Section 13.5, prove $\mathbf{F} \Rightarrow \mathbf{G}$ using only modus ponens and axiom schemas of the form $((a \wedge b) \Rightarrow (a \Rightarrow (b \Rightarrow c)))$ and $((a \vee b) \Rightarrow c) \Rightarrow (\neg a \Rightarrow (b \Rightarrow c))$.
8. Show using truth table analysis that modus ponens is both sound and complete.
9. Express Figure 13-14 as a proof sequence.

CHAPTER 14

PREDICATE LOGIC AND THE FIRST INFERENCE ENGINE

As discussed in Chapter 13, one of the problems with propositional calculus is its inability to make "logical generalizations" about objects. Consider, for example, the statement, "If something is human, it has a mother." To express this in propositional calculus would require separate axioms for each "something":

- If Mary is human, she has a mother.
- If Mike is human, he has a mother.
- If Tim is human, he has a mother.
- . . .

where "Mary is human" is one propositional letter, "Mary has a mother" is another, and so on.

What we really want to say is "For any *x*, if *x* is human, then *x* has a mother." ***Predicate-based logics*** allow us to do this by replacing the statement letters of propositional logic by ***predicate expressions*** that test whether or not certain combinations of objects, or tuples of objects, are members of certain relations. The objects being tested are listed as arguments to the name of the relation.

The simplest, and most used, such system is the ***first-order predicate calculus***. Besides predicates, this calculus introduces ***constants, functions, variables,*** and ***quantifiers,*** all of which deal with the identification and manipulation of objects that make up arguments for predicates.

Constants identify specific objects, such as "Tim" or "3." Functions indicate how to map one or more objects specified as arguments into other objects. Variables may take on values from the domain of objects expressible from the functions and constants, and may be used freely any place where constants or function expressions are permitted. All three of these may be combined into ***logic expressions*** that describe relations, similar to those discussed for function-based computing.

Finally, a quantifier takes a variable and a logical expression containing copies of that variable, and specifies how much of the relevant domain of objects needs to be substituted for that variable to make the complete expression true. Just as **λ** did for functions, these quantifiers ***bind*** identifiers found in some expression, and specify where values for those identifiers should be substituted. Unlike lambda expressions, however, the purpose of a quantifier is to indicate how many of the domain values, when substituted for the identifier, must make the bound expression true before the whole quantified expression is considered true. The most common cases of this are "all objects" and "at least one object value." The former means that the bound expression must reduce to true regardless of what value is substituted. The latter requires only one such value, with no concern as to which it is.

Together, these additions make it possible to express directly statements such as "For all *x*, if *x* is human, then *x* has a mother." Of course, this complicates the domain of possible interpretations, but there still exists a fairly mechanical process to generate this domain. Also becoming somewhat more complex are the details of possible inference rules, although their general operation still follows equivalent inference rules from the propositional model. The following sections discuss these issues by starting first with the syntax of the first-order predicate calculus, then with ways of converting expressions into a standard form, and finally with ways of mechanically generating something akin to all possible interpretations. The latter will directly drive the first real inference engine for predicate logic, the Herbrand algorithm.

Very complete and mathematical discussions of such topics can be found in references such as Chang and Lee (1973) or any good book on logic, such as Mendelson (1964, 1979).

As a side note, the logics we address here are called first-order because they permit identifiers to occur only when "first-order objects" such as constants might be expected. This is as arguments to functions, predicates, or relations. Other logics more advanced than first-order exist in mathematics. These logic systems permit such things as an uncountable number of predicates, function symbols, variables, axioms, and more sophisticated quantifiers, or even permit variables to take the place of things other than simple objects, such as functions or predicates.

14.1 BASIC SYNTAX

As with proposition-based logics, there are a variety of models for first-order predicate calculus. Perhaps the most natural one uses a syntax which looks very much like a combination of a prefix-notation functional language and the logical connectives used in the propositional notation, coupled with the introduction of two quantifiers for scoping identifiers. Figure 14-1 diagrams one such syntax where, as with lambda calculus, considerable freedom will be taken in such respects as parentheses, multiple identifier names in quantifier positions, and use of infix notation for common operations.

14.1.1 Basic Components

The "meaning" of much of this notation is straightforward. There is a subsyntax (***term***) which describes how to build expressions describing objects. Within a term, a ***constant*** is a character string which represents specific objects such as "3," "apple," etc. An ***identifier*** is much like its equivalent in lambda calculus; it represents a ***placeholder*** for some object value. A ***functor*** is the name of some function which takes some set of arguments (each representing some object) and produces some object as a result. Unlike lambda calculus, however, there is no mechanism here for describing how this result object is actually built from the input terms.

```
<constant> := <symbol> | <number>

<functor> := <symbol>

<predicate-name> := <symbol>

<identifier> := <symbol>

<tuple> := (<term> {,<term>}*)

<term> := <constant> | <identifier> | <functor><tuple>

<atom> := <predicate-name><tuple>

<literal> := <atom> | ¬<atom>

<binary-connective> := ∧ | ∨ | ≡ | ⇒ | ⊗

<quantifier> := ∀ | ∃

<wff> := <literal>
        | ¬<wff>
        | (<wff><binary-connective><wff>)
        | (<quantifier><identifier>"|"<wff>)
```

FIGURE 14-1
A typical first-order syntax.

As before, liberties will be taken with the syntax of terms where the intent is obvious, such as in the infix use of standard arithmetic operators. Also, we have not made any distinction among the various syntatic terms that are symbols; with the exception of constants and identifiers, context will be sufficient to distinguish them. Real languages give some mechanism for distinguishing between these latter two (constants and symbols), such as leading upper- or lowercase letters, quotation marks, etc. Again, we will let context and common practice distinguish the two: x is a variable; Tim is a constant.

14.1.2 Atoms, Literals, and Predicates

Another subsyntax describes an ***atom*** (short for ***atomic symbol***), which consists of the name of a relation prefixed to a tuple whose elements are terms. The meaning of this is that under an interpretation the atom evaluates to true if the object specified by the tuple is in the set defined by the relation, and is false otherwise. Note that this is a different definition of ***atom*** than that used previously for an s-expression, which does not involve any subcomponents from a **cons** operation.

A ***literal*** is an atom with an optional ¬ prefixed. With a "¬" applied, an atom which evaluates to T causes the literal to evaluate to F, and vice versa.

Note that while atoms and terms have virtually identical syntax, they are used in different places. Terms represent objects (which might include the values T and F) and can only appear as arguments to functions or atoms. Atoms represent the most basic wff construction and may never appear as arguments to other predicates or functions, except, in a sense, as arguments for the logical connectives.

Note also that atoms can be interpreted as boolean tests, with the predicate name reflecting the type of test to be applied to the tuple. Perhaps the most useful such predicate is ***equality,*** which takes two objects and returns true only if the objects are identical. These objects can, of course, be the results of functions applied to other objects, as in "**height**(Pete)=6×30.48cm." Because of the frequency of such equality test, a very common simplification of this kind of an expression is to make the major functor used in the test (e.g., **height** in the example) into a predicate that takes one more argument than the function did, gives this argument the value of the other side of the "=," and drops the "=" as a predicate. The interpretation is that the new form is true if and only if the original function, when applied to all but the last argument, returns a values equaling that in the last argument position. Thus "**has-height**(x,y)" would be true only if the object x is y units high (as in **has-height** (Pete, 6×30.48cm.)). This notation actually matches the original definition of a function as a relation (Chapter 1), and when used in many real logic languages actually permits the programmer to define a function by defining the relation it represents.

14.1.3 Wffs and Quantifiers

A *wff* is a logical expression built up from atoms combined by the standard logical connectives from propositional calculus plus quantifier symbols to scope identifier bindings. The two quantifiers available in first-order logic scope the domain of values that identifiers take on within a wff. As with the lambda symbol, they are followed by a ***binding variable*** and a wff making up the quantifier's "body." Just as in lambda calculus, all ***free occurrences*** of the binding variable in the body are bound to the matching quantifier and always represent the same object, whatever it is. Also as with lambda calculus, there are many different syntatic notations in the literature for specifying the binding variables, such as surrounding a symbol by parentheses (i.e., $\forall(x)$—), following the variable by a "." (e.g., $\forall x.$—), and so on. The notation used here (namely, "$\forall x|$—") was chosen simply to match that chosen earlier for lambda expressions, set descriptions, and the like. As before, simplifications will be used where the intent is obvious, such as in cascading a string of the same quantifiers into one, with a string of identifier names following it. Thus we will feel free to write an expression like "$(\forall x|(\forall y|(\forall z|(\exists q|(\exists w|\ldots)))))$" in the simpler form "$\forall xyz|\exists qw\ldots.$"

The first of the two quantifiers is the ***universal quantifier*** $\forall$ (pronounced "for all"). This defines a wff that is true only if the sub-wff making up its body remains true regardless of what values are given to the binding variable and substituted into it. ***Substitution*** here means exactly what it did for lambda calculus.

The ***existential quantifier*** $\exists$ (pronounced "there exists") is similar except that it forms a true wff if there exists as little as one object (value not specified) which, if substituted for the free instances of the binding variable in the body, make it true.

As an example, the statement from the beginning of this section ("For any *x*, if *x* is human, then *x* has a mother") can be written very precisely as:

$$(\forall x|\mathbf{is\text{-}human}(x) \Rightarrow (\exists y|\mathbf{is\text{-}mother\text{-}of}(y,x) \wedge \mathbf{is\text{-}human}(y)))$$

If we assume that this wff is true, then for any object *x*, if *x* has the property **is-human,** then there is at least one other object *y* such that *y* **is-mother-of** *x*, and *y* is itself human. This object *y* is not identified other than that it exists.

14.2 BASIC INTERPRETATIONS

Again, an ***interpretation*** is a description of what a wff "means." It is the ***semantics*** of the wff. Unlike functional languages, however, such interpretations are much more important to the programmer of a logic-based program than they are to the program, mainly because most logic systems are concerned with the "form" of a logical argument and not their content.

For our purposes, we define an interpretation as having two parts. First is the permanent part, which is constant across all wffs in a language. This includes primarily the definition of the logical connectives and the quantifiers. Thus, "$\forall x|$" means the same thing regardless of the wff it is embedded in, as does a connective such as "$\wedge$."

Second are a programmer-defined mapping of values to the various other symbols that make up a wff. Such interpretations may vary from wff to wff, or even sometimes among different parts of a large wff. This includes:

- Constant symbols are assigned some specific actual "values."
- Function symbols are assigned some mappings, and terms computed as function applications from those mappings and the constant interpretations.
- Predicate symbols are assigned relations, with predicates given true/false values on the basis of whether or not their arguments form valid tuples in the relation.

Now, given an interpretation, we can ***evaluate*** an unquantified wff by the following:

- Compute terms as the appropriate function applications.
- See if the tuples formed by the evaluated terms are in the associated relations.
- If so, mark those atoms as true.
- If not, mark the atoms as false.
- Combinations of atoms joined via the logical connectives are interpreted exactly as in propositional calculus.

A universal quantifier in front of a wff causes the above procedure to be repeated for every possible object substituted for all free occurrences of the binding variable in the wff body. If any of these evaluations yields false, the wff is marked false. If all evaluations yield true, the whole wff yields true. Multiple binding variables, as in $\forall xyz$, require all possible combinations of substitutions to be true.

What this means is that if we have a wff such as $(\forall x|\mathbf{P}(x))$, it is totally equivalent to multiple copies of the body **and**ed together, one per possible domain value. For example, if the domain of *x* is the set {**red, white, blue**}, then

$$(\forall x|\mathbf{P}(x)) \equiv \mathbf{P}(\mathbf{red}) \wedge \mathbf{P}(\mathbf{white}) \wedge \mathbf{P}(\mathbf{blue})$$

The advantage of the $\forall$ quantifier is that it presents a more concise notation, especially when the domain of some variable is an infinite set, as it usually is.

An existential quantifier is similar to the above except that if any of the evaluations yield true, the result is true. Only if all evaluations are false is the whole wff false. This is equivalent to taking the body of the expression, making a copy for each possible value that the variable might take on, and **or**ing the results together. Thus, for the domain {**red, white, blue**}:

$$(\exists x|\mathbf{P}(x)) \equiv \mathbf{P(red)}\vee\mathbf{P(white)}\vee\mathbf{P(blue)}$$

Note that a wff containing a variable not bound to any quantifier (a ***free variable***) is not usually evaluatable.

14.3 STANDARD EXPRESSION FORMS

(Mendelson, 1964, 1979, chaps. 1, 2; Chang and Lee, 1973, chaps. 2, 3)

When humans describe wffs in formal logic systems, they usually prefer to use a wide mix of logical connectives in the construction of the wffs. Although there is certainly nothing wrong with this technique, researchers have found that the most efficient inference rules and decision procedures seem to be applicable when the number of different connectives is limited to a basic few. This section describes a series of standard forms for wffs that seem particularly attractive, and that have found widespread use in real logic-based computing systems. For the most part these standard forms are only syntatically different from the richer forms used by humans, and they have the same expressive power. In these cases we will describe the general translation steps needed to bridge the gap. As always, we leave to the references any proofs of the correctness of these translations.

Figure 14-2 summarizes the most common of these forms. In general, these forms are equally applicable to both propositional and first-order predicate calculus models. Of course, in propositional logic there are no quantifiers, so there is no specific prenex form equivalent, and there need be no quantifiers in any of the special forms.

The ***prenex*** form is one where all the quantifiers are at the beginning of the wff, leaving the body a quantifier-free expression. The other forms then deal with variations of the body.

The disjunctive normal and conjunctive normal forms often go by the more common names ***sum of products*** and ***product of sums,*** respectively, and describe whether the body of the wff is a logical "and" of "or" literals, or vice versa.

The ***clausal form*** is a special case of the conjunctive form. Its body is a conjunction ("and") of ***clause***s, each of which is itself a disjunction ("or") of ***literal***s that in turn are either individual atoms or the negation ("not") of individual atoms. Again, it is usable in both propositional and predicate logic-based systems, with the addition that in predicate logic systems there is a set of universal quantifiers ("∀") surrounding the disjunction, one for each distinct free variable in any predicate argument. There are no existential quantifiers left in this representation.

Atom: A propositional letter or a single predicate symbol with arguments.

Literal: An atom or a negated atom.

Prenex Form: $Q_1x_1|(Q_2x_2|\ldots(Q_nx_n|M))$ where Q_k is a quantifier, and M is a wff without quantifiers.

Conjunctive Form: $Q_1x_1|(Q_2x_2|\ldots(Q_nx_n|(A_1\wedge A_2\wedge\ldots A_n))$ where A_k does not use $\wedge$.

Disjunctive Form: $Q_1x_1|(Q_2x_2|\ldots(Q_nx_n|(A_1\vee A_2\vee\ldots A_n))$ where A_k does not use $\vee$.

Conjunctive Normal Form: $Q_1x_1|(Q_2x_2|\ldots(Q_nx_n|(C_1\wedge C_2\wedge\ldots C_n))$ where $C_k = (L_1\vee L_2\vee\ldots L_n)$ is a disjunction of literals L_i.

Disjunctive Normal Form: $Q_1x_1|(Q_2x_2|\ldots(Q_nx_n|(D_1\vee D_2\vee\ldots D_n))$ where $D_k = (L_1\wedge L_2\wedge\ldots\wedge L_n)$ is a conjunction of literals L_i.

Clausal Form: $\forall x_1|(\forall x_2|\ldots(\forall x_n|(C_1\wedge C_2\wedge\ldots C_n))$ where $C_k = (L_1\vee L_2\vee\ldots L_n)$ is a disjunction of literals L_i.

Notes:
- Q_k is a quantifier ∀ or ∃
- A_k is an arbitrary wff,
- L_k is a single literal

FIGURE 14-2
Some standard logic forms for wffs.

The rigid structure of the clausal form will shortly form the basis of the most common set of inference rules and decision procedures. Because of this importance, Figure 14-3 gives a complete BNF syntax for wffs in clausal form. Note the obvious definitions of a ***positive literal, negative literal,*** and ***unit clause.*** Obvious extensions define a ***positive clause*** as one containing only positive literals, a ***negative clause*** as one containing

```
<atom> := <predicate-symbol>(<term> | {,<term>}*)

<positive-literal> := <atom>

<negative-literal> := ¬<atom>

<literal> := <positive-literal> | <negative-literal>

<unit> := <literal>

<clause> := <unit> | <literal>{∨<literal>}+

<disjunctive-form> := (<clause>){∧(<clause>)}*

<clausal-form> := {∀<variable>"|"}*(<disjunctive-form>)
```

FIGURE 14-3
Syntax of clausal form for predicate logic.

only negative literals, and a ***nonunit clause*** as one with multiple literals. Any clause with both positive and negative literals is a ***mixed clause.***

As before, where the intent is unambiguous, we will feel free to delete parentheses and otherwise simplify examples.

14.3.1 Form Conversions in Propositional Calculus

The rules for converting a propositional calculus wff in any of the standard formats (or with an arbitrary mix of connectives) into either of the normal forms is direct. One simply sequentially replaces subexpressions involving various connectives with equivalent subexpressions with some possible simplifications as required. Figure 14-4 lists these conversion rules and the relative order in which they should be applied. Note that it may be necessary to repeat the steps (in sequence) several times to work one's way through the layers of parentheses. In addition, the standard simplifications such as "$\neg \mathbf{A} \wedge \mathbf{A} \rightarrow \mathrm{F}$" can be applied at any time. Figure 14-5 gives an example of this conversion process in action.

14.3.2 Prenex Normal Form

(Mendelson, 1979, pp. 88–92)

The existence of quantifiers in predicate logic systems slightly complicates the conversion of arbitrary predicate logic wffs into normal forms, particularly clausal form. Besides the rearrangement of the simpler logical connectives such as $\Rightarrow$ and $\vee$, any existential quantifier ("$\exists$") must be removed or converted to a universal one ("$\forall$"), and all universal quantifiers moved up front of the rest of the wff. The approach for first-order predicate logic involves an initial conversion of the wff into what is called ***prenex normal form:***

$$Q_1x_1|(Q_2x_2|\ldots(Q_nx_n|\mathbf{M}))$$

where Q_k is a quantifier of either type, and **M** is a wff without quantifiers. Note that this form has all the quantifiers in a line up front, with all other logical connectives in a quantifier-free subexpression **M.** There are no constraints at this time on the type or order of connectives within **M.**

Step	replace	by (a, b, c arbitrary wffs)
1.	$a \equiv b$	$(a \Rightarrow b) \wedge (b \Rightarrow a)$
2.	$a \Rightarrow b$	$\neg a \vee b$
3.	$\neg(\neg a)$	a
4.	$\neg(a \vee b)$	$\neg a \wedge \neg b$
5.	$\neg(a \wedge b)$	$\neg a \vee \neg b$
6a	$a \vee (b \wedge c)$	$(a \vee b) \wedge (a \vee c)$—for conjunctive form
6b	$a \wedge (b \vee c)$	$(a \wedge b) \vee (a \wedge c)$—for disjunctive form

Also remember associativity and commutativity of $\wedge$ and $\vee$:

$(a \wedge (b \wedge c)) \rightarrow ((a \wedge b) \wedge c) \rightarrow (a \wedge b \wedge c)$

$(a \wedge b) \rightarrow (b \wedge a)$

$(a \vee (b \vee c)) \rightarrow ((a \vee b) \vee c) \rightarrow (a \vee b \vee c)$

$(a \vee b) \rightarrow (b \vee a)$

Standard simplifications:

$a \wedge T \rightarrow a$

$a \wedge F \rightarrow F$

$a \wedge \neg a \rightarrow F$

$a \vee T \rightarrow T$

$a \vee F \rightarrow a$

$a \vee \neg a \rightarrow T$

FIGURE 14-4

Conversion steps for propositional logic.

Sample wff: $((D \wedge (E \vee F)) \Rightarrow G)$

1. $\rightarrow \neg(D \wedge (E \vee F)) \vee G$—Rule 2
2. $\rightarrow (\neg D \vee \neg(E \vee F))) \vee G$—Rule 5
3. For disjunctive normal form: $\rightarrow \neg D \vee (\neg E \wedge \neg F) \vee G$—Rule 4
4. For conjunctive normal form:
   a. $\rightarrow ((\neg D \vee \neg E) \wedge (\neg D \vee \neg F)) \vee G$—Rule 6a
   b. $\rightarrow (\neg D \vee \neg E \vee G) \wedge (\neg D \vee \neg F \vee G)$—Rule 6a

FIGURE 14-5

Sample conversion to normal forms.

After conversion of a wff into this form, conversion can proceed to any of the other desired standard forms. For example, the standard rules from Figure 14-4 can be used to convert the quantifier-free wff **M** into other forms such as conjunctive normal or disjunctive normal. Later sections will discuss what is involved to finish up the process and remove existential quantifiers from the Q list to yield clausal form.

Figure 14-6 gives a set of rules for converting arbitrary wffs into prenex form. As before, this set of rules might have to be applied repeatedly if quantifiers are nested deeply inside a wff. Also note that, as with the substitution rules for lambda calculus, there are cases where it may be necessary to ***rename*** a binding variable in a quantified expression to prevent possible conflicts when two separately quantified wffs are combined. As before, the notation "$[\mathbf{A}/x]\mathbf{M}$" means to replace all occurrences of the variable x in the wff **M** by the expression **A.**

While it is possible to formally prove the correctness of these transformations, most of them should be obvious by "common sense," including the cases where renaming is needed. The only ones that do deserve special comment are special cases 5a and 6a, where we are dealing with the combination of two wffs using the same leading quantifier with the "same name" for their binding variable. In both cases, the domain of

Assume x an arbitrary variable
z a unique variable name appearing nowhere in wff
M,N arbitrary wffs.
Q an arbitrary quantifier ($\forall$ or $\exists$)

Steps: (repeat as necessary)

1. $\neg(\forall x|M) \rightarrow \exists x|\neg M$

2. $\neg(\exists x|M) \rightarrow \forall x|\neg M$

3.* $(Qx|M)\vee N \rightarrow Qx|(M\vee N)$ N has no free x's
$\rightarrow Qz|([z/x]M\vee N)$ otherwise

4.* $(Qx|M)\wedge N \rightarrow Qx|(M\wedge N)$ N has no free x's
$\rightarrow Qz|([z/x]M\wedge N)$ otherwise

5.* $(Q_1x|M)\wedge(Q_2x|N) \rightarrow Q_1x|Q_2z|(M \wedge [z/x]N)$

5a.* $(\forall x|M)\wedge(\forall y|N) \rightarrow \forall x(M\wedge([x/y]N))$ if no free x's in N

6.* $(Q_1x|M)\vee(Q_2x|N) \rightarrow Q_1x|Q_2z|(M \vee [z/x]N)$

6a.* $(\exists x|M)\vee(\exists x|N) \rightarrow \exists x|(M\vee N)$ if no free x's in N

* if N has free x's, then rename all x's in M to some unique new name z first.

FIGURE 14-6
Conversion steps for prenex form.

the binding variable is the same (for example, the Herbrand universe to be discussed in Section 14.5.1). For case 5a, if each wff body is true for whatever value is substituted, it is certainly still true if that value happens to be the same as that substituted in the other wff. For case 6a, if we find a value that makes one of the wffs true, then the whole wff is true, and we need not worry about what happens to the other half for that value.

Figure 14-7 diagrams a sample conversion using these rules.

14.3.3 Skolem Standard Form

The next step in the conversion process is elimination of all the existential quantifiers in a wff in prenex normal form. The result is a ***Skolem standard form*** wff:

$$\forall x_1\forall x_2\ldots\forall x_n|\mathbf{M}=\forall x_1x_2\ldots x_n|\mathbf{M}$$

Example: $\forall x|(\text{human}(x) \Rightarrow \exists y|\text{is-mother-of}(y,x))$

$\rightarrow \forall x|(\neg\text{human}(x) \vee \exists y|\text{is-mother-of}(y,x))$

$\rightarrow \forall x|\exists y|(\neg\text{human}(x)\vee\text{is-mother-of}(y,x))$

FIGURE 14-7
Sample conversion into prenex normal form.

The rules for conversion are a combination of prior procedures and several rather different new ones. Clearly we must start with a conversion of a wff into prenex normal form. This gets all the quantifiers up front to the left. Next, however, removal of the existential quantifiers (if any) requires invention of what are called ***Skolem constants*** and ***Skolem functions***.

Consider first the meaning of a wff in prenex normal form where the leftmost quantifier is "$\exists$." Assume for argument's sake that the binding variable for this quantifier is "x." The fact that the quantifier is leftmost means that if the wff is to be true, there must exist at least one object from the quantifier's domain that will allow the rest of the wff to become true when that value is substituted for x throughout the wff. Note also that this object's value is "independent" of any values given to any other variables in the wff by any other quantifiers to the right of this one in the wff. Consequently, if we knew the value of this magic object, we could substitute it in the wff and delete the leading quantifier. ***Skolemization*** of a wff does just this: It removes a leading "$\exists x$" and replaces all free occurrences of x in the rest of the wff with a new constant symbol, which is not used anywhere else but is used solely in the wff to represent this magic value.

This new constant symbol is a ***Skolem constant,*** and its purpose is to "name" the value which satisfies the wff. Instead of saying "There exists an x such that...," we are saying that the object exists with some unambiguous name. Remember from our discussions of interpretations, however, that "naming" an object by writing a constant value does not inherently specify any particular value—that is up to the interpretation we give it. All the "name" does is signify that an object with appropriate values exists—exactly the same meaning as what we gave to the existential quantifier in the first place.

This process of generating unique Skolem constants is repeated as long as the leftmost quantifier of the converted wff is still existential, but with a new Skolem constant each time.

The more complex case occurs when there is an existential quantifier in the wff, but it is not the first. Consider the wff

$$\forall x_1\ldots\forall x_{k-1}\exists x_k\ldots\mathbf{M}$$

The meaning of this is that regardless of what values we pick for x_1 through x_{k-1}, there is still some value we can give to x_k to make the rest of the wff true. However, because of the nesting of quantifiers, this value for x_k may vary as the values for x_1 through x_{k-1} vary. In other words, x_k is *a function of* x_1 through x_{k-1}. To remove the existential quantifier we will do as before: We will "create" a special symbol to represent the value for x_k. However, this cannot be a constant. It must be a ***Skolem function*** of k−1 arguments. Assuming that the function name "g_k" is used nowhere else, we can delete "$\exists x_k$," by replacing all free x_k's in the rest of the wff by "$g_k(x_1,\ldots,x_{k-1})$."

The meaning of this Skolem function again mirrors that of a Skolem constant. The system has no fixed interpretation of what its mapping is, but we know that it exists. The value that it returns is the same value that we would have given x_k to begin with.

As before, we repeat the process as often as required. Note that each Skolem function must be distinct from all others, even though it is possible that they may have the same argument list.

As an example, consider Figure 14-8. We read the original wff as "For any object x there exists some y, such that either x is not a human (note that we do not care about y here) or the pair y and x satisfy the relation "**is-mother-of.**" Skolemizing this wff replaces the "∃" and the y by a function "**mother-of**" which, given an x, returns its mother. Note that the value of this function is irrelevant when the value given to x does not have the property "**human.**" Reading this new wff results in "For any object x, either x is not a human or the object computed by the expression "**mother-of(*x*)**" satisfies the relation "**is-mother-of**" with respect to x." Clearly the veracity of the two wffs tracks exactly.

Notice also that this transformation does not depend on the interpretation given it. Even though it makes perfect sense to humans as it stands, we could invent valid interpretations in which the predicates deal with numbers, coaches and football teams, etc. Only the interpretation of "**mother-of**" changes to match.

14.3.4 Clausal Form

Pure ***clausal form*** for predicate calculus occurs when a wff is in the Skolem standard form, and the quantifier-free sub-wff on the right is in conjunctive normal form, i.e.,

$$\forall x_1|\forall x_2\ldots|\forall x_n|(\mathbf{C}_1 \wedge \mathbf{C}_2 \wedge \ldots \wedge \mathbf{C}_m)$$

Here each of the $\mathbf{C}_k$'s is a quantifier-free disjunction ("or") of literals, with no other quantifiers or connectives used anywhere. Each $\mathbf{C}_k$ is a ***clause,*** with the order of clauses within the conjunction and the order of literals within a clause totally immaterial.

To get a better understanding of what this form means, consider any one clause $\mathbf{C}_k$. Assume that all the positive literals are of the form $\mathbf{p}_m$ and that the negative literals are of the form $\neg\,\mathbf{q}_n$, where the **p**'s and **q**'s are atoms (predicates). Regrouping all the negated atoms on the left gives:

$$\mathbf{C}_k=(\neg\,\mathbf{q}_1\vee\ldots\vee\neg\,\mathbf{q}_N\vee\mathbf{p}_1\vee\ldots\vee\mathbf{p}_M)$$

Temporarily factoring out the "¬" leaves

$$\mathbf{C}_k=(\neg\,(\mathbf{q}_1 \wedge \ldots \wedge \mathbf{q}_N)\vee(\mathbf{p}_1\vee\ldots\vee\mathbf{p}_M))$$

From the definition of " ⇒ " this is equivalent to

$$\mathbf{C}_k=((\mathbf{q}_1 \wedge \ldots \wedge \mathbf{q}_N) \Rightarrow (\mathbf{p}_1\vee\ldots\vee\mathbf{p}_M))$$

Thus if the whole wff is true, each $\mathbf{C}_k$ must be true, and each of these that have both positive and negative atoms may in turn be considered to be rules of the form: "If all $\mathbf{q}_n$'s are true, then so is at least one $\mathbf{p}_m$."

For obvious reasons, many logic programming systems call such a clause an ***if-then rule,*** with the **q**'s representing the ***antecedent*** or ***conditions,*** and the **p**'s representing the ***consequent, conclusion,*** or ***result.***

Example: $\forall x|\exists y|(\neg\text{human}(x)\vee\text{is-mother-of}(y,x))$

Choice of y depends on value of x

Define Skolem function mother-of:D→D
wff = $\forall x|(\neg\text{human}(x)\vee(\text{is-mother-of}(\text{mother-of}(x),x))$

FIGURE 14-8
Sample Skolem functions.

14.4 HORN CLAUSES

A ***Horn clause*** is a clause **C** with the property that at most one atom in the clause is not negated (i.e., is positive).

With this constraint there are exactly three forms a clause can take:

1. Exactly one unnegated atom and one or more negated ones: $\mathbf{C}=(\neg\,\mathbf{q}_1\vee\ldots\vee\neg\,\mathbf{q}_N\vee\mathbf{p}) \equiv ((\mathbf{q}_1 \wedge \ldots \wedge \mathbf{q}_N) \Rightarrow \mathbf{p})$. In our prior terminology, this matches an if-then rule with exactly one positive literal in the consequent and all positive literals in the antecedent. The rule reads "if all of the antecedent predicates $\mathbf{q}_1$ through $\mathbf{q}_N$ are true, then so is **p**.
2. Exactly one unnegated atom and no negated ones: **C=p.** This is often called an ***assertion*** or ***fact,*** since it must be true by itself in order for the whole wff to be true.
3. No unnegated atoms and one or more negated atoms: $\mathbf{C}=(\neg\,\mathbf{q}_1\vee\ldots\vee\neg\,\mathbf{q}_N)$. For reasons to be discussed later, this is often called a ***question*** or ***goal clause.***

Now consider what happens if we build a logic system where all wffs must be expressed in ways that permit conversion to a pure Horn clause form. It is clear that not all wffs can be expressed in this form [e.g., $\mathbf{A} \Rightarrow (\mathbf{B}\vee\mathbf{C})$]. Even so, this kind of a restriction forms the basis for many real logic programming systems, such as ***PROLOG.*** There are several reasons for this, including:

- Constraints can be placed on the wff syntax (as seen by the programmer) that guarantee that each wff entered into the system converts into exactly one clause.
- Reasonably complete and efficient decision procedures exist for axiom sets in this form.
- For many practical problems, the limitations of one predicate in the conclusion is not severe, and can be worked around.
- The three distinct forms (rule, fact, goal) seem to represent a clean match to normal human thinking and problem-solving activities.

Figure 14-9 gives an example of a Horn clause. Note in particular the convenient way in which an existential quantifier inside an implication becomes a universal quantifier over the whole wff.

14.4.1 The Empty Clause

As defined above, a clause is the conjunction of literals. To this point, we have assumed that there is always at least one such literal in any clause. However, this is not necessary: It is permissible for a clause to have no literals in it. Such a clause is termed an ***empty clause*** (written "ø"), and it has a rather special property. Basically, there is no possible interpretation that can make it true. The clause is unsatisfiable and always false, and makes any wff which includes it in a conjunction always false.

The value of this clause is that if it is ever inferred from other clauses, it is a positive indication that the original set of axioms which drove the inference process are somehow contradictory and cannot all be true simultaneously. As such, it forms the basis for many of the practical decision procedures to be discussed later.

14.4.2 Combining Multiple Wffs

The previous transformations describe conversion of a single wff into a form in which the only quantifiers are universal, they are all to the left, and the body is a conjunction of one or more clauses. Now consider some step in a logical computation in which we have a set of wffs, all in this form and all assumed to be true. While it is possible to deal with the individual wffs as such, it is often advantageous to combine them, strip out all quantifiers, and create simply a giant conjunction of clauses. Many of the more efficient inference rules work on clauses anyway, and the ability to pick and choose needed pieces without regard for which wff they originally came from is a distinct advantage.

Conventional Form: $\forall xy|(\exists z|\mathrm{is\text{-}parent\text{-}of}(x,z)\wedge \mathrm{is\text{-}parent\text{-}of}(z,y))$
$\Rightarrow \mathrm{is\text{-}grandparent\text{-}of}(x,y))$

Horn Form: $\forall xyz|(\neg \mathrm{is\text{-}parent\text{-}of}(x,z)$
$\vee \neg \mathrm{is\text{-}parent\text{-}of}(z,y)$
$\vee \mathrm{is\text{-}grandparent\text{-}of}(x,y))$

Alternatively: $\forall xyz|(\mathrm{is\text{-}parent\text{-}of}(x,z)\wedge \mathrm{is\text{-}parent\text{-}of}(z,y))$
$\Rightarrow \mathrm{is\text{-}grandparent\text{-}of}(x,y))$

FIGURE 14-9
Some sample Horn clauses.

The formal reason why we can perform this combination is simple. Logically, if we have a set of wffs, all true, then we can "and" them together without loss of validity. Further, rules 4 and 5a in the Prenex conversion process (Figure 14-4) then permit us to move all the universal quantifiers to the left of the wff bodies. This "and" of what was already a conjunction of clauses creates the giant conjunction we were after to begin with. Figure 14-10 gives an example of this.

Once in this form, the commutativity and associativity of the **and** connective permits the decision procedure to reorder the individual clauses any way it wishes.

One point that needs to be discussed is the potential requirement to ***rename*** variables in individual wff bodies when the quantifiers are moved up front. Clearly the decision procedure must be aware of the name dis-

Original axioms:

- $(\forall xy|(f(x,y) \vee m(x,y)) \Rightarrow p(x,y)$
- $\forall xy|(\exists z|p(x,z) \wedge p(z,y)) \Rightarrow gp(x,y)$
- $\forall xy|(\exists z|(p(x,y) \vee (p(x,z)\wedge a(z,y))) \Rightarrow a(x,y)$

Note: This splits into two Horn clauses:
—$\forall xy|\exists z|(p(x,z) \wedge a(z,y)) \Rightarrow a(x,y)$
—$\forall xy|p(x,y) \Rightarrow a(x,y)$

Combined single wff form: $\forall x_1x_2x_3x_4x_5y_1y_2y_3y_4y_5z_1z_2|$
$((f(x_1,y_1)\Rightarrow p(x_1,y_1))$
$\wedge(m(x_2,y_2)\Rightarrow p(x_2,y_2))$
$\wedge((p(x_3,z_1)\wedge p(z_1,y_3)) \Rightarrow gp(x_3,y_3))$
$\wedge(p(x_4,y_4)\Rightarrow a(x_4,y_4))$
$\wedge(p(x_5,z_2)\wedge a(z_2,y_5) \Rightarrow a(x_5,y_5)))$

Assume:
f is is-father-of
m is is-mother-of
p is is-parent-of
gp is is-grandparent-of
a is is-ancestor-of

FIGURE 14-10
Combination of multiple axioms.

tinctions, but most humans would prefer to see the original x's and y's than some made-up tags. Consequently, what many real systems do in such circumstances is to leave the original variable names alone, but assume that each clause has its own distinct set, with an unwritten "subscript" appended to each variable in a clause. This subscript is often some unique name given to the clause by the system and varies from clause to clause, or even from use to use of the same clause.

Having gone through this argument, we can now see another reason why the Horn clause subset of logic is so popular with real logic-based computing systems. Each of the original wffs translates into exactly one clause, and thus lumping all the clauses together still permits total accountability and traceability between the clauses used for inferencing and the original set of statements. Further, the unique subscript mentioned above can include the index by which the original programmer named or accessed the statement from which it was created.

14.5 DECIDABILITY ISSUES—THE HERBRAND RESULT

As stated before, a logic program corresponds closely to a set of axioms, and the input to such a program corresponds to a wff or set of wffs whose relation to the axioms is not immediately obvious to the user. Given this, it is fairly clear that the bread and butter of logic computing will be bound up with determining whether or not these input wffs follow logically from the program—that is, they are theorems of the axiom set. An algorithm that does such a determination is termed a ***decision method*** or ***decision procedure,*** and will thus form a large part of the inference engine for a logic computing system. Computationally, monitoring such a decision procedure (namely, recording a proof sequence and the interpretations so involved) often represents the outputs desired from the program.

Given the importance of a decision procedure, a critical question to ask is whether or not such a procedure exists for an arbitrary set of axioms. This question is called a ***decision problem,*** and for propositional logic the answer is yes: We need only enumerate the finite (but potentially large) set of truth assignments, and try each one. The results will tell us unambiguously if the input wff is, or is not, a theorem. This is comforting, since it gives us something to refer to when we try for other algorithms which give the same result but which are faster.

Unfortunately, the decision problem is not true for first-order predicate logic; there is no universal algorithm that, given an arbitrary set of axioms and another arbitrary wff **G**, will unambiguously say "yes, it is a theorem" or "no, it is not" in guaranteed finite time. Predicate logic is ***undecidable.***

A quick overview of how this result comes about is instructive. The problem would be decidable if we could write two programs for two computers, one of which computes in some order all possible wffs that are theorems of a set of axioms, and the other of which computes all possible wffs that are not theorems (see Figure 14-11). Running both computers in parallel and comparing the wffs so generated to the desired wff will eventually produce a match, with the identity of the processor causing the match thus indicating if the wff is a theorem or not.

It is possible to write the first of the above problems: We can generate in some order all possible theorems of an axiom set, and compare with the desired wff. For theoretical reasons, however, there exists no algorithm that is guaranteed to produce all and only wffs that are not theorems. Thus, given a wff, we can in fact guarantee that in finite time we can "check" that the wff is a theorem, but if it is not a theorem, we will never find it in the derived theorem list, and the program will most probably hang up in an unending loop.

Thus the decision problem for first-order logic is at best ***semidecidable,*** and we will limit our search for inference engines to those that check that wffs are theorems and ignore what happens if they are not theorems. Although this is weaker than we would like (no one relishes the idea of a program that can loop forever on "bad" data), it is the best that is possible.

14.5.1 The Herbrand Universe

(Chang and Lee, 1973, pp. 52–54)

Just as with truth tables and propositional logic, the process of generating theorems in first-order logic seems to go hand in hand with generating interpretations in some well-ordered and structured fashion. Such an order exists in what is called a ***Herbrand interpretation.*** For each axiom set this interpretation defines a special domain of objects, called the ***Herbrand universe,*** to which any other domain could be mapped, and then assigns mappings to the function and predicate symbols. Although there are many possible Herbrand interpretations for a given system, they differ only in the relations assigned to the predicate symbols and not in the objects used by the relations.

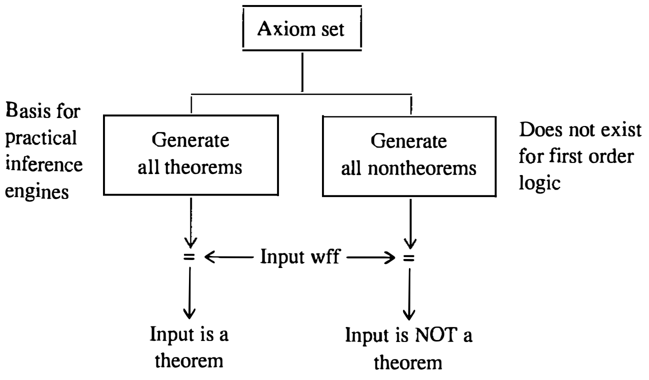


FIGURE 14-11
Showing decidability.

The Herbrand universe is a potentially infinite set **H** which includes all objects expressible as ***terms*** in the system (i.e., expressions which can be used as arguments) and is constructed iteratively from a series of simpler sets $\mathbf{H_0}$, $\mathbf{H_1}$,.... The original set $\mathbf{H_0}$ represents all the constants we know to be present, namely, those designated by the constant symbols used in the set of axioms. If no constant symbols exist, we invent one. Then, from any set $\mathbf{H_k}$, we create a new $\mathbf{H_{k+1}}$ by appending to $\mathbf{H_k}$ any term expressible as any one of the function symbols used in the axiom set applied to the appropriate number of objects from $\mathbf{H_k}$. Duplicates are eliminated. This is repeated until no new objects are added, at which point the set **H** corresponds to this final set.

It is possible that this process of building a new $\mathbf{H_k}$ goes on forever. In this case $\mathbf{H}=\mathbf{H_\infty}$.

Each of the expressions so added to **H** is treated as an object whose distinguishing name is its expression. There is no other value assumed or implied for it.

As an example of the basic step, if $\mathbf{H_1}$ consists of the objects **{a,g(a,a)}**, where **a** is a constant found in the axioms and **g** is a function symbol with arity 2, then $\mathbf{H_2}$ consists of

{a,g(a,a),g(a,g(a,a)),g(g(a,a),a),g(g(a,a),g(a,a))}

This process essentially finds all objects either explicitly defined in the system or constructible by applying functions to already defined objects. There is no "meaning" given to these constants other than their "names" as the appropriate character strings.

As a more complete example, Figure 14-12 derives one such universe. If we interpret the constant **t** as the boy Tim and the function symbol **f** as a function which returns its argument's father, then these Herbrand sets correspond to our notion of the paternal family tree of Tim. However, one could also consider **t** as standing for the number 0, and the function **f** being the successor function. In this case the Herbrand universe would equally well represent the set of all nonnegative integers.

Assume:
set of constant symbols = {t}
set of function symbols = {f} where f takes one argument.

Then:

H_0 = {t}

H_1 = {t,f(t)}

H_2 = {t,f(t),f(f(t))}

H_3 = {t,f(t),f(f(t)),f(f(f(t)))}

$H_\infty = \{t\}\cup\{f^i(t)|i>0\}$

FIGURE 14-12
A sample Herbrand universe.

Figure 14-13 gives another example. Here there are three constants and two functions. Again, for human readers the intended meaning is obvious, but, as before, other interpretations are possible. For example, all three constants could represent the tuple (0,0), with **father-of** incrementing the first component of its argument and **mother-of** incrementing the second component. The resulting universe would represent all pairs of nonnegative integers, which in turn could translate to all possible positive rational numbers (numerator and denominator).

Also as before, it is important to remember that each element of each $\mathbf{H_k}$ can be distinct, unlike the "human" interpretation, where several people can have the same parent.

All the expressions in **H** are called ***ground terms*** because they are firmly "grounded" in base constants and have no variables.

14.5.2 Herbrand Interpretations

Now the ***atom set*** or ***Herbrand base*** of a set of axioms is defined as the set of all predicate symbols applied to all possible tuples (of the right size) of elements from the Herbrand universe. Thus if **is-father-of** is a predicate symbol found in the axiom set, then the atom set would include all expressions of the form **is-father-of (t),** where **t** is a member of **H.**

Each such combination of a predicate symbol and a tuple of objects

Assume:
- set of constant symbols = {Mike, Mary, Tim}
- set of function symbols = {father-of, mother-of} where each takes one argument.

Then:

H_0 = {Mike, Mary, Tim}

H_1 = {Mike, Mary, Tim, father-of(Mike), mother-of(Mike),
father-of(Mary), mother-of(Mary),
father-of(Tim), mother-of(Tim)}

H_2 = {Mike, Mary, Tim, father-of(Mike), mother-of(Mike),
father-of(Mary), mother-of(Mary),
father-of(Tim), mother-of(Tim),
father-of(father-of(Mike)), father-of(mother-of(Mike)),
mother-of(father-of(Mike)), mother-of(mother-of(Mike)),
father-of(father-of(Mary)), father-of(mother-of(Mary)),
mother-of(father-of(Mary)), mother-of(mother-of(Mary)),
father-of(father-of(Tim)), father-of(mother-of(Tim)),
mother-of(father-of(Tim)), mother-of(mother-of(Tim))}

H_3 = ...

FIGURE 14-13
Another sample Herbrand universe.

from **H** is termed a ***ground atom,*** or ***ground literal,*** in symmetry with the previous definition of ground terms.

With these definitions we can describe a ***Herbrand interpretation*** as any mapping that has the following characteristics.

- Each constant symbol is mapped to some unique constant of the same name (the actual value is immaterial).
- Each function symbol **f** of arity n is mapped to a function from $\mathbf{H}^n$ to **H**, where the tuple $(\mathbf{t}_1, \ldots \mathbf{t}_n)$ (each $\mathbf{t}_i$ from **H**) is mapped to the unique object in **H** with name $\mathbf{f}(\mathbf{t}_1, \ldots \mathbf{t}_n)$.
- Each element in the atom set is mapped to either T or F.

The third step above assigns a relation to each predicate by assigning a T or F value to all possible tuples from the Herbrand universe applied to the predicate symbol. Those cases receiving a T value designate tuples that are in the relation; those receiving an F are not in the relation.

Note that all Herbrand interpretations share exactly the same mapping for constants and functions, and differ only in the mappings assigned to the predicates. Thus if N is the size of the atom set, then there are 2^N Herbrand interpretations for it, one for each possible way of mapping T and F to the individual ground atoms.

Figure 14-14 gives several such interpretations for Figure 14-12 where the predicate **is-father-of.** Any of the other possible truth assignments is equally valid. Thus if there were r possible tuples formable from elements in the Herbrand universe, there would be 2^r possible relations that could be assigned to each predicate, and 2^{pr} possible Herbrand interpretations for a system with p predicate symbols.

Finally, one of the beauties of Herbrand interpretations is that for any interpretation that we as humans might find useful for the symbols in a wff, it is possible to find a Herbrand one that has the same effects. The

Predicate symbol: is-father-of
Herbrand Universe from Figure 14–12 (t = Tim, f = father-of)

Ground Literal	Possible Interpretations		
is-father-of(Tim,Tim)	False	True	... True
is-father-of(father-of(Tim),Tim)	True	False	... True
is-father-of(Tim,father-of(Tim))	False	True	... True
is-father-of(father-of(Tim), father-of(Tim))	False	True	... True
...			
is-father-of(father-of(father-of(Tim)), father-of(Tim))	True	False	... True
....			

FIGURE 14-14
Some simple Herbrand predicate interpretations.

basic difference between a non-Herbrand and a Herbrand interpretation is in the assignment of function mappings. A Herbrand interpretation assumes that each unique tuple constructed from unique constant symbols always gives a new unique result. Real functions often cascade multiple argument tuples into the same result. Consider the function symbol+and the constants **a** and **b.** A Herbrand interpretation assumes that $+(\mathbf{a},\mathbf{b})$ and $+(\mathbf{b},\mathbf{a})$ are different, whereas conventional arithmetic assumes that $+(\mathbf{a},\mathbf{b})=+(\mathbf{b},\mathbf{a})$. The approach to finding a Herbrand interpretation whose effect is the same as one assuming a function mapping as above is to assign consistent true/false values to elements of the atom set whose argument expressions represent tuples that the other interpretation says should be the same. For example, if **p** is a unary predicate, then if we want the effect of the above interpretation for +, we must be sure to choose a Herbrand interpretation in which $\mathbf{p}(+(x,y))$ and $\mathbf{p}(+(y,x))$ receive the same truth value for any expressions x and y.

The consequence of this result is that when answering questions about the satisfiability of a wff, we need look only at Herbrand interpretations, with a guarantee that this covers all other possible interpretations.

14.6 THE HERBRAND INFERENCE ENGINE

(Gilmore, 1960)

The Deduction Theorem mentioned in Section 13.2.5 gives two ways of proving that some new wff **G** follows from some set of axioms $\mathbf{A}_1, \ldots, \mathbf{A}_n$. One either shows that the new wff $(\mathbf{A}_1 \land \ldots \land \mathbf{A}_n) \Rightarrow \mathbf{G}$ is true for all interpretations, or that $(\mathbf{A}_1 \land \ldots \land \mathbf{A}_n \land \neg \mathbf{G})$ is false for all interpretations (is unsatisfiable). The previous discussions on Herbrand interpretations give an orderly progression for generating such interpretations, but the task of testing them all still seems infinite.

What saves the day is a fundamental result called ***Herbrand's Theorem,*** which says that if a wff is unsatisfiable, we can determine this after analyzing only a finite number of possible substitutions from the Herbrand universe. This is most easily understood by building a wff according to the second form of the Deduction Theorem, and converting it into clausal form, $\forall x_1 \ldots |(\mathbf{C}_1 \land \ldots \mathbf{C}_n)$, where each $\mathbf{C}_k$ is the **or** of some literals. As mentioned earlier, this is equivalent to repeating the quantifier-free expression $(\mathbf{C}_1 \land \ldots \mathbf{C}_n)$ an infinite number of times, each time substituting a different set of objects for the variables. Each substitution leaves no variables behind, only connections of predicates applied to constants or functions of constants. Such clauses are called ***ground instances*** of the original clause. Given that they are now variable-free, the resulting wff is equivalent to an expression in propositional logic.

Note that in this form, as soon as any one of the ground clauses becomes false, the whole wff is false. Thus, if the wff is truly unsatisfiable, then under any interpretation (which results in assigning true/false values to each literal), at least one of these clauses will become false. Again, each literal, having no variables in it, is equivalent to a propositional letter.

The Herbrand Theorem states that we need not look at the infinite conjunction of such ground clauses. After only a finite number of substitutions from the Herbrand Universe, we will end up with a finite number of clauses which are themselves inconsistent (false under all interpretations). Further, testing a finite number of ground clauses for inconsistency is feasible: Just as in propositional logic, we need only do a truth table analysis assuming that each distinct predicate and argument expression is a separate propositional letter. If the final column of such an analysis is all false, then that set of clauses is inconsistent under all possible interpretations, and we are done.

This process leads directly to a possible inference engine for verifying that a wff **G** is a theorem of a set of axioms (Figure 14-15). We start by combining the axioms with the negation of **G** and converting to clausal form. Then, starting with $\mathbf{H}_0$, we iterate through the Herbrand sets $\mathbf{H}_k$. For each set we form all possible substitutions from the set into the variables in the clauses of the body, make the substitutions, and combine together in a large conjunction. This conjunction is then analyzed for unsatisfiability using any technique suitable for propositional calculus. If it is satisfiable, the process is repeated for the next bigger set. As soon as we find a case where it is unsatisfiable, we stop—the original wff is a theorem.

Within months of the appearance of Gilmore's original paper, a dramatically improved procedure was developed (Davis and Putnam, 1960). Their procedure used a clausal form of the expansion but with the following analysis rules:

1. Delete all clauses that are known tautologies. These will never yield a false, so they can be ignored.
2. If there is some unit clause **L** (i.e., there is only one atom in it), delete it and all clauses that include it. If the resulting wff is empty, stop because the wff is satisfiable (go on to the next Herbrand set).

   If the resulting wff is not empty, delete ¬ **L** from any clause that contains it. If any of these deleted clauses are themselves unit clauses, stop; the wff is unsatisfiable.
3. If a literal appears ***pure***, that is, **L** appears but ¬ **L** is never used, delete all clauses that contain it.
4. If the wff is of the form

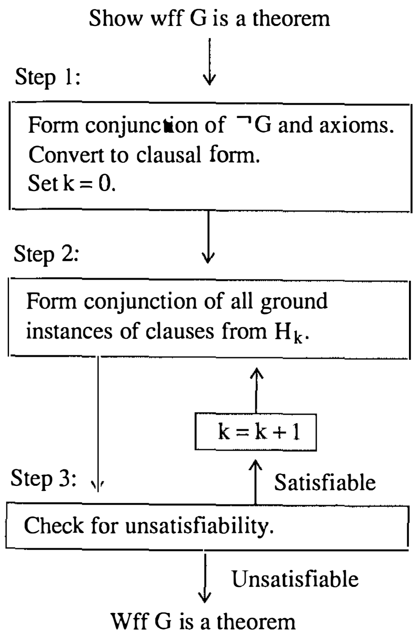
   $(\mathbf{A}_1 \vee \mathbf{L}) \wedge \ldots \wedge (\mathbf{A}_m \vee \mathbf{L}) \wedge (\mathbf{B}_1 \vee \neg \mathbf{L}) \wedge \ldots \wedge (\mathbf{B}_n \vee \neg \mathbf{L}) \wedge \mathbf{C}$

   where all **A**'s, **B**'s, and **C** are clauses free of either **L** or ¬ **L**, then split the wff into two:

   $$\mathbf{A}_1 \wedge \ldots \wedge \mathbf{A}_m \wedge \mathbf{C}$$
   $$\mathbf{B}_1 \wedge \ldots \wedge \mathbf{B}_n \wedge \mathbf{C}$$

   If both of these are themselves unsatisfiable, then so is the original wff.

If after taking this process as far as it will go there still is some wff left, then we must go on to the next Herbrand set.



- Step 1: Convert program $A_1, \ldots A_n$ and goal G to
  $\forall x \ldots | (A_1 \wedge \ldots \wedge A_n \wedge \neg G)$
- Step 2: $\alpha_1 (A_1 \wedge \ldots \wedge A_n \wedge \neg G) \wedge \ldots \wedge \alpha_m (A_1 \wedge \ldots \wedge A_n \wedge \neg G)$
  where α_k is a substitution from H_k (of size m)
- Step 3: Do Truth Table Analysis

p_1	p_2	$c_{i,1}$	...	$c_{i,n}$	wff
T	T	T	...	T	?
T	T	T	...	F	?
		...			
F	F	F	...	F	?

where p_k from atom set and $c_{i,j} = \alpha_i c_k$

- Step 4: Repeat 2 and 3 with next H_{k+1}

FIGURE 14-15
A Herbrand inference engine.

The reader should, at this point, be able to understand the origins of each of these steps. Figure 14-16 gives a sample analysis where unsatisfiability occurs in the 0-th step.

14.7 PROBLEMS

1. For each variable *x, y, z, or w* in the following wffs, indicate whether it is free or bound (and to which quantifier):

   a. $\forall w | (\forall x | (\exists y | \mathbf{A}(y,x,w))) \wedge ((\exists x | \mathbf{Q}(x,w)) \Rightarrow (\exists y | \mathbf{Q}(y,x)))$

Axioms: $p(1), p(2), \forall x | p(x) \wedge p(y) \Rightarrow q(f(x),y)$
Show: $q(f(1),2)$ is a theorem

Clausal form: $\forall xy | (p(1) \wedge p(2) \wedge (\neg p(x) + \neg p(y) + q(f(x),y)) \wedge \neg q(f(1),2)$

For H_0 expression is:

$$
\begin{aligned}
&p(1) \wedge p(2) \wedge (\neg p(1) \vee \neg p(1) \vee q(f(1),1)) \wedge \neg q(f(1),2) \quad \alpha = [1/x, 1/y] \\
&\wedge p(1) \wedge p(2) \wedge (\neg p(1) \vee \neg p(2) \vee q(f(1),2)) \wedge \neg q(f(1),2) \quad \alpha = [1/x, 2/y] \\
&\wedge p(1) \wedge p(2) \wedge (\neg p(2) \vee \neg p(1) \vee q(f(2),1)) \wedge \neg q(f(1),2) \quad \alpha = [2/x, 1/y] \\
&\wedge p(1) \wedge p(2) \wedge (\neg p(2) \vee \neg p(2) \vee q(f(2),2)) \wedge \neg q(f(1),2) \quad \alpha = [2/x, 2/y]
\end{aligned}
$$

Analysis:

p(1)	p(2)	q(f(1),1)	q(f(1),2)	q(f(2),1)	q(f(2),2)	Result
T	T	T	T	T	T	F
		.....				...
F	F	F	F	F	F	F

All false, thus unsatisfiable————————↑

FIGURE 14-16
A sample Herbrand proof.

b. $\forall x | (\mathbf{P}(a,x) \Rightarrow \exists x | \mathbf{Q}(a,y,x))$
c. $\exists y | ((\forall x | (\mathbf{P}(x) \Rightarrow \mathbf{Q}(\mathbf{f}(x),y))) \Rightarrow ((\exists y | \mathbf{P}(y)) \Rightarrow (\exists x | \mathbf{Q}(x,y))))$

2. Convert the first three axiom wffs for the picnic problem of Chapter 13 into clausal form, showing all steps.
3. Transform the following into clausal form:
   a. $((\mathbf{p} \Rightarrow \mathbf{q}) \wedge \neg \mathbf{p}) \Rightarrow \neg \mathbf{q}$
   b. $\exists u | \exists v | \forall w | \forall x | \exists y | \forall z | \exists q | \forall s | \mathbf{p}(q,s,u,v,w,x,y,z)$
   c. $\forall x | (\mathbf{P}(a,x) \Rightarrow \exists y | \mathbf{Q}(a,y,x))$
   d. $\forall x | (\mathbf{P}(a,x) \Rightarrow \exists x | \mathbf{Q}(a,y,x))$
   e. $\exists y | ((\forall x | (\mathrm{P}(\mathrm{x}) \Rightarrow \mathbf{Q}(\mathbf{f}(x),y))) \Rightarrow ((\exists y | \mathbf{P}(y)) \Rightarrow (\exists x | \mathbf{Q}(x,y))))$
4. Assume that wffs in propositional logic have been converted to clausal form and written as s-expression lists, where each element of the topmost list corresponds to a clause and each clause is written as a list itself of either ⟨proposition⟩ or "(NOT ⟨proposition⟩)." For example, $(\neg \mathbf{d} \vee \mathbf{e} \vee \mathbf{f}) \wedge (\mathbf{h} \vee \neg \mathbf{f})$ would be:

   (((NOT d) e f) (h (NOT f)))

   Write an abstract function **unsat**(*wff, letters*), where *letters* is a list of the propositional letters in the wff, which determines whether or not such a wff is unsatisfiable.
5. Assume the axioms $\forall x | (\mathbf{p}(x) \Rightarrow \mathbf{q}(x))$ and $\neg \forall y | \mathbf{q}(y)$. Prove using the Herbrand Inference Engine that $\neg \forall z \mathbf{p}(z)$ is a theorem of them.
6. Show that the analysis rules of Davis and Putnam (1960) will in fact determine if a wff is unsatisfiable. What happens if it is satisfiable?

CHAPTER 15

FUNDAMENTALS OF PRACTICAL INFERENCE ENGINES

An inference engine is the computational heart of a logic-based computing system. As described earlier, it takes the logic program, which consists of a set of logic statements, and an input statement, and deduces either what else is true, or what substitutions need to be made to make part of the input true. The process of making these deductions involves three steps (see Figure 15-1):

1. Selecting which subset of statements from the program to look at next
2. Selecting which general rule of inference might govern the deduction of a new statement from this set
3. Verifying that the set of statements does in fact satisfy the inference rule, and developing the resulting inference

The first two steps are performed by the inference engine's ***decision procedure.*** The final step usually consists of a tightly integrated pattern-matching process and a substitution-generation process.

This chapter gives an overview of the possible options for each step. Other chapters will detail specific combinations for specific language models, such as PROLOG or OPS. The order of presentation here is the opposite of the steps given above. We will start with the pattern-matching and substitution-generation process, called ***unification*** in its

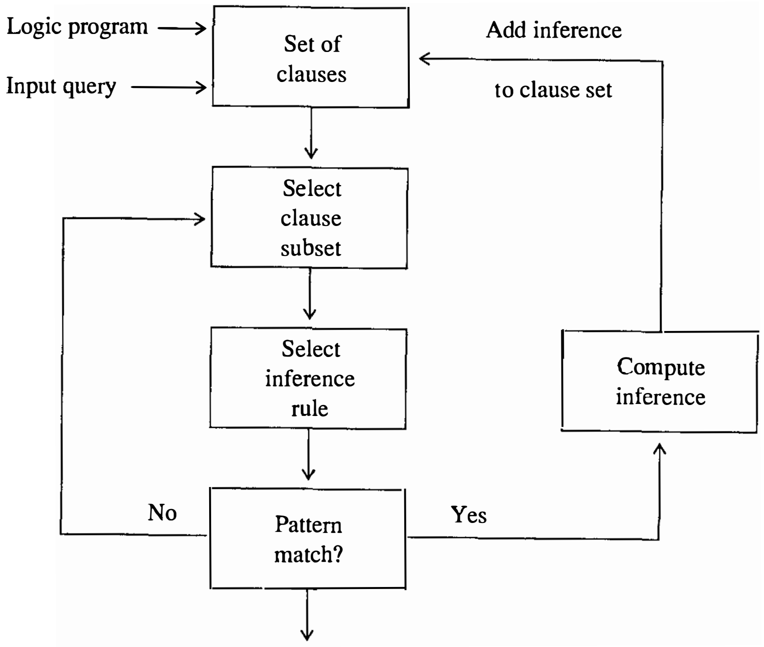


FIGURE 15-1
Parts of an inference engine.

most general form. This is the core of the inference engine in terms of computation, and a good understanding of it is needed before one tackles the possible options for a decision procedure.

Following this we will address the major varieties of inference rules in use today, with emphasis on the one that probably is most important, ***resolution.*** We will discuss which ones are computationally more "powerful" than others, but as with the different kinds of application orders for functional languages, we will give only the key results. Derivations of these results will be left to the references.

Finally, we will discuss decision procedures. Again, this will consist of introductions of the major variations that are possible, and an indication of why certain ones may be of more use than others. Specific decision procedures will be developed in later chapters in conjunction with specific inference rules and logic languages.

15.1 PATTERN MATCHING AND UNIFICATION

(Wos et al., 1984, pp. 421–423; Nilsson, 1980, pp. 140–144; Chang and Lee, 1973, pp. 76–79)

All of the inference rules described in this text share at least one common property: They all take two or more wffs, find some part of these wffs that "match" to some degree, "cancel out" the matching part, and "assemble" a new wff from the remaining pieces. This matching process is straightforward for propositional calculus, since we can easily convert wffs to some standard form and then need only look for the same propositional letter and/or identical wffs constructed from the same letters. For example, from wffs

$$((\mathbf{A} \wedge \mathbf{C}) \Rightarrow \mathbf{B}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{D}) \qquad \text{and} \qquad (\neg(\mathbf{C} \wedge \mathbf{A}) \vee \mathbf{B})$$

the modus ponens rule might convert the second term to $((\mathbf{C} \wedge \mathbf{A}) \Rightarrow \mathbf{B})$, recognize it as identical to a term in the first, cancel it out of both, and create the new wff, $(\mathbf{C} \Rightarrow \mathbf{D})$.

This process is not so easy in predicate logic systems because of the existence of variables. Consider, for example, the wffs

$$\textbf{is-father-of}(\text{Peter,Tim})$$

and

$$(\forall xy | \textbf{is-father-of}(x, y) \Rightarrow \textbf{is-parent-of}(x, y)).$$

Clearly, the inferred wff should be **is-parent-of**(Peter,Tim), but both the initial match and the construction of the new wff involves more complexity than before. The matching process must assign values to the variables x and y, and use these as substitutions during the construction phase. This process of finding substitutions that make wffs match is the heart of logic-based computing, and is termed ***unification.***

15.1.1 The Role of Substitutions

More concretely, unification takes subexpressions (positive literals in most cases) from two (or occasionally more) wffs and finds a ***substitution*** that when applied to both expressions makes them identical. As always, this substitution can be represented as the assignment of some values to some free identifiers in the subexpressions. The process of finding this general substitution works recursively over each argument position of the literals, making it match, saving the needed local substitutions, and moving on.

It is possible for unification to ***fail***—for example, if two argument positions are found that cannot be matched, or if the substitution requires assigning two different values to the same variable. In such cases there is no match, and the set of wffs that was being tried is marked as inapplicable in this situation. As an example, consider trying to find a substitution to make $\mathbf{p}(12,22)$ and $\mathbf{p}(x,x)$ the same. This must fail because it requires assigning two different values, 12 and 22, to the variable x.

Formally, we say that a substitution $\alpha=[\mathrm{A}_1/x_1,\ldots,\mathrm{A}_n/x_n]$ is a ***unifying substitution*** or ***unifier*** for literals $\mathrm{C}_1,\ldots,\mathrm{C}_m$ if, when applied to each of the literals individually, it results in absolutely the same common literal, i.e.,

$$\alpha C_1 = \alpha C_2 = \ldots = \alpha C_m$$

For the example at the beginning of this section, the result of unifying the literals **is-father-of**(Peter,Tim) and **is-father-of**(x,y) is the substitution [Peter/x,Tim/y].

It is possible for more than one unifier to exist for the same pair of expressions. For example, consider the literals **p**(Peter,Tim,y,z) and **p**(Peter,x,y,w), where w, x, y, and z are all free variables. There are an infinite number of unifiers possible (consider [Tim/x,z/w], [Tim/x,z/w,Beth/y], [Tim/x,Mary/z,Mary/w,**f**(π)/y], . . .). Applying any of these substitutions to the two literals makes them the same, although this "same" result differs from unifier to unifier. The first one, for example, yields **p**(Peter,Tim,y,z), while the second yields **p**(Peter,Tim,Beth,z).

Out of all possible unifiers between a set of literals, a unifier **α** is the ***most general unifier*** (or ***mgu***) if any other unifier for the same set of literals can be constructed from **α** by adding more substitutions. In a sense the most general unifier is also the ***minimal substitution*** needed to match the literals. It makes the ***least commitment*** necessary to make the literals match.

In the example above, only one of the unifiers ([Tim/x,z/w]) is most general. The others can be constructed from it by adding additional substitutions (e.g., [Tim/x,z/w,Beth/y]=[Beth/y][Tim/x,z/w]).

In nearly all situations, this mgu leaves as many variables without substitution values as possible.

15.1.2 Classical Unification

The classical algorithm for computing such most general unification substitutions is fairly straightforward. Figure 15-2 gives one version using abstract programming and an abstract syntax representation that can easily be adapted to almost any real wff syntax. The function ***unify*** accepts as input two literals, checks whether they have the same predicate symbol, and calls the function **unify-args** to process the arguments. This function returns a list of pairs of variables and values representing the substitution. If unification is not possible, one of the two functions returns F. Note that the format for the substitution is the same as that used before for association lists, namely, a list of pairs of the form (⟨name⟩.⟨value⟩).

Unify-args is basically a loop over the argument lists which for each argument position executes an if-then-else tree to compare the actual arguments in that position. Note that before we compare two arguments we first apply the unification substitution as it currently exists. This is to take into account the matching efforts that have gone before, which may have assigned values to various variables. If after the substitution the ar-

```
unify(x, y) = ''function to unify positive literals x and y and return most general
substitution as association list.''
  if get-predicate-symbol(x) = get-predicate-symbol(y)
  then unify-args(get-args(x),get-args(y),nil)
  else F

unify-args(x,y,s) = ''unify the argument lists x and y, using the substitution list s of
                    pairs (name.value).''
                    ''Return s if successful, F if not.''
  if null(x) then s
  else let a=substitute(car(x),s)
      and b=substitute(car(y),s) in
         if is-a-constant(a) and is-a-constant(b)
         then if a=b then unify-args(cdr(x),cdr(y),s) else F
         elseif is-a-variable(a)
         then if is-a-variable(b)
             then unify-args(cdr(x),cdr(y),(a.z).(b.z).s)
                 where z=newid()
             else unify-args(cdr(x),cdr(y),(a.b).s)
         elseif is-a-variable(b)
         then unify-args(cdr(x),cdr(y),(b.a).s)
         elseif get-function-symbol(a) = get-function-symbol(b)
         then let s1=unify-args(get-args(a), get-args(b),s) in
             if sl =F then F
             else unify-args(cdr(x),cdr(y),s1)
         else F

substitute(x,s) = ''returns the expression resulting from applying substitution list s to
                  expression x.''
  if is-a-constant(x) then x
  elseif is-a-variable(x)
  then let a=assoc(x,s) in
      if null(a)
      then x
      else substitute(cdr(a),s)
  else ''x must be an expression''
  create-term(get-function-symbol(x), subs-args(get-args(x),s))
  whererec subs-args(x,s) =
          if null(x) then nil
          else substitute(car(x),s).subs-args(cdr(x),s)

assoc(x,s) = ''returns (x.value) if that pair in s, else nil'' =
  if null(s) then nil
  else let pair=car(s) in
      if car(pair)=x
      then pair
      else assoc(x,cdr(s))
```

FIGURE 15-2
Basic unification algorithms.

gument expressions are constants and identical, no further substitutions need be added, and we can advance to the next argument. Two constants that are different are grounds for declaring an immediate mismatch for the entire expressions.

If exactly one of the arguments is a variable, we can add a new pair to the substitution and advance. If both are variables, then whatever is substituted for one must be substituted for the other. The approach taken in Figure 15-2 adds two entries to the substitution list. A new variable that has never been used before is substituted for each of the variables. This guarantees that any future substitution into either one will result in replacement of the other. There are other variations, such as are taken in many PROLOGs, that do not require a new variable but depend on some specific implementation details.

In the case of an expression in an argument position, the unification process checks that the same function symbol starts the expression in both arguments and then recursively attempts to unify all arguments for that function. If this is successful, we return to unifying the original argument list; if not, we abort the unification process immediately.

Figure 15-3 lists some sample literals and the substitution resulting from their unification. The reader should go through them to check his or her understanding. Additionally, Figure 15-4 diagrams a more complex unification, with intermediate details shown.

15.1.3 Occurs Check

The ***substitution*** function of Figure 15-2 appears at first glance to be straightforward. If the input expression is a constant, it is returned directly. If it is a variable, but not one that is already in the substitution, it too is returned without modification. In the case of an expression, the expression is taken apart, substitutions are applied to each argument, and the results are put back together again.

Literal 1	Literal 2	Unifier
p(Roy,Tim)	p(x,Tim)	[Roy/x] = ((x.Roy))
p(x,x)	p(Tim,Tim)	[Tim/x] = ((x.Tim))
p(x,x)	p(Tim,Roy)	no match
p(f(Pete),Tim)	p(x,Tim)	[f(Pete)/x] = (x.f(Pete))
f(x,x)	f(y,3)	[z/x,z/y,3/z] = ((x.z)(y.z)(z.3))
f(g(x,h(2,y)),z)	f(g(1,h(3,4)),4)	no match
f(g(x,h(1,y)),z)	f(g(2,h(w,3)),w)	[2/x,1/w,3/y,1/z] = ((x.2)(w.1)(y.3)(z.1))
f(x,g(x,x))	f(y,g(12,y))	[z/x,z/y,12/z] = ((x.z)(y.z)(z.12))
f(3,g(h(h(x)),x))	f(y,g(h(z),h(y))))	[3/y,h(x)/z,h(3)/x] = ((y.3)(z.h(x))(x.h(3)))

FIGURE 15-3
Sample unifications.

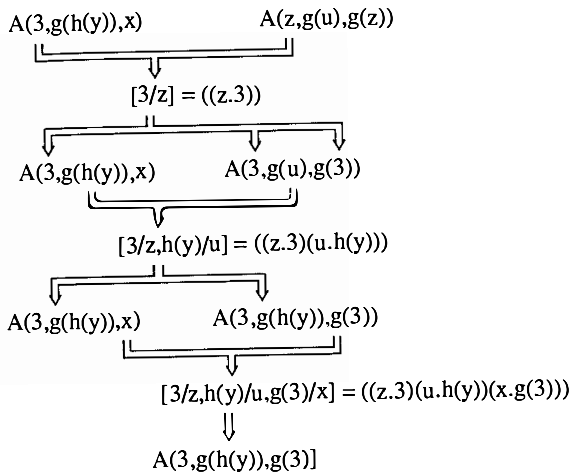


FIGURE 15-4
A detailed unification example.

The subtlety in this process occurs when we find a variable for which a value exists in the current substitution list. We cannot simply return the value directly, since it might itself be a variable (or an expression containing a variable) for which a value exists in the substitution. Consequently, we must repeat **substitute** on this value using the full substitution list. When no more substitutions are possible, the pieces are put back together and returned.

The problem with this is that there are situations where the substitution process can get caught in an infinite loop. Consider, for example, a substitution containing "[. . .f(x)/y,g(y)/x. . .]." Upon finding a lone x, **substitute** will replace it by g(y), then by g(f(x)), then substitute for x again to yield g(f(g(y))), then g(f(y(f(x)))), and so on forever. This problem occurs whenever the ultimate value to be substituted for some variable includes in its representation one or more copies of exactly the same variable.

To protect against this possibility, the functions in Figure 15-2 can be modified to perform an ***occurs check*** for this possibility every time a new substitution is added to a substitution list. This typically involves recording at each level of a substitution what variable is being replaced, and if its replacement value contains free occurrences of either that variable or any of the variables in the chain of substitutions leading to it. If so, an occurs check has been detected. Depending on the system, this may either be taken as an error condition or somehow flagged as a special, potentially infinite, object. Further, depending on the data structures used and the exact implementation of the comparison and substitution process, the process itself can be computationally quite expensive.

Given that useful instances of this situation are quite few in practice, most logic system forego the occurs check testing and use some variation of Figure 15-2, even though it may loop forever when presented with such a pathological input.

15.1.4 Notes

One other note about the unification process is that as stated above it applies no interpretation to function symbols. It is a pure ***pattern matcher.*** Thus it will not attempt to "simplify" expressions such as "3+4," and will declare a mismatch if the terms presented for unification include argument pairs of the form "3+4" and "7." At first glance this seems improper, but in reality it is a better match to the underlying framework of logic, plus it buys significant computational advantages. To cover the case where the user really wants such simplifications, most real logic-based computing systems include explicit escapes in the syntax to signal to the unifiei when "evaluation" should be performed before pattern matching.

It is interesting to contrast this deliberate lack of function application reduction with functional languages such as LISP, in which one needs an explicit function such as **quote** to turn off evaluation.

Finally, wе should note that this definition of **unify** is complete only for first-order logic, where the quantified variable represent objects that can appear in the arguments of predicates. In higher-order logics, function symbols, and even predicate symbols, are quantifiable, and the unification algorithm given above must be so adapted.

15.2 COMMON INFERENCE RULES

(Chang and Lee, 1973; Wos et al., 1984, pp. 162–168)

The Herbrand-based inference engines of Chapter 14 are conceptually simple but so computationally intensive as to be almost worthless for any realistic problem. To overcome this, researchers have for the last 25 years investigated alternative methods for theorem verification of less computational complexity. These investigations have largely involved the development of ***inference rules*** that take one set of wffs and deduce directly from them that some other wff is a theorem (provable) from them, without exhaustive examination of different interpretations. Further, as described earlier, inference rules that are most useful are ***complete*** and ***sound.*** If used properly, they can produce all, and only, those wffs which are derivable from the set of axiom wffs.

This section describes the most common such rules in terms of the number and kind of wffs they need as input, and when and how they invoke a pattern-matching or unification process as discussed in the last section. The description here is in the framework of first-order predicate calculus; other formulations of the rules will be addressed in later chapters when real logic systems are reviewed.

Also note that the mere description of an inference rule gives little or no indication of under what conditions it might be invoked in a real system, and what such a system would do with the results of the inference. Such strategies are the domain of the decision procedure part of an inference engine, and are discussed in the next section.

15.2.1 Major Inference Rules

Figure 15-5 lists the three major classes of inference rules: ***modus ponens, modus tollens,*** and ***resolution,*** with examples of each. The first two are fairly straightforward, and are primarily of academic interest. The last one, resolution, is the cornerstone of many current logic systems, and will be addressed in detail shortly.

The primary difference between these rules for first-order logic and equivalent ones for propositional logic are that the pattern match is not a simple exactness test. Two literals in the wffs must be unifiable, with the resulting substitution used in the creation of the inference.

Variations and extensions of all three rules are clearly possible. For example, a variation of modus ponens termed ***syllogism*** takes two wffs of the form $(\mathbf{B}_1 \vee \mathbf{B}_2)$ and $(\mathbf{B}_2 \Rightarrow \mathbf{A})$ and infers the wff $(\mathbf{B}_1 \vee \mathbf{A})$. Another variation, this time of resolution, might take n wffs $\mathbf{B}_1$ through $\mathbf{B}_n$ and the wff $(\mathbf{B}_{1a} \wedge \ldots \wedge \mathbf{B}_{na} \wedge \mathbf{C}) \Rightarrow \mathbf{A}$ and deduce the wff $\mathbf{C} \Rightarrow \mathbf{A}$ as a result.

Rule Name	Inference
Modus Ponens:	$B_a,(B_b \Rightarrow A) \vdash \alpha A$ where α = unify(B_a,B_b) Eg., from: is-father-of(Peter,Tim) and: $\forall xy \mid$is-father-of(x,y)$\Rightarrow$is-parent-of(x,y) deduce: is-parent-of(Peter,Tim) unifier: [Peter/x,Tim/y]
Modus Tollens:	$\neg A_a,(B \Rightarrow A_b) \vdash \alpha \neg B$ where α = unify(A_a,A_b) Eg., from: $\neg$is-parent-of(Ralph,Tim) and: $\forall xy \mid$is-father-of(x,y)$\Rightarrow$is-parent-of(x,y) deduce: $\neg$is-father-of(Ralph,Tim) unifier: [Ralph/x,Tim/y]
Resolution:	$(P \vee Q_a),(\neg Q_b \vee R) \vdash \alpha (P \vee R)$ where α = unify(Q_a,Q_b) Eg., from: ($\neg$is-father-of(Peter,x)$\vee$is-parent-of(Peter,x)) and: ($\neg$is-parent-of(y,z)$\vee$is-ancestor-of(y,z)) deduce: ($\neg$is-father-of(Peter,w)$\vee$is-ancestor-of(Peter,w)) unifier: [Peter/y,w/x,w/z]

Notes: A and B arbitrary wffs, and P,Q, and R arbitrary literals.

FIGURE 15-5
The major inference rules.

15.2.2 The Generality of Resolution

Of these inference rules, resolution is the most general. To demonstrate this, consider Figure 15-6. Here each of the nonresolution rules is recast into what it would do to appropriate wffs in clausal form. In all cases, the inferences made by these inference rules correspond exactly to what could have been derived by resolution. For this reason, once the basics of resolution are understood, most people find resolution much more comprehensible than the other forms. This carries over to implementations in computer programs, and is one reason why resolution is considered a watershed in the development of logic languages.

In reviewing Figure 15-6 the reader should pay particular attention to the numbering and signs assumed about each literal and clause, and where the unifying substitution $\boldsymbol{\alpha}$ comes from.

15.3 RESOLUTION-BASED INFERENCE RULES

(Robinson, 1965)

Of the three primary inference rules, resolution and its variants are recognized today as the most general and efficient. At its core, resolution assumes wffs converted into clausal form, with all clauses **and**ed together. Resolution takes two or more of these clauses (called the ***parent clauses*** or ***ancestor clauses***) as input, combines them (**or**s the literals) using appropriate substitutions, with one or more literals "cancelled out." This resulting clause is the ***resolvant.***

The literals that are cancelled out have certain properties. First, one must come from each parent clause, and they must have the same predicate symbol but opposite signs. Second, there must be no variables between the two clauses with the same name but referring to different objects (renaming may be necessary). This might come about if each of the literals is within the scope of different quantifiers that quantify the same identifier symbol. Finally, after stripping off the negation from the negative literal, it must unify successfully with the positive one. If all these conditions are met, the inferred clause consists of applying the substitution from the unification process to all literals except the matching ones in both parent clauses, and **or**-ing the results together.

Inference Rule	Normal Form	Clausal Form
Modus Ponens	$B_1,(B_2 \Rightarrow A) \vdash \alpha A$	$B_1,(\neg B_2 \vee A) \vdash \alpha A$
Modus Tollens	$\neg B_1,(A \Rightarrow B_2) \vdash \neg \alpha A$	$\neg B_1,(\neg A \vee B_2) \vdash \neg \alpha A$
Syllogism	$(A \vee B_1),(B_2 \Rightarrow C) \vdash \alpha(A \vee C)$	$(A \vee B_1),(\neg B_2 \vee C) \vdash \alpha(A \vee C)$

Where in all cases $\alpha = \text{unify}(B_1,B_2)$

FIGURE 15-6
Resolution form of nonresolution rules.

As an example, consider resolving $(\mathbf{P} \vee \mathbf{L}_1)$ and $(\mathbf{Q} \vee \neg \mathbf{L}_2)$, where **P**, **Q**, and **L** are all literals. If both of these clauses are in the same wff, then at least one of the literals in each must be true under any possible interpretation. If the two **L**'s unify, that is, $\boldsymbol{\alpha}=\mathbf{unify}(\mathbf{L}_1,\mathbf{L}_2)$, then $\boldsymbol{\alpha}\mathbf{L}_1$ is identical to $\boldsymbol{\alpha}\mathbf{L}_2$. In turn, this means that whenever $\boldsymbol{\alpha}\mathbf{L}_1$ is true, $\neg\boldsymbol{\alpha}\mathbf{L}_2$ is false, and vice versa. And since both $\boldsymbol{\alpha}(\mathbf{P} \vee \mathbf{L}_1)$ and $\boldsymbol{\alpha}(\mathbf{Q} \vee \neg \mathbf{L}_2)$ are true under all interpretations, then it must be that either $\boldsymbol{\alpha}\mathbf{P}$ or $\boldsymbol{\alpha}\mathbf{Q}$ must be true. This leads to the conclusion that $(\boldsymbol{\alpha}\mathbf{P} \vee \boldsymbol{\alpha}\mathbf{Q})$ must also be true under all interpretations—exactly what resolution projects.

15.3.1 The Primary Resolution Rules

Actually, there are not one but a whole spectrum of resolution-based inference rules. Variations come about primarily by considering the number of clauses brought together in a single inference step, the number of literals to be found in each clause, and the types of literals permitted. Figure 15-7 lists some of the more standard variations. ***Binary resolution*** is the most basic form, with the others expressible as a series of binary resolutions.

15.3.2 Common Auxiliary Inference Rules

In addition to the above rules, there are several other auxiliary resolutionlike inference rules that are often useful. Figure 15-8 lists these.

Binary resolution: $(A \vee L_a), (\neg L_b \vee B) \vdash \alpha(A \vee B)$ where $\alpha = \text{unify}(L_a,L_b)$

Unit resolution: $L_a, (\neg L_b \vee B) \vdash \alpha B$ where $\alpha = \text{unify}(L_a,L_b)$

UR resolution; $L_{a1},\ldots L_{am}, (\neg L_{b1} \vee \ldots \vee \neg L_{bm} \vee B) \vdash \alpha B$
where $\alpha_1 = \text{unify}(L_{a1},L_{b1})$
and $\alpha_2 = \text{unify}(\text{substitute}(L_{a2},\alpha_1),\text{substitute}(L_{b2},\alpha_1))$
and ...
and $\alpha = \alpha_m = \text{unify}(\text{substitute}(L_{am},\alpha_{m-1}),\text{substitute}(L_{bm},\alpha_{m-1}))$

Hyperresolution: $(L_{a1}{}^{+} \vee A_1{}^{+}),\ldots (L_{am}{}^{+} \vee A_m{}^{+})$,
$(\neg L_{b1}{}^{+} \vee \ldots \vee \neg L_{bm}{}^{+} \vee B^{+}) \vdash \alpha(A_1{}^{+} \vee \ldots A_m{}^{+} \vee B^{+})$
where α computed as in UR resolution.

Negative Hyperresolution: $(\neg L_1{}^{+} \vee A_1{}^{+}),\ldots (\neg L_m{}^{+} \vee A_m{}^{+})$,
$(L_1{}^{+} \vee \ldots \vee L_m{}^{+} \vee B^{+}) \vdash \alpha(A_1{}^{+} \vee \ldots A_m{}^{+} \vee B^{+})$
where α computed as in UR resolution.

Notes:

- A and B all arbitrary conjunctions of literals
- L_k arbitrary literal
- "+" denotes literal is positive.

FIGURE 15-7
Variations in resolution-based inference rules.

Paramodulation: $((e_1=e_2)\lor A)$, $(L(\ldots e_3\ldots)\lor B) \vdash \alpha(L(\ldots e_2\ldots)\lor A\lor B)$
where α = unify-term(e_1,e_3).
Example: $(g(h(1))=2)\lor A)$, $(L(f(g(x)))\lor L1(x)) \vdash (A\lor L(f(2))\lor L1(h(1)))$

Demodulation: $(e_1=e_2)$, $(B(\ldots e_3)\lor A) \vdash \alpha(B(\ldots e_2\ldots)\lor A)$
Where α = unify-term(e1,e3) without replacing any variables.
Also discard original nonequality clause.
Example: $(1\times x=x)$, $(p(1\times x)\lor ..) \vdash (p(x)\lor \ldots)$

Factoring: $(L_1\lor L_2\lor A)\vdash\alpha(L_2\lor A)$
Where L_1 and L_2 have same sign and α = unify(L_1,L_2).
Also original clause is deleted.
Example: $(p(1)\lor p(x)\lor \ldots)\vdash(p(1)\lor \ldots)$

Subsumption: $A,B\vdash B$
Where $A = \alpha B$ for some α.
Also clause A is deleted.
Example. $(p(1)\lor q(1,2))$, $(p(x)\lor q(x,y)) \vdash (p(x)\lor q(x,y))$

Notes:
- A and B arbitrary disjunction of literals (clauses)
- L_k arbitrary literal
- e_k represents an arbitrary expression

FIGURE 15-8
Other common inference rules.

Two of them, ***paramodulation*** and ***demodulation,*** are applicable when one of the literals in a clausal form has the predicate name ***equality*** (often written as "="). The third, ***factoring,*** is useful on any single clause to reduce the number of essentially identical literals. The ***subsumption*** rule permits deleting clause that are essentially duplicates or special cases of others.

In ***paramodulation,*** one clause (called the ***modulator***) has an equality relation as one of the literals. A unification is attempted between one of the terms in this equality and a term used as an argument of some predicate in a literal in the other clause. If unification is successful on just these terms, then a new clause can be deduced, consisting of the "or" or the two original clauses (minus the equality literal), with the matching term in the second of the original clauses replaced by the term on the other side of the "=." In a sense, this rule permits substitution of one expression by another whose value has been declared to be equal to it.

Demodulation is similar to paramodulation in that it uses a clause containing an equality predicate, called the ***demodulator.*** In this case, however, the clause is a unit clause, and a restricted form of unification is applied between one term in the equality and a term in the other clause. The unification succeeds only if no variables in the nonequality term need to be replaced. As with paramodulation, upon a successful match, the other term from the equality is substituted for the matching term in the other clause.

The purpose of this inference rule is to "standardize" terminology. For example, consider the demodulator clause **sister(mother(**x**))=aunt(**x**)**. Applying this to any term in any other clause will replace the **sister(mother(. . .))** combination by the single function **aunt.** Because of this standardization, demodulation often has the side effect of deleting the original nondemodulator clause from the set of clauses and replacing it with the deduced one.

As another example, assume that we have as an argument to some literal the term $y\times$**E** where y is a variable and **E** is some other functional expression (e.g., **L(**$y\times$**E)**). Assume also that some prior unification had bound a 1 to y. If there is a modulator of the form "$1\times x=x$," paramodulating this with the literal containing $[1/y]y\times$**E** would then replace the $y\times$**E** by simply **E** [yielding **L(E)**]. In a sense we have deduced a "special case" of the original clause.

Factoring "simplifies" a clause by eliminating literals that are essentially equivalent to other literals in the clause. The unification test establishes this equivalency.

Subsumption permits deletion of a clause if there exists some substitution that can derive it from some other clause. This permits elimination of easily derived special cases from a program, thus shortening it and probably improving its efficiency.

15.4 REFUTATION COMPLETENESS

One obvious question that one can ask about all these inference rules deals with their relative computational "power." Are there theoretical reasons why some of them should be considered over others?

The answer is yes, with the key distinguishing feature bound up in the unsatisfiability problem described earlier in Section 14.5. Mathematics guarantees that inference engines exist that can show a set of wffs is in fact unsatisfiable, but the guarantee does not extend to the use of any arbitrary set of inference rules. For example, consider the two clauses

$$\mathbf{p}(1)\lor\mathbf{p}(2) \qquad \text{and} \qquad \neg\,\mathbf{p}(x)\lor\neg\,\mathbf{p}(y)$$

Clearly, these two clauses represent a contradiction. The first says that the predicate **p** is true for argument values 1 and 2. The second says (redundantly) that **p** is false for any argument value. Unsatisfiability is shown by observing the substitution $[1/x,2/y]$, and looking for a truth assignment to the ground literals **p**(1) and **p**(2). None of the four possible assignments makes both clauses true.

Now consider all the possible inferences that one could make using just the inference rule ***binary resolution.*** Given the two initial clauses, there are four possible resolvants:

$$(\mathbf{p}(1)\lor\neg\,\mathbf{p}(y)) \qquad (\mathbf{p}(1)\lor\neg\,\mathbf{p}(x)) \qquad (\mathbf{p}(2)\lor\neg\,\mathbf{p}(y)) \qquad (\mathbf{p}(2)\lor\neg\,\mathbf{p}(x))$$

All of these still have two literals. In fact, the resolution of any pair of these clauses yields yet another clause with two literals. We n̆ever seem to make progress.

The problem, of course, is that we never make obvious simplifications, such as replacing $\mathbf{p}(1) \vee \mathbf{p}(1)$ by $\mathbf{p}(1)$, or $(\neg \mathbf{p}(x) \vee \neg \mathbf{p}(y))$ by $\neg \mathbf{p}(w)$. If, however, we include the factoring rule (Figure 15-8), then it is possible to develop the proof sequence of Figure 15-9.

Together, binary resolution and factoring are ***refutation complete***; that is, if a set of wffs is unsatisfiable, then some sequence of inferences using just these two rules will find a contradiction (i.e., ''refute'' the assumption that the original set is all true).

Other inference rules that are refutation complete by themselves include hyperresolution, paramodulation, and unit resolution when the clauses are all Horn clauses.

Just having a set of inference rules with this property guarantees only that an inference engine can be built; it says nothing about the other half of the engine—the decision procedure. In fact, just as with inference rules, there are decision procedures that are not sufficient. Basically, such procedures never try the (possibly unique) sequence of inferences that lead to the contradiction. The next section addresses such issues.

15.5 DECISION PROCEDURES

(Wos et al., 1984, pp. 168–173; Chang and Lee, 1973, chaps. 6, 7, 8)

In analogy with a conventional imperative computer program, the inference rules are like the instructions in a program that actually do computation (''add,'' ''multiply,''...), and the ***decision procedure*** is like the rest of the program (''loop,'' ''procedure call,'' ''compare,'' ''I/O,''...), which stitches the basic computational steps together into a controlled solution of the problem at hand. In logic programming this procedure does not itself do computation as much as it ''navigates'' through all the possible combinations of wff sets and inference rules to select those that might make progress toward the desired answer.

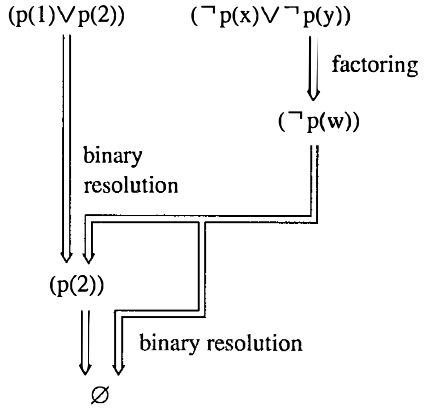
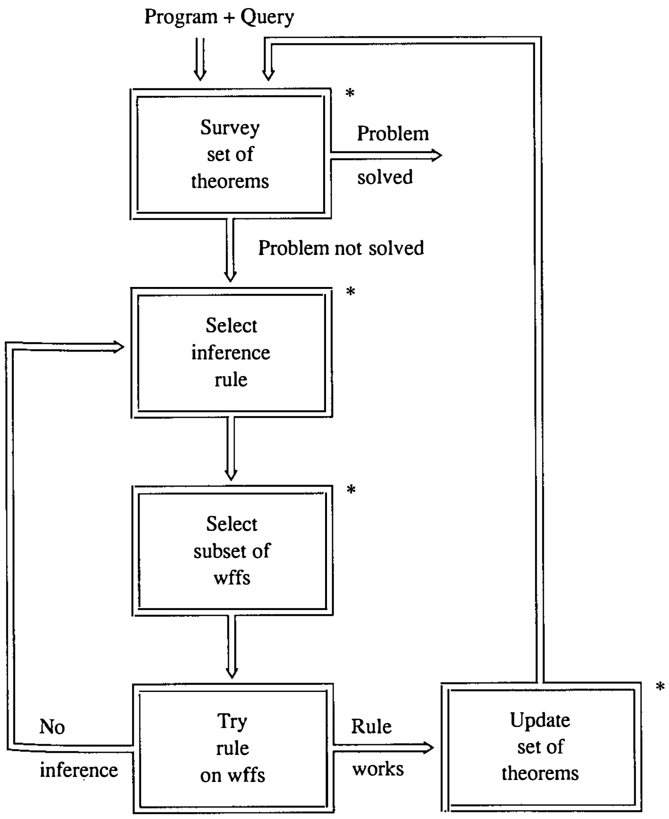


FIGURE 15-9
A refutation sequence.

Figure 15-10 diagrams the basic flow of a decision procedure.

As with conventional programs, a ''good'' decision procedure is one that minimizes the number of ''computations'' (inference rules tried). A ''bad'' decision procedure is one that never tries the right combinations, pursues sequences of inferences which do not contribute to the final answer, or does redundant computations. This means that more work might be required by a bad decision procedure to reach the same solution found by a more efficient one. However, unlike conventional programming, given one ''good'' decision procedure, it is usually not necessary to write completely new ones from scratch for each possible problem that one might see. These procedures are fairly generic, and ca-



* = Parts of decision procedure.

FIGURE 15-10
A generic inference engine.

pable of solving a wide variety of related problems without modifications, although exact solution efficiencies might vary.

This section discusses the most common approaches to the design of decision procedures. For simplicity, many examples will be given in terms of propositional calculus, with discussions on the (usually direct) extensions to predicate logics. More detailed examples will be addressed in later chapters, where real logic computing systems are reviewed.

15.5.1 Recognizing the Answer—Proof by Contradiction

A typical conventional computer program has no knowledge of what it is trying to compute; it stops when it encounters an instruction that tells it to. In contrast, most decision procedures accept explicitly as input some sort of goal which must be reached. Often the goal is stated as a wff for which a proof sequence is desired. Other times it is a condition to be satisfied by the set of inferred wffs.

In either case, the decision procedure must periodically monitor the state of the computation for possible completion of execution. This usually occurs after each successful inference, and can take at least two forms. First, the decision procedure could compare the new wff to the goal wff, or see if addition of the new wff satisfies the termination conditions. This can be quite time-consuming, and may require use of other inference rules such as subsumption to detect cases where the new wff just inferred includes the goal as a special case.

The second approach is somewhat simpler to implement but more complex to understand. It involves initially converting all goals into forms in which the inference of some specific and easily identified wff can signal completion. The most typical approach of this kind is based on the concept of ***proof by contradiction*** or ***proof by refutation.*** Here we use the second form of the Deduction Theorem by assuming that the negation of the initial goal is true and see if from this we can infer a contradiction.

Assuming that a goal is false usually consists of creating a wff which is the negation of the original wff, and adding it to the assumed set of axioms. From there this negated wff is used like any other axiom to create inferences. Now, if the original goal really was a theorem (a proof sequence actually exists), and the original set of axioms was ***consistent*** (no contradiction could be derived from them), then adding the goal's negation creates an ***unsatisfiable*** system. As described previously, testing a set of wffs for unsatisfiability is a decidable problem, and solvable via properly designed inference engines in finite time.

For wffs in clausal form, and resolution-based inference rules, the test for unsatisfiability is equivalent to looking for the ***empty clause*** ("ø"), the clause with no literals.

Note that in any case a positive result from such a test does not say which axiom is at fault, only that, together, the set used in the detection of the contradiction has a problem. However, if the original axioms were consistent (and this is up to the programmer to guarantee), then the inescapable conclusion is that the wff added at the beginning of the decision procedure's execution is false. But this wff was the negation of the desired goal, which in turn means that the goal must be true—exactly what we were after.

Another important observation about this procedure is that if care is not taken in the choice of the order in which inference rules are applied, and the wff sets they use, then it may be quite difficult to reconstruct a proof sequence for the original goal from the sequence that inferred unsatisfiability. The reader should look for how real logic languages avoid this when they are discussed in later chapters.

Independent of this problem, we must note that the unifying substitution information gathered by the inference sequence is valid regardless of the order of the inferences or the mechanism used to terminate the computation. Consequently, if what we want is a set of values for the free variables in the original goal, then the proof-by-contradiction approach is useful and will yield the desired results.

Figure 15-11 diagrams a very simple proof by contradiction using the resolution inference rule. The sequence shown was chosen to illustrate the prior point about loss of the proof sequence for the original goal. At no point is there derivation of the original goal wff **is-parent-of**(Peter, Tim).... The reader should contrast this to an alternative sequence of the form **A, B, D, C**⊢ø with the substitution [Peter/*w*]. In this form **D** essentially represents **C,** the original goal, but with the specified substitution for *w*. This is the desired answer.

Initial Axioms: (Each consists of one clause)
A: is-father-of(Peter,Tim)
B: is-father-of(x,y)⇒is-parent-of(x,y)
= (¬is-father-of(x,y) ∨ is-parent-of(x,y))

Initial Goal: is-parent-of(w,Tim)

Steps:

1. Add ¬is-parent-of(w,Tim) as clause C
2. Resolve B and C:
   unify(is-parent-of(x,y)), is-parent-of(w,Tim)) = [w/x,Tim/y]
   Result: D = [w/x,Tim/y](¬is-father-of(x,y))
   = ¬is-father-of(w,Tim)
3. Resolve D and A:
   unify(is-father-of(w,Tim), is-father-of(Peter,Tim))
   = [Peter/w]
   Result: [Peter/w]∅ = ∅

Final Proof Sequence: B,C,D,A,∅,
with substitutions [Peter/w][w/x,Tim/y]

FIGURE 15-11
A simple proof by contradiction.

15.5.2 Generic Decision Procedures

In general there are two major forms of decision procedures in use today: ***forward chained*** and ***backward chained.*** Roughly speaking, these forms correspond to the two forms of the Deduction Theorem. A forward-chained engine typically expands the initial axiom set outward, beating wffs against each other, using modus ponens-like rules, and recording cases where new wffs have been discovered to be consistent with prior ones (theorems). When this envelope of theorems intersects the goal, the procedure stops.

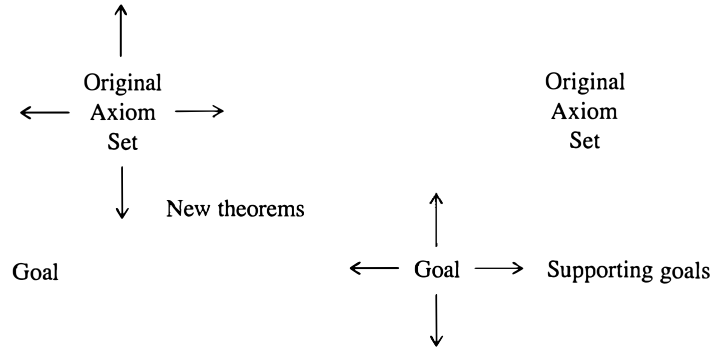
Conversely, a backward-chained engine starts with a particular wff to be proved a theorem, and works recursively backward from it to show that if other, simpler wffs are true, then so is this one. When each of these wffs has been found to be part of the original axiom set, the procedure stops. Very often this process involves assuming that the negation of the goal is valid and proving a contradiction.

Figure 15-12 diagrams simplistic views of how each approach expands through the universe of wffs.

To get a better feeling for what these decision procedures look like, consider a simple subset of propositional calculus in which all wffs are either a fact of the form ⟨letter⟩ or a rule of the form (⟨letter⟩ ⇒ ⟨letter⟩). Assume also that we limit ourselves to modus ponens as an inference rule.

The resulting inference engines might look like those shown in Figure 15-13. These programs accept two inputs, the goal wff (or wff set, all of which must be shown to be true) and the set of initial axioms. Using auxiliary functions of the form **pick-xxx,** the programs select wffs, test them for possible use as inferences, and then expand the appropriate set.

As expressed here, these programs are at best inefficient and at worst susceptible to infinite loops. There is no information given the **pick-xxx** functions to help them select the most appropriate set of wffs to try inferencing next, or even to avoid choices already made. The next sections will discuss some general strategies that might be employed, and combinations of techniques that have been used in real inference engines.



(*a*) Forward chained. (*b*) Backward chained.

FIGURE 15-12
Today's inference engines.

```
prove-forward(goal, axioms) =
  let fact = pick-next-fact(axioms)
  and rule = pick-next-rule(axioms) in
      if fact = antecedent(rule)
      then if goal = consequent(rule) then T
           else let new-axioms = append(consequent(rule),axioms) in
                prove-forward(goal,new-axioms)
      else prove-forward(goal, axioms)
```

(*a*) Forward chained.

```
prove-backwards(goal-set, axioms) =
  if null(goal-set)
  then T
  else let goal = pick-next-fact(goal-set) in
       if goal∈axioms
       then prove-backwards(delete(fact,goal-set), axioms)
       else let rule = pick-next-rule(axioms) in
            if consequent(rule) = goal
            then let new-goals = append(antecedent(rule),
                                        delete(goal,goal-set)) in
                 if prove-backwards(new-goals, axioms)
                 then T
                 else prove-backwards(goal-set, axioms)
            else prove-backwards(goal-set, axioms)
```

(*b*) Backwards chained.

```
Assume: <axiom> := <fact> | <rule>
        <fact> := <letter>
        <rule> := <antecedent>⇒<consequent>
        <antecedent> := <letter>
        <consequent> := <letter>
```

FIGURE 15-13
Skeletal decision procedure.

15.5.3 Resolution Permutations

Generating all possible wffs as input to an inference rule is not enough to guarantee that all possible inferences will be made. It is also possible for the same two clauses to be resolved together using the same inference rule in many different ways. For example, consider the following clauses, where **p, s,** and **t** are predicates, **A** and **B** are constants, and *x, y,* and *z* are variables:

$$\mathbf{p}(\mathbf{A})\vee\mathbf{p}(\mathbf{f}(z))\vee\neg\,\mathbf{s}(x)\vee\mathbf{t}(\mathbf{B},x))$$
$$(\neg\,\mathbf{p}(y)\vee\mathbf{s}(y))$$

Using binary resolution, there are two ways of resolving the second clause with the first, based on which literal involving the **p** predicate is used. These yield, respectively,

$$(\mathbf{s}(\mathbf{A})\vee\mathbf{p}(\mathbf{f}(z))\vee\neg\,\mathbf{s}(x)\vee\mathbf{t}(\mathbf{B}))$$

or

$$(\mathbf{p}(\mathbf{A})\vee\neg\,\mathbf{s}(x)\vee\mathbf{t}(\mathbf{B})\vee\mathbf{s}(\mathbf{f}(z))$$

In addition, one could also choose first to resolve the **s**'s, yielding

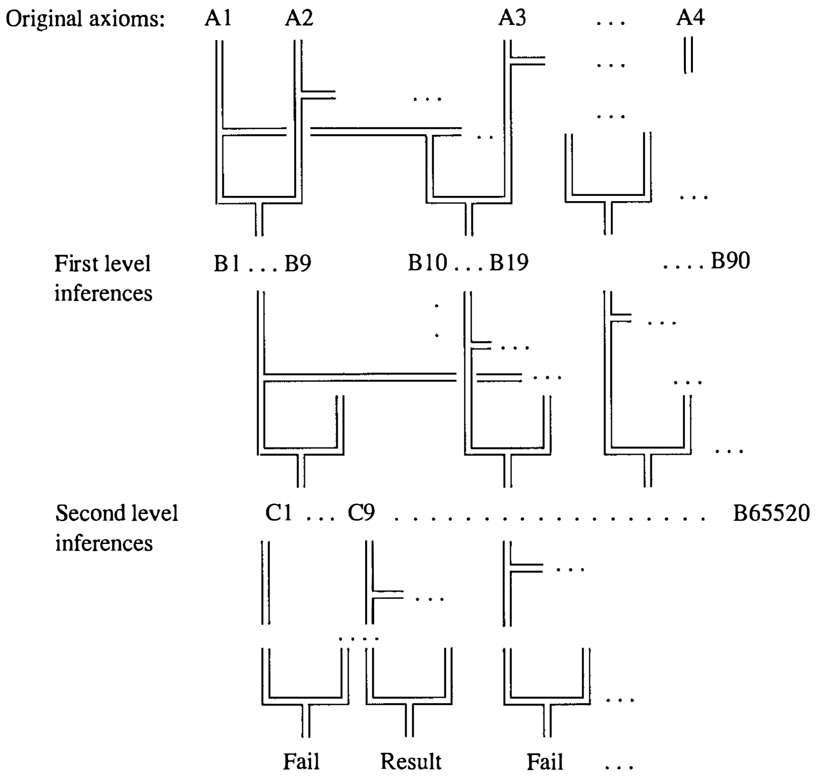
$$(\mathbf{p}(\mathbf{A})\vee\mathbf{p}(\mathbf{f}(z))\vee\mathbf{t}(\mathbf{B},x)\vee\neg\,\mathbf{p}(x))$$

In general, for two clauses of m and n literals, respectively, there are m×n possible resolutions to be tried, each potentially yielding a different resolvant. Although some of these may be redundant, or simplifiable (using the auxiliary inference rules discussed earlier), in general they exhibit significant differences, with only secondary indications of which ones might be most appropriate to pursue in later inferences. Consequently, a good decision procedure cannot overlook any of the possibilities, and must be capable of trying all of them at some point if the first few resolutions do not lead to the desired proof sequence.

15.6 STRATEGIES—AN OVERVIEW

The "decision procedures" developed by Gilmore et al. (as discussed in Chapter 14) were based on proving the unsatisfiability of a set of wffs by an ordered trial-and-error search through an infinite set of possible interpretations. The explosive growth in computation for this approach as problem size grew led many researchers in the 1960s to investigate alternatives. Inference rule-based engines as described above (particularly ones using resolution) were a tremendous first step, but by themselves still suffered from computational inefficiencies. Basically, to guarantee that a solution will be found if one exists, the decision procedure must be capable of trying all possible inferences, both in terms of picking pairs of possible parent clauses and in terms of picking which literals within those clauses to use in actually attempting unification. The generic procedures of the last section had this property if the **pick-xxx** functions were designed properly.

The possible combinations of wffs are often presented pictorially as a ***search tree*** (Figure 15-14), where the nodes at any level represent a possible inference. To get a feel for the potential growth in such a tree, consider a problem with an initial set of four clauses of three literals each, and using an inference engine incorporating the binary resolution rule.



Breadth first: Try B1, B2, . . Bn, then C1, C2, . . .

Depth first: Try B1, C1, D1, . . X1, then X2, . . .
then W2, X3, . . .

FIGURE 15-14
Search tree.

Just among those four clauses there are 10 possible pairs (remember that a clause may resolve with itself), each of which has nine possible ways of selecting literals to **unify.** Thus there are 90 possible resolvants, each of which must be checked for termination and possibly employed in the next iteration. If all 90 resolvants were generated, the next level of resolutions could approach 65,000+ combinations, with further levels rapidly exploding in size. (Remember that each of the first-level resolvants may have up to four literals each).

From this example it is obvious that brute force will solve only the simplest problems in any reasonable period of time. The decision procedure employed in a realistic inference engine must be more sophisticated and employ a variety of strategies to focus the search in the most productive directions. These strategies can be divided into three major categories as follows:

- *Ordering strategies*—strategies within the decision procedure that specify which clauses (or sets of wffs in nonresolution systems) should be tried against the inference rules first and which should be deferred until later
- *Restriction strategies*—strategies that prevent certain combinations from ever being tried
- *Simplification strategies*—strategies that check to see if newly inferred wffs are simply generalizations or special cases of existing wffs, and if so, which one should be dropped

The selection of any strategy, or combination of strategies, must be done with care, for it is possible to quickly lose completeness of the inference engine, that is, to lose guarantees that the engine will find a solution if one exists.

The following sections discuss each category of strategy in more detail.

15.6.1 Ordering Strategies

An ***ordering strategy*** guides the order in which various combinations of wffs are beat against the available inference rules. This involves defining some criterion on expectations for success, applying it to the various possible combinations, and sorting the results. The combinations with the most favorable rating should be tried first, and those with less favorable combinations tried only when these first ones have failed.

The criterion using for the ordering can be either provided by the user or built into the system. The user, for example, may want the system to try wffs reflecting "rules of thumb" or "typical cases" before using wffs that go back to basic principles. Alternatively, the user may know from experience that certain predicates are easier to prove or disprove than others, and so should be attempted first. Both of these heuristics could be employed by permitting weighting factors to be assigned by the user before the computation begins, and providing in the system some mechanism for propagating the factors to new inferences as they are developed.

Many real logic systems provide built-in criteria that have proved useful across a broad range of problems. For example, one could constrain the next inference to include the last wff inferred as one parent, if there are still possibilities untried for it. If there are no possibilities, the system ***backtrack***s to the wff inferred before that, and picks up from whatever combinations were still possible for it. This approach is termed a ***depth-first search,*** since it dives as far down one possible trail of inferences from a single wff as it can before trying another. If there are a lot of possible answers buried deep in the tree, this approach is usually desirable. However, if there are solutions close to the top, but in paths not connected to the first ones tried, the system may spend a great deal of effort on essentially wasted computation and overlook answers that are very close.

An ordering criterion that is more suitable for such "close-in" answers is ***breadth-first search.*** Here we work through one level of the tree at a time, picking wffs in succession and holding each as one input to an inference rule while all other wffs are tried as the other inputs. Any wffs inferred during this process are marked as "unsearched" and added to the next level of the tree. Only when all wffs at one level are tried do we move to the next level.

In general this approach will find close-in answers long before depth-first searches will, but it can take an exponentially longer period of time to solve problems when the answers are deep because it does generate the whole tree.

Another difference between these two search techniques is the amount of storage needed to save information on prior wffs. A depth-first approach tends to need something proportional to the maximum depth of the tree, while breadth-first tends toward something proportional to the total number of wffs inferred. Again, depending on the problem, either one of these can mushroom while the other uses a minimal amount of storage.

Combinations of these two approaches often make a lot of sense. For example, in resolution-based systems where we are looking for the inference of the empty clause, resolvants where the number of literals shrinks over that of the parents should be preferred over cases where the number of literals grow. This leads to the ***unit preference strategy,*** where clauses are sorted according to the number of literals in them, and combinations with the smallest number of literals are tried first. The result is that if any solutions are just "one inference away" (i.e., an empty clause resulting from the combination of unit clauses), they will be tried first. This is essentially breadth-first. On the other hand, if there are no obvious immediate possibilities, this strategy will keep plugging at one deep trail of inferences as long as the number of literals continues to decrease.

15.6.2 Restriction Strategies

One way to cut down on a tremendous amount of processing is simply to avoid trying certain combinations of wffs altogether. ***Restriction strategies*** do this; they identify to the rest of the decision procedure search paths that should not be pursued. Obviously, this must be done with care, or the system may fail to find solutions that are really there. In many cases, however, this may be acceptable; there may be several answers although the user needs only one. Alternatively, the user may know that a solution exists but requires going back to certain basic axioms and instigating extensive inferencing to derive it. He or she may wish to avoid this solution and see if there is some faster solution based on special cases or more "heuristic" axioms.

Perhaps the most direct restriction strategy puts an upper limit on the criterion used in an ordering strategy. If a particular wff or wff combination exceeds this limit, it is never used in an attempted inference. The simplest such technique is ***bounded depth first,*** which prevents attempting to use the most recently generated wff if there is some wff that is *d* or more levels farther up the tree and has not been explored yet. This may suffer some of the same problems as breadth first, but it has the distinct advantage of easy implementation and clean integration with an ordering strategy.

A different approach, termed the ***set of support strategy,*** can, if used correctly, both exclude certain inferences and still guarantee completeness. Basically, the original set of axioms is split into two pieces, one of which is termed the ***set of support axioms.*** Now, whenever a possible inference is encountered, it is skipped unless at least one parent wff either is itself a member of the set of support or has a proof sequence with such a member in it. As the name implies, all wffs inferred with such a strategy are "supported" at some point by axioms from the designated set.

Although the choice of wffs for a set of support is arbitrary, for refutation-based inference engines one choice is particularly meaningful. In such a system the original goal clause has been negated and added to the rest of the axioms, and a test for unsatisfiability of the set has been made. If we assume that the rest of the axiom set is itself satisfiable, there is no reason to derive inferences using strictly wffs from them. One would be simply adding more satisfiable wffs to an already satisfiable set, and the empty clause would never appear. If, on the other hand, we make the negated goal clauses the set of support, then such "redundant" computations are avoided, and if we reach an empty clause, it is because of the original goal. In fact, there is a theorem (the ***Set of Support Theorem***) which states that if the inference engine is complete without the support restriction, then the addition of this restriction preserves refutation completeness. We have both saved computation and kept guarantees about finding solutions.

15.6.3 Simplification Strategies

Simplification strategies look at a wff deduced by an inference rule and try to either convert it into some standard form, simplify it, or eliminate it outright if it duplicates an existing wff. Performing such operations may be built into the basic inference engine or performed by an auxiliary inference engine (see Figure 15-15). Such an engine very often has its own set of inference rules (e.g., paramodulation, demodulation, factoring, and subsumption) and axioms which are very specific to the kind of problem being solved. For example, a system involving mathematical formulas may have an entire set of rules for simplifying arithmetic expressions, while one dealing with genealogy may wish to replace expressions such as **brother(parent(*x*))** by **uncle(*x*)**.

While such approaches can greatly reduce the number of inferred wffs, extreme care must be taken that the computational cost of doing the simplifications does not overwhelm the cost of performing some extra inferences, particularly when there are good ordering or restriction strategies that minimize useless inference attempts.

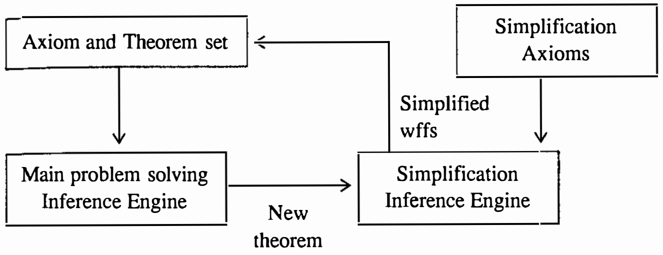

Axiom and Theorem set
Simplification Axioms
Simplified wffs
Main problem solving Inference Engine
New theorem
Simplification Inference Engine

FIGURE 15-15
Auxiliary inference engine for simplification.

15.7 EXAMPLE

Figure 15-16 gives one possible resolution-based solution to the picnic problem described in Section 13.5. All the individual wffs have been converted into clausal form, the goal **G** negated, and all clauses **and**-ed together into a single collection. Binary resolutions are performed using a

Axioms: $(A \Rightarrow B), (B \wedge C \Rightarrow D), (D \wedge (E \vee f) \Rightarrow G), F, A, C, \neg E$

Goal: G

Conversion to clausal form:

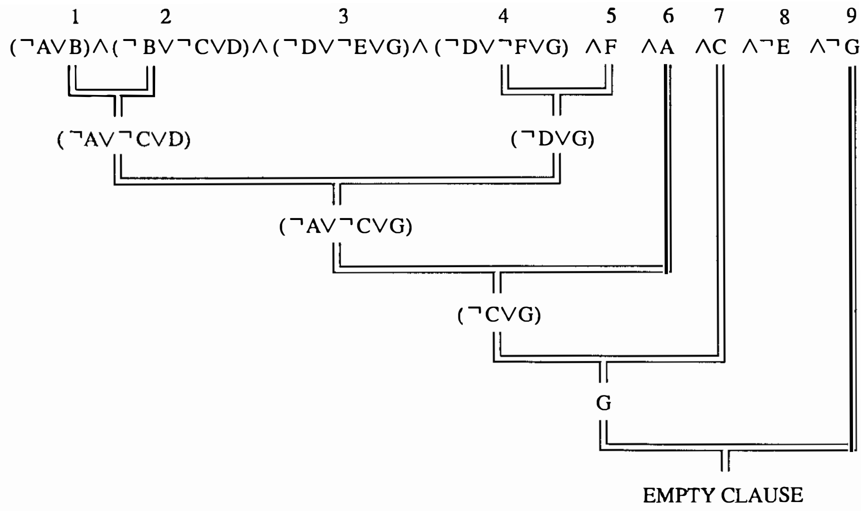


FIGURE 15-16
A resolution to the picnic problem.

basically depth-first decision procedure until the empty clause has been inferred. At this point a contradiction has been found, which can be true only if the clause ¬**G** was false, and thus **G** is true.

To show the actual computational effects of various strategies, Figure 15-17 gives the combinations of wffs that would be tried for three different decision procedures:

- Breadth first, where newly inferred clauses are added to the end of the list of axioms
- Depth first, where again newly inferred clauses are added to the end of the list of axioms
- A combination of unit preference and set of support in which we start with the negated goal, always use the most recent resolvant as one parent, and use as the other parent of each attempted resolution the clause of shortest length that includes one of the literals of the first parent in it.

As can be seen, for this example there is a significant difference in performance. The breadth-first approach will take an enormous amount of time to discover the contradiction. The depth-first approach is better, making significantly more early resolutions than breadth-first, but still going down some early blind alleys.

In contrast, the combined unit preference and set of support zeros in on a solution with a minimum of steps.

The reader should be cautioned that this is not always true. It is easy to devise problems for which just the opposite happens, and a breadth-first or depth-first approach wins.

15.8 DEDUCTION TREES

From time to time it will be useful to represent pictorially the sequence of inferences produced by an inference engine for some problem. A ***deduction tree*** (also called a ***proof tree***) is one such representation that is particularly useful for Horn clauses and backward-chained inference engines. Figure 15-18 diagrams how such a diagram is drawn. Clauses entered as program statements are represented as subtrees in which the single positive literal is at the root and each negative literal is a child. Goal wffs which are to be proved true as theorems are also drawn as subtrees, but with no explicit roots. A resolution between a goal literal and the positive literal of a clause is represented by joining the root of the subtree corresponding to the clause to the bottom of the equivalent goal literal. The negative literals of the clause become new nodes to be matched to other positive literals.

Note that when a fact, as a positive literal, resolves with one of these negative literals, it essentially terminates that chain.

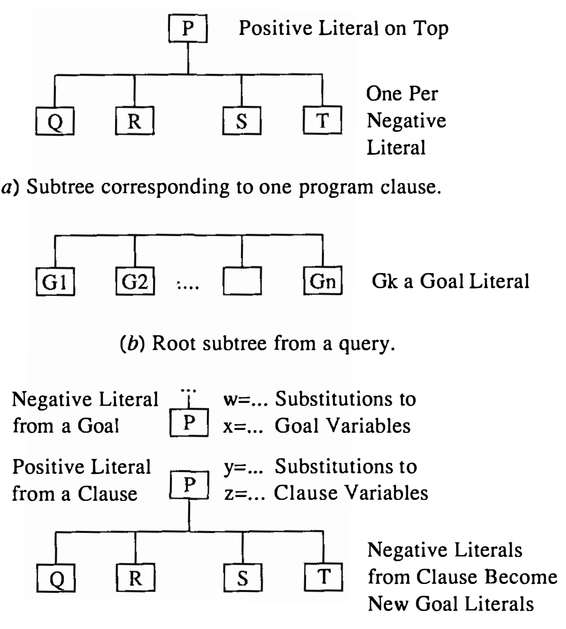
Figure 15-19 diagrams a deduction tree that represents the solution

	Breadth First	Depth First	Unit Preference Set of Support
1.	$1-1$	$1-1$	$9-4 \Rightarrow 10=(\neg D \vee \neg F)$
2.	$1-2 \Rightarrow 10=(\neg A \vee \neg C \vee D)$	$1-2 \Rightarrow 10=(\neg A \vee \neg C \vee D)$	$10-5 \Rightarrow 11=\neg D$
3.	$1-3$	$10-1$	$11-2 \Rightarrow 12=(\neg B \vee \neg C)$
4.	$1-4$	$10-2$	$12-7 \Rightarrow 13=\neg B$
5.	$1-5$	$10-3 \Rightarrow 11=(\neg A \vee \neg C \vee \neg E \vee G)$	$13-1 \Rightarrow 14=\neg A$
6.	$1-6 \Rightarrow 11=\neg B$	$11-1$	$14-6 \Rightarrow$ nil*****
7.	$1-7$	$11-2$	Solution
8.	$1-8$	$11-3$	Found!!
9.	$1-9$	$11-4$	
10.	$1-10$	$11-5$	
11.	$1-11$	$11-6 \Rightarrow 12=(\neg C \vee \neg E \vee G)$	
12.	$2-2$	$12-1$	
13.	$2-3 \Rightarrow 12=(\neg B \vee \neg C \vee \neg E \vee G)$	$12-2$	
14.	$2-4 \Rightarrow 13=(\neg B \vee \neg C \vee \neg F \vee G)$	$12-3$	
15.	$2-5$	$12-4$	
16.	$2-6$	$12-5$	
17.	$2-7 \Rightarrow 14=(\neg B \vee D)$	$12-6$	
18.	$2-8$	→ $12-7 \Rightarrow 13=(\neg E \vee G)$	
19.	$2-9$	$13-1$	
20.	$2-10$	$13-2$	
21.	$2-11$	$13-3$	
22.	$2-12$	$13-4$	
23.	$2-13$	$13-5$	
24.	$2-14$	$13-6$	
25.	$3-3$	$13-7$	
26.	$3-4$	$13-8$	
27.	$3-5$	$13-9$	
28.	$3-6$	$13-10$	
29.	$3-7$	$13-11$	
30.	$3-8$	$13-12$	
31.	$3-9 \Rightarrow 15=(\neg D \vee \neg E)$	→ $13-13 \Rightarrow 14=\neg E$	
32.	$3-10$	$14-1$	
33.	$3-11$	$14-2$	
34.	$3-12$	$14-3$	
35.	$3-13$	$14-4$	
36.	$3-14$	$14-5$	
37.	$3-15$	$14-6$	
38.	$4-1$	$14-7$	
39.	$4-2 \Rightarrow 16=(\neg B \vee \neg C$	$14-8$	
40.	$4-3$ $\vee \neg F \vee G)$	$14-9$	
41.	$4-4$	$14-10$	
42.	$4-5 \Rightarrow 17=(\neg D \vee G)$	$14-11$	
43.	$4-6$	$14-12$	
44.	$4-7$	$14-13$	
45.	$4-8$	$14-14$ * Backtrack	
46.	$4-9 \Rightarrow 18=(\neg D \vee \neg F)$	$13-14$ * Backtrack	
47.	$4-10$	$12-8$	
48.	$4-11$	$12-9 \Rightarrow 15=(\neg C \vee \neg E)$	
49.	$4-12$	$15-1$	
	....	....	

Notes:
m − n means resolve
clause m with clause n
⇒ m means infer new
clause m

FIGURE 15-17
Effects of different decision procedures.

Horn Clause: $((Q \wedge R \wedge S \wedge T) \Rightarrow P) = (\neg Q \vee \neg R \vee \neg S \vee \neg T \vee P)$



(*a*) Subtree corresponding to one program clause.

(*b*) Root subtree from a query.

(*c*) A Resolution.

FIGURE 15-18
A deduction tree.

to the picnic problem from Section 13.5. The program consists of eight Horn clauses, and the goal is the single clause **G**. The number next to each resolution indicates which clause was used.

Finally, we can also show the required unification substitutions on such a diagram. Figure 15-20 diagrams a simple version of the grandparent problem discussed earlier. The required substitutions are shown next to each resolution. This allows a careful tracing of how each variable gets its final binding.

15.9 PROBLEMS

1. Indicate whether or not the following pairs of literals unify, and if so, what is the unifying substitution. (Do not rename any variables—assume they are as labeled.)
   a. p(Roy,Tim), q(Roy,Tim)
   b. p(Roy,*x*), p(*y*,Tim)
   c. p(*x*,*x*), p(1,1)
   d. p(*x*,*x*), p(1,2)
   e. p(*x*,*x*), p(1,*y*)
   f. p(f(*x*,*y*)), p(*z*)
   g. p(f(*x*,*y*),3),p(f(*z*,*z*),h(3))
   h. p(f(*x*,*y*),*x*), p(*z*,*z*)

1 2 3 4 5 6 7 8
Program Wff: $(\neg A \vee B) \wedge (\neg B \vee \neg C \vee D) \wedge (\neg D \vee \neg E \vee G) \wedge (\neg D \vee \neg F \vee G) \wedge F \wedge A \wedge C \wedge \neg E$
Goal Wff: G

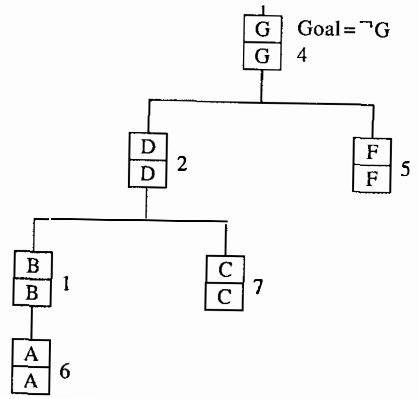


FIGURE 15-19
Deduction tree for the picnic problem.

2. Find the most general unifier for the three following literals:

   p(*x*,*x*,1) p(*y*,*w*,*y*) p(*z*,*q*,*r*)
3. After how many steps do the breadth-first and depth-first strategies for Figure 15-17 find a solution? How many inferences did they make?
4. Modify the unify routines of Figure 15-2 to detect when an occurs check happens, and return the value "error."
5. Describe the kinds of modifications that one would have to add to the unification process to account for quantifiers over functions or predicates.

(¬p(x,z)∨¬p(z,y)∨gp(x,y))
(¬f(x,y)∨p(x,y))
f(Roy,Pete)
f(Pete,Tim)

Goal: gp(x,Tim)

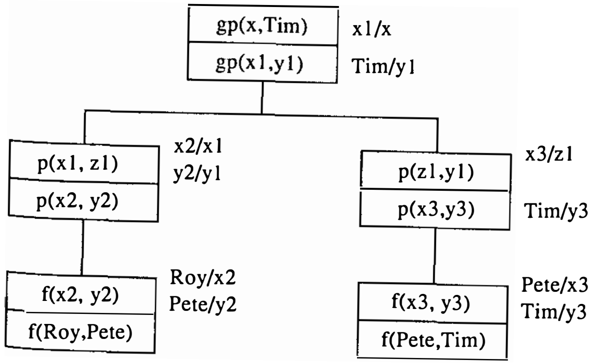


FIGURE 15-20
Deduction tree for grandparents problem.

6. Prove that the following set of clauses is unsatisfiable using resolution. Show the results of each unification.

   (1) $\mathbf{A}(\mathbf{f}(x,y),x,y)$

   (2) $\neg\mathbf{A}(z,\mathbf{g}(z,1),1)$

   (3) $\neg\mathbf{A}(2,w,u)\vee\mathbf{A}(w,v,3)\vee\neg\mathbf{A}(2,3,t)\vee\mathbf{A}(u,v,t)$

   (4) $\neg\mathbf{A}(\mathbf{h}(m),m,\mathbf{h}(m))$

7. Redo Figure 15-16:

   a. Using a breadth-first strategy and showing all wffs generated at each level

   b. Using breadth-first plus set-of-support

8. Show a deduction tree for proving $\neg\mathbf{F}\vee\mathbf{G}$ for the picnic problems.

9. Assuming the following clauses:

   (1) $\mathbf{m}(0,y,0)$

   (2) $(\mathbf{m}(\mathbf{s}(x),y,z)\vee\neg\mathbf{m}(x,y,z1)\vee\neg\mathbf{a}(y,z1,z))$

   (3) $\mathbf{a}(0,y,y,)$

   (4) $\mathbf{a}(\mathbf{s}(0),y,\mathbf{s}(y,))$

   (5) $\mathbf{a}(\mathbf{s}(x),y,\mathbf{s}(z))\vee\neg\mathbf{a}(x,y,z))$

   Use linear resolution to find proofs for the following (show deduction trees):

   a. $\mathbf{a}(0,\mathbf{s}(0),x)$

   b. $\mathbf{m}(\mathbf{s}(0),\mathbf{s}(\mathbf{s}(0)),x)$

   c. $\mathbf{a}(w,w,x) \wedge \mathbf{m}(w,w,x)$

CHAPTER 16

THE PROLOG INFERENCE ENGINE

Previous chapters have discussed the basic theory behind logic-based computing: the formal expression of symbolic statements in a predicate logic framework, inference rules to govern what statements can be deduced, and decision procedures to control the attempted inferences. This and the following chapters show how this theory can and has been put into practice for a class of logic-based computing systems (languages and machines) with the following characteristics:

- First-order predicate logic
- Horn clause form for individual statements
- Largely backward-chained inference engine using the second form of the Deduction Theorem
- Use of resolution (without simplification) as the inference rule
- A linear input set of support strategy as the decision procedure
  A computer model not much different from the classical von Neumann one

The most successful and well known such language is PROLOG, a language second only to LISP in its popularity in the artificial intelligence and symbolic computing community. The language was invented by Alain Colmerauer (1985) and his associates in Marseilles, France, around 1970, and stands for "PROgramming in LOGic" (see also Cohen, 1985). Its theoretical foundation came from Horn clauses and resolution, but it quickly grew to include "extralogical" features that made it a complete programming language with more efficiency and programmability than just a pure logic theorem prover. A testament to its versatility can be seen in the explosion of new programming languages, and extensions to old ones, that are "PROLOG-like."

The rest of this chapter gives a brief introduction to the syntax and semantics of PROLOG. This includes (most importantly) several different ways of viewing the basic PROLOG inference engine, starting with a propositional logic version and building to a more complete predicate logic one. Several simple examples are discussed in detail, including an interpreter of a large set of PROLOG written in PROLOG itself and the Towers of Hanoi problem discussed earlier in functional forms (see Section 3.3.3). However, as mentioned before, this is not a programming text: The major emphasis will be on developing a series of models of PROLOG's inference engine that permit a solid understanding of both implementation techniques and alternative languages that are spin-offs. More extensive detail needed to learn how to write full programs are left to the relevant programming manuals.

For a particularly interesting example of a large PROLOG program, the reader is referred to an article describing the encoding of a piece of legislation, the British Nationality Act (Sergot et al., 1986) as a PROLOG program.

As with LISP, we note that there are several major dialects of "standard" PROLOG in existence, although in this case the primary differences are in superficial syntax. For the most part, this text adopts the syntax described in Clocksin and Mellish's text (1984), with the one exception being the notation for s-expressions. This is to permit notational commonality with other parts of this book, particularly with discussions of LISP.

Finally, despite its popularity the inference engine described here is by no means the only one of significance, particularly to a computer architect or language designer. Later chapters discuss other inference engines suited for either specialized offshoots of PROLOG (particularly to handle parallelism), or for significantly different languages.

16.1 STRUCTURE OF A PROLOG PROGRAM

(Clocksin and Mellish, 1984)

The basic syntax of a PROLOG program (see Figure 16-1) is rather simple and consists of a partially ordered set of ***statements*** or ***clauses*** of three types: ***facts, rules,*** and ***queries,*** where each corresponds to some form of a Horn clause. These are all built from logical combinations of positive ***literals*** or ***atoms*** (syntatically called *⟨goal⟩* in Figure 16-1) normally consisting of predicate symbols and prefixed to argument lists surrounded by "( )." The arguments (called ***terms***) can be relatively arbitrary expressions involving functions, variables, and constants, and basically follow the conventions of first-order predicate logic. Note that terms and goals have virtually identical syntax, although their uses and meaning are obviously different.

```
<symbol> := <char-string>
<constant> := <number> | <symbol>
<variable> := _| <char-string>
<functor> := <symbol>
<term> := <constant> | <variable> | <functor>{(<term> {,<term>}*)}
<goal> := <variable> | <predicate>{(<term> {,<term>}*)}
<head> := <goal>
<body> := <goal>{,<goal>}*
<fact> := <head>.
<rule> := <head>:-<body>.
<query> := ?-<body>.
<program> := {<fact> | <rule>}* <query>
```

FIGURE 16-1
Simplified PROLOG syntax.

Individual statements terminate with a ".".

Not shown in the figure are several kinds of terms (such as s-expressions) and some specialized predicates that do have a "built-in" meaning. These are discussed later, as are some cases where structures use infix functors rather that the prefix form used here.

16.1.1 Basic Symbols

With a few important exceptions, the symbols used for functions, predicates, and constants have absolutely no predefined "meaning" or "interpretation," and are defined strictly by the relations developed by the logic program of which they are part. The only constraint in these meanings is that the symbols mean the same thing wherever they appear in the program.

In contrast, identifiers used as variables in PROLOG are considered to be local to the statement in which they reside. Thus all occurrences of the same identifier name within a single statement refer to the same object(s). However, the same name appearing in two separate statements reflects two separate variables, each unique to its own statement. This is true even when the same statement is used multiple times in a single program execution: Each use gets a "fresh" set of variables. PROLOG's inference engine will acknowledge this by internally renaming all variables in a statement each time it is used.

In conventional Clocksin and Mellish syntax, all function, predicate, and constant symbols must start with lowercase letters, and all variables must start with uppercase letters. We will follow this convention for function and predicate symbols, but will also use symbols starting with lowercase letters for variables, and either leading lower- or uppercase letters for constants. This agrees with both mathematical convention and what we have used earlier in function-based computing. The font style will also delineate their usage (thus x is a variable while, **X** or **x** is the name of some constant). Those cases where possible confusion might result will be flagged.

16.1.2 Fact and Rule Statements

As described by Figure 16-1, the typical PROLOG program consists of a set of fact and rule statements, followed by a single query. Of these a ***fact*** is the simplest, and consists of a single literal (syntatically the same as a ⟨***goal***⟩), usually a predicate symbol with a set of arguments. It corresponds to a Horn clause that is a single positive literal, and as such defines one or more tuples in the relation named by the predicate symbol.

Any variables in the arguments of the clause are universally quantified, meaning that the clause is equivalent to a potentially infinite number of equivalent ground clauses, each with a different combination of constants substituted for the variables. As an example, the following two facts specify that the predicate relation **ancestor** includes the specific tuple **(Roy,Tim)** and the infinite set of tuples whose first element is **Adam:**

$$\mathbf{ancestor(Roy,Tim)}.$$
$$\mathbf{ancestor(Adam,}x\mathbf{)}.$$

In terms of first-order logic, the second statement is equivalent to:

$$\forall x | \mathbf{ancestor(Adam,}x\mathbf{)}$$

The next major statement form, a ***rule,*** has the form

$$\mathbf{a}(\ldots x){:}-\mathbf{b}_1(\ldots),\mathbf{b}_2(\ldots),\ldots,\mathbf{b}_n(\ldots).$$

which is equivalent to a single wff of the form

$$\forall x \ldots | (\mathbf{b}_1(\ldots) \wedge \mathbf{b}_2(\ldots) \wedge \ldots \wedge \mathbf{b}_n(\ldots) \Rightarrow \mathbf{a}(\ldots))$$

or in clausal form as the disjunction

$$\forall x \ldots | (\mathbf{a}(\ldots) \vee \neg \mathbf{b}_1(\ldots) \vee .. \vee \neg \mathbf{b}_n(\ldots))$$

By convention, a rule has two parts, a ***head*** and a ***body***. The head consists of one literal (again syntatically the same as a ⟨***goal***⟩) and represents the consequent of the implication form of the wff, or the single positive literal in the clausal form. The body consists of a series of goals separated by commas and represents the implication form's antecedents, or the unnegated form of the negative literals in the clausal form.

In typical PROLOG terminology the head is the "left-hand side" of the statement and the body is the "right-hand side," with the "," interpreted as an "and" and the ":−" as "→" (but with antecedents and consequent reversed).

The following example defines a rule which says that if someone is the parent of someone else, who in turn is the ancestor of a third person, then the first person is an ancestor of the third:

$$\mathbf{ancestor}(x,y){:}-\mathbf{parent}(x,z),\mathbf{ancestor}(z,y).$$

With the mental conversion between statements and Horn clauses in mind, it is easy to relate a PROLOG program to a large wff which is the "and" of a series of clauses, one per program statement, with all universal quantifiers pulled to the left. Although logically the order of the clauses in this program is immaterial, PROLOG's decision procedure actually attempts to use these statements in the specified order, giving the programmer some sense of "sequential" control over when rules and facts are used by the inference engine.

Figure 16-2 gives the PROLOG form of much of the **grandparent** problem described earlier.

16.1.3 Queries

The final type of statement associated with a PROLOG program (the ***query***) represents the initial question to be answered, the primary "input" to the program, or the clause to be "proven" a theorem. For each

```
parent(x,y): − mother(x,y).
parent(x,y): − father(x,y).
grandparent(x,y): − parent(x,z),parent(z,y).
father(Jim,Mary).
mother(Mary,Tim).
mother(Mary,Mike).
father(Peter,Marybeth).
father(Roy,Peter).
mother(Martha,Roy).
mother(Louise,Peter).
brother(Stephen,Peter).
```

FIGURE 16-2
Sample PROLOG program.

run of a PROLOG program there is exactly one such query. Figure 16-3 lists several sample queries for the program in Figure 16-2.

The format of a query is basically the same as the right-hand side of a rule, and represents a set of literals all of which have to be proven true simultaneously for some (at least one) consistent replacement of variables by constants. Thus

$$?\!-\mathbf{q}_1(\ldots x\ldots),\ldots,\mathbf{q}_n(\ldots).$$

is equivalent to the wff

$$\exists x\ldots|\mathbf{q}_1(x\ldots) \wedge .. \wedge \mathbf{q}_n(\ldots)$$

PROLOG will try to show that this wff is in fact a theorem of the other clauses that make up the program. Again we can relate this directly back to logic by using the second form of the Deduction Theorem, which states that to prove that the query follows from the rest of the program we negate it and "and" it to the wff formed by the conjunction of the other clauses. If the resulting wff is inconsistent, the original query is a theorem of the program's "axioms."

Note that this negation process converts the above query to the form:

$$\forall x\ldots|(\neg\mathbf{q}_1(\ldots)\vee..\vee\neg\mathbf{q}_n(\ldots))$$

which is simply another single Horn clause with no positive literals. Appending this to the clausal form of the other statements and moving the quantifiers to the left gives a single giant wff in clausal form.

Note also that this form of a query is not universally powerful; it is impossible to state as single queries in PROLOG wffs such as

$$\exists x|\mathbf{a}(x)\vee\mathbf{b}(x) \qquad \text{or} \qquad \exists x|\neg\mathbf{a}(x) \wedge \neg\mathbf{b}(x)$$

```
?-grandparent(Louise,Marybeth). ;Is tuple in relation?

?-parent(x,Tim). ;Who is parent of Tim.

?-parent(x,y). ;look for any parent-child pairs.

?-parent(x,y),father(Roy,x) ;2 goals to satisfy
```

Note: clausal equivalent of last query is:
$\neg$(parent(x,y)$\wedge$father(Roy,x)) = ($\neg$parent(x,y)$\vee\neg$father(Roy ,x))

FIGURE 16-3
Sample queries.

16.2 PROCEDURAL INTERPRETATION
(Kowalski, 1979)

As described above, a PROLOG program can be "read" logically as a series of clauses "and"ed together, one of which represents the negation of a potential theorem for which we would like a proof. "Executing" the program is then equivalent to turning on an inconsistency checker which looks for a substitution for the universally quantified variables in the wffs that causes a contradiction. This is in fact very close to what happens inside the PROLOG inference engine, and will be discussed fully in the next section.

There is, however, another way of interpreting such a program, which is of particular appeal to programmers schooled in conventional languages and which gives an alternative view of the real PROLOG inference engine. This is called a ***procedural interpretation*** of a PROLOG program because of its relationship to the semantics of a procedure call in conventional programming.

In this approach to understanding a PROLOG program, a query is treated as a series of goals to be solved in the specific left-to-right order in which it is written. Each goal is equivalent to a ***procedure call*** to a boolean functionlike procedure which can return either true or false (***succeed*** or ***fail***). The name of the procedure is the predicate name in the goal. In addition, a call which returns success may also return values to bind to some of the variables in the call's ***actual arguments,*** that is, the argument expressions at the time the call was made.

If a particular call returns successfully, our procedural reading would move to the next goal to the right, and assume that we then do a similar procedure call to the routine so mentioned. Any use of variables that were present in earlier calls will have whatever values were bound to them by these previous calls. The values associated with the query's variables by the time the last goal has successfully completed are the "answers" to the query.

For example, the query

$$?\!-\mathbf{p}(x,y),\mathbf{q}(x,z),\mathbf{r}(y,z).$$

would be read as a call to a procedure **p** which should return values for variables x and y, the first of which is used by procedure **q** to find a value for z, which, in conjunction with the original x value, makes the call to procedure **r** return true.

The ***procedure*** called by a goal consists of all the facts and rules whose predicate symbols are the same as that in the goal calling it. The ***body*** of the procedure consists of all these statements, in the order in which they were written in the original program. "Calling" the procedure consists of going from top to bottom and looking at each statement until one is found whose head ***formal arguments*** (as written in the original program statement) unify with the actual arguments of the call. ''Executing'' such a statement consists of making another series of left-to-right procedure calls, this time from the body of the statement.

Variables in the head of the statement are given values from the matching actual arguments during the unification process, and these values are propagated to the right to all occurrences of those variables. In a real sense these variables are the ***formal parameters*** of the procedure (or at least this particular statement of the procedure body).

As each goal on the right is executed, any variables which receive values likewise have these values propagated right.

Successful execution of all the goals in a statement's body causes a successful return from a procedure to the goal which called it, for continued execution of the statement in which the goal is embedded.

As long as goals continue to be solved successfully, PROLOG continues to move relentlessly forward, from left to right, expanding single goals to the list of goals found in the right-hand side of some statement, until there are no more goals left to be solved. At this point, PROLOG stops and reports success to the user. Any values that were bound to the variables in the query can be reported as the answer.

Failure to execute any goal in a statement successfully changes this interpretation into something that is not found in a normal procedural language. Rather than simply stopping and reporting failure to the user, PROLOG will ***backtrack*** to the goal executed just before the failing one, and ask that goal to look for a different solution.

This backtracking involves two actions:

- Erasing all substitutions generated by all procedure calls associated with the goal that failed
- Locating the next statement from the old goal's procedure that might work

After these actions, the new statement is called just as if the one that failed had never been tried. Execution continues as before with this new statement, left to right. If such a solution can be found, the call again completes successfully, the new variable bindings are propagated forward into the original goal's arguments, and an attempt is made to resolve the next goal. Another failure with this new statement causes another backtrack, and another attempt to resolve the goal with yet a third set of bindings. This process can be repeated indefinitely, with backtracks cascading arbitrarily far back.

If PROLOG backtracks to the very first goal, and that fails with no more solutions, then it stops and reports failure to the user. We cannot prove from the given program statements that the query is a theorem.

In summary, a procedural interpretation of PROLOG is very similar to executing a sequential series of procedure calls to boolean functions which can assign values to their arguments, with the exception that each call ''remembers'' which statement it used to compute its most recent bindings, and if any call returns false, the previous call is ''restarted'' from its previous point to find another potential solution. In this respect, each procedure behaves more like a ***co-routine*** than a simple subroutine.

In conventional PROLOG terminology, the information used by a procedure that permits it to be restarted at a later date is termed a ***choice point,*** and is similar to a ***frame*** in a conventional language, or more accurately to what was called a ***continuation*** in function-based programming. As with either of these latter structures, a good mental image of how these choice points are remembered is via a ***stack.*** Each call to a new goal pushes a new choice point onto the stack, just as a call to a procedure in a conventional language pushes a frame of return information to a stack.

There are some differences from this conventional model, however. First, a choice point is not popped at a successful return. Instead it is kept on the stack, and further calls are pushed on top of it. Only the backtracking process described above modifies this stack, and then in one of two ways. A ***shallow backtrack*** occurs when a statement chosen as a candidate for the current goal fails to unify with it, and the next statement down must be tried. The only information that needs to be modified is in the current choice point, which is updated to point to the next statement.

A ***deep backtrack*** occurs when there are no more statements to try, and the current goal must be considered unsatisfiable. In such cases the goal solved prior to the current one is resurrected, and an attempt is made to solve it using a different statement. This involves popping the current choice point and restarting the previous one at the next statement it has listed as a possibility.

We will revisit these terms in later discussions on the PROLOG inference engine and methods of implementing it.

16.3 A PROLOG-LIKE PROPOSITIONAL INFERENCE ENGINE

(Campbell, 1984)

When reduced to basics, the formal PROLOG inference engine consists of a backward-chained decision procedure which uses linear input resolution in a depth-first search fashion to show the inconsistency of a specified clause (the negation of the input query) with respect to another set of clauses (the program). The general approach is easily expressed as a recursive function, and is amenable to a variety of stack-oriented implementations.

To simplify the discussion of the process details, the following sections will proceed in a stepwise fashion. First, a simplified but still PROLOG-like propositional-based logic language will serve as a basis for understanding the decision procedure. This will then be modified to more accurately reflect the backtracking mechanism described in the last section. After that will come an overview of the modifications needed to handle first-order predicate logic, namely, unification and the handling of variables. Finally, indications on how to implement features of PROLOG which are "extralogical" will be discussed in conjunction with the features themselves.

Figure 16-4 describes a propositional logic-based language that "looks like" PROLOG. It has "facts," "rules," and "queries," but is without variables, negation, or several other real PROLOG features. For this text we will call such a language ***Proplog***. Figure 16-8 includes a simple example.

Figure 16-5 diagrams an abstract program description of a PROLOG-like inference engine for this language. It expects two arguments. The first, ***goal-list***, is a list of the goals left to be proved for some statement in the left-to-right order specified by the programmer. Initially it is the query clause provided as input. At any point in time the first, or leftmost, element of this list is the goal the program is currently trying to satisfy.

The other argument is an index into the program to select the next statement to try for a match. As the program executes with a particular *goal-list*, this index starts with the first statement and proceeds toward the last, in order.

If the program can successfully prove all elements of the *goal-list*, it returns true, indicating that it has completed the body of some statement. If it fails to find a consistent solution, it returns false. This matches the general intent of the "procedural reading" discussed earlier.

Internally, the program works on one goal at a time, from left to right, with the basic computation being a check if the leftmost literal of the *goal-list* resolves with the head of the specified program statement. In this case resolution is equivalent to an equality test between the two predicate symbols. If there is a mismatch, **prove** will proceed to try the next statement in the program against the same *goal-list*. If a resolution is possible, the procedure will then call itself recursively, with the resolvant's body used as the new *goal-list* and rule 1 used as the first possible other parent. A successful execution of this call causes **prove** to restart the original *goal-list*, but with the leftmost goal discarded.

```
<goal> := <letter>
<head> := <goal>
<body> := <goal>{,<goal>}*
<fact> := <head>.
<rule> := <head>:-<body>.
<query> := ?-<body>.

Example: (See Figure 13–13)
b:-a.
d:-b,c.
g:-d,e.
g:-d,f.
a.
c.
?-d.
```

FIGURE 16-4
Propositional logic language—
Proplog.

```
prove(goal-list, rule#) =
  if null(goal-list) then T ;Success
  elseif rule#>size-of-program then F ;failure—deep backtrack
  else let rule = program[rule#] ;get next statement
      and goal = get-leftmost-goal(goal-list)in
          if goal≠get-head(rule) ; Simple unification
          then prove(goal-list, rule#+1) ; Shallow backtrack
          elseif prove(body(rule),1) ; Try body as new query
          then prove(drop-leftmost-goal(goal-list),1) ; Yes
          else prove(goal-list, rule#+1) ; No—shallow backtrack
```

FIGURE 16-5
Simplistic Proplog inference engine.

16.3.1 Goal-Stacking Version

The inference engine of Figure 16-5 works fine for propositional logic, but fails when adapted to first-order logic. It has no mechanism for, or need of, propagation of unification substitutions from the head/goal match into both the clause's body and the other remaining goals.

Figure 16-6 diagrams a modification of the above inference engine which exhibits the desired properties (even though propositional logic does not need it). From this version the jump to first-order logic will be much clearer.

At any point in time, the original **prove** had a *goal-list* argument which corresponded to at most the righthand side of a single statement. The new version has not only this right-hand side, but the remaining

```
prove(goal-list, rule#) =
  if null(goal-list) then T ;Success
  elseif rule#>size-of-program then F ;deep bactrack
  else let rule = program[rule#]
      and goal = get-leftmost-goal(goal-list) in
          if goal≠get-head(rule)
          then prove(goal-list, rule#+1) ;shallow backtrack
          else let resolvant=append(body(rule),
                                 drop-leftmost-goal(goal-list))
              and temp=prove(resolvant, 1) in
                  if temp then T ;success
                  else prove(goal-list, rule#+1)
```

FIGURE 16-6
Revised Proplog inference engine.

right-hand sides of all the statements that have not yet been finished. Thus the remaining goals from the query are on the right end, and the most recently established goals (from the body of the just chosen clause) are on the left. The major change from the previous version is that after a successful resolution, the new *goal-list* constructed for the next call is constructed by appending the right-hand side of the successful statement to the front of the *goal-list* obtained by dropping the leading goal.

The reader should verify that this new *goal-list* corresponds exactly to the clause that would have been developed by resolving the original *goal-list* (in clausal form) with the specified statement.

There are two terminating conditions inside the function, when all the statements are exhausted against a particular literal and when the *goal-list* reaches empty. The latter case clearly indicates final success; a contradiction has been reached for the entire initial query. Note that this is a more general result than that for Figure 16-5 because in the latter the true result proves only that one set of goals has been solved for one statement. The recursively-built stack of callers to **prove** may have other goals left to solve.

The other terminating case indicates that there are no resolvants from the first literal of the current *goal-list* that lead to the empty clause. ***Deep backtracking*** occurs here to return **prove** to the prior *goal-list* and ***rule#,*** where it will pick up as if statement *rule#* had not resolved to begin with, and the next program statement will be tried against the original goal. If there are no more statements to try here, the backtrack process repeats. Only when this stack of recursive calls to **prove** is exhausted does the procedure stop with a "T" answer.

In summary, there are two conditions that could cause **prove** to try a different statement against a particular goal. First is if the prior statement failed to unify with it. Second is when there are no more statements to try. An action of the former type is a ***shallow backtrack,*** while the latter is a ***deep backtrack.***

16.3.2 Flowchart Form

Figure 16-5 is a procedure that uses recursion to handle both types of backtracking, although only in the latter is the full power of recursion really needed. To demonstrate this, Figure 16-7 gives a flowchart equivalent of the same algorithm, but with the shallow ***tail recursion*** calls to **prove** replaced by simple looping iteration, and only the absolutely necessary recursion handled by explicit stacking and unstacking of parameters. The reader should carefully compare these two figures to verify their correspondence.

For obvious reasons this form of the PROLOG inference engine is termed a ***goal-stacking model.***

There are a few things worth noting here. First, if the clause chosen from the program is a fact (a single literal), then the *goal-list* is shorter after the resolution than before it. Thus, if the length ever reaches 0, Proplog can stop because it has reached a contradiction, implying that the original query (before conversion into a disjunction of negated literals) is a valid theorem.

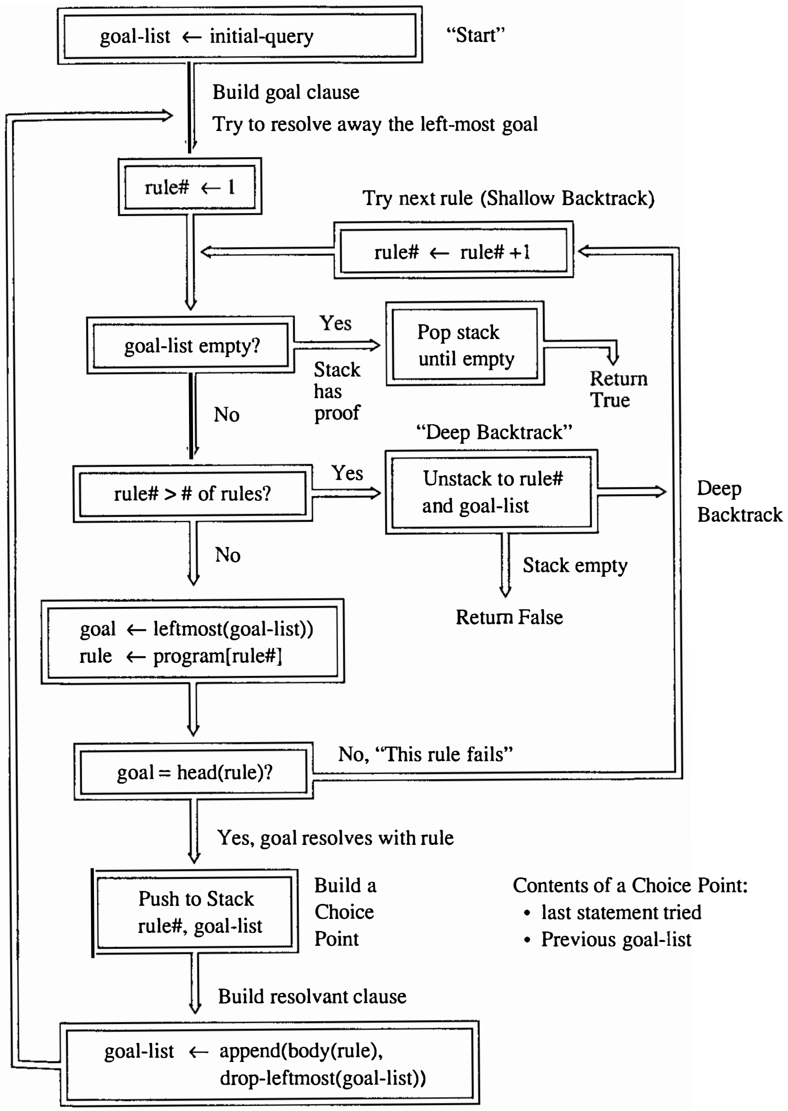


FIGURE 16-7
Flowchart form of Proplog inference engine.

Clearly, if the parent statement was a rule, the length of the *goal-list* will either stay the same or increase, depending on how many literals there are in the body of the clause.

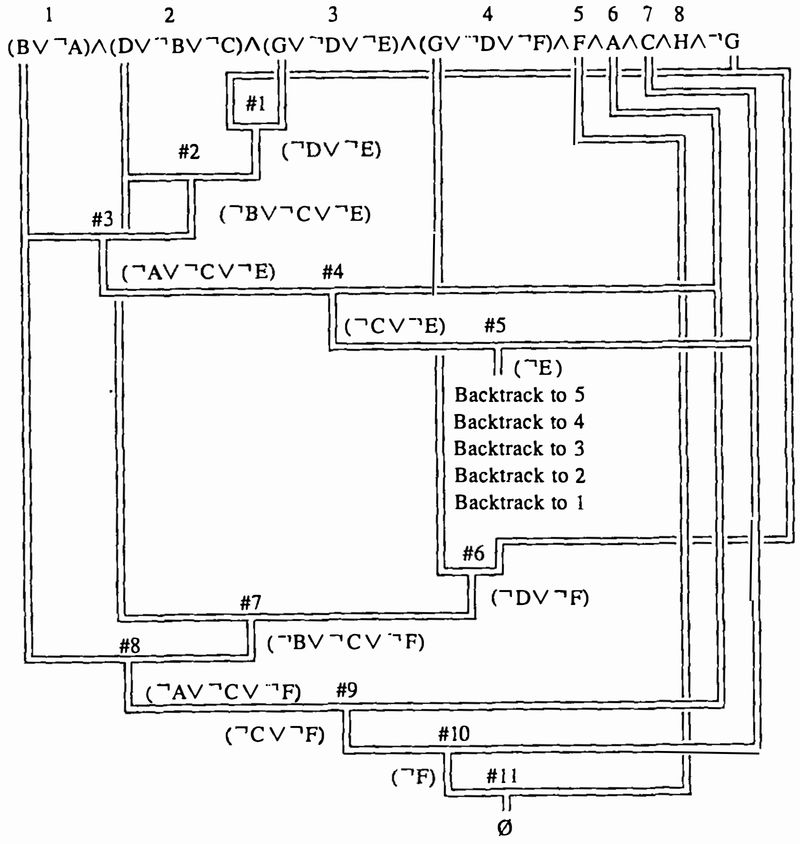
Second, the information placed on the stack at each successful resolution is exactly that information needed to restart the computation if the first goal of the new *goal-list* proves insolvable. This information is exactly what we earlier identified as a ***choice point.***

Finally, at the moment when the program successfully completes (the *goal-list* becomes empty), the contents of the stack contains all the information required to reconstruct a complete proof tree. Each entry contains a clause and an index into the program of the clause that resolved successfully with the leftmost literal in it. The clause in the entry on top of it is the result of the resolution. The bottommost entry is the original query, and the topmost is the empty clause.

16.3.3 A Simple Example

Figure 16-8 diagrams both a sample problem in this simple language, a clausal form of the whole program plus query, and a diagram of the sequence in which inferences are tried in its solution. Figures 16-9 and 16-10 diagram the stack frames built by Figure 16-7 as it progresses through the same problem. Note the contents when the empty clause is finally discovered.

Axioms:	1.B:−A.	2.D:−B,C.	3.G:−D,E.	4.G:−D,F.
	5.F.	6.A.	7.C.	8.H.
Query: G				

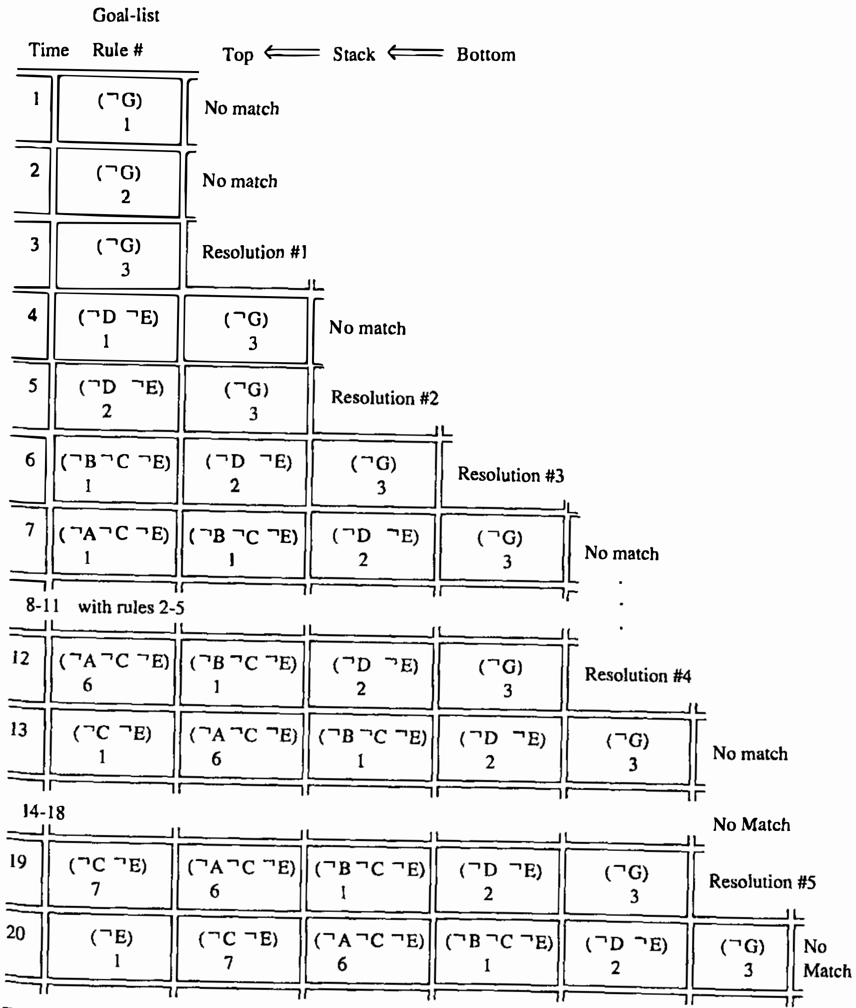


Note: #n denotes the n'th inference made by the program.

FIGURE 16-8
Sample Proplog derivation.

16.4 THE PROLOG INFERENCE ENGINE

The major complication in adapting the prior inference engine to first-order predicate logic is the need to consider variables and other expressions as arguments to predicate relations. Handling this requires at a minimum replacement of the "if *goal* ≠ **get-head**(*rule*)" test by a more complex, unification-based pattern matcher. Additionally, we must de-



Time	Goal-list Rule #	Top ⟸	Stack ⟸			Bottom	
1	(¬G) 1	No match					
2	(¬G) 2	No match					
3	(¬G) 3	Resolution #1					
4	(¬D ¬E) 1	(¬G) 3	No match				
5	(¬D ¬E) 2	(¬G) 3	Resolution #2				
6	(¬B ¬C ¬E) 1	(¬D ¬E) 2	(¬G) 3	Resolution #3			
7	(¬A ¬C ¬E) 1	(¬B ¬C ¬E) 1	(¬D ¬E) 2	(¬G) 3	No match		
8-11	with rules 2-5						
12	(¬A ¬C ¬E) 6	(¬B ¬C ¬E) 1	(¬D ¬E) 2	(¬G) 3	Resolution #4		
13	(¬C ¬E) 1	(¬A ¬C ¬E) 6	(¬B ¬C ¬E) 1	(¬D ¬E) 2	(¬G) 3	No match	
14-18						No Match	
19	(¬C ¬E) 7	(¬A ¬C ¬E) 6	(¬B ¬C ¬E) 1	(¬D ¬E) 2	(¬G) 3	Resolution #5	
20	(¬E) 1	(¬C ¬E) 7	(¬A ¬C ¬E) 6	(¬B ¬C ¬E) 1	(¬D ¬E) 2	(¬G) 3	No Match

FIGURE 16-9
Stack growth for sample problem (part 1).

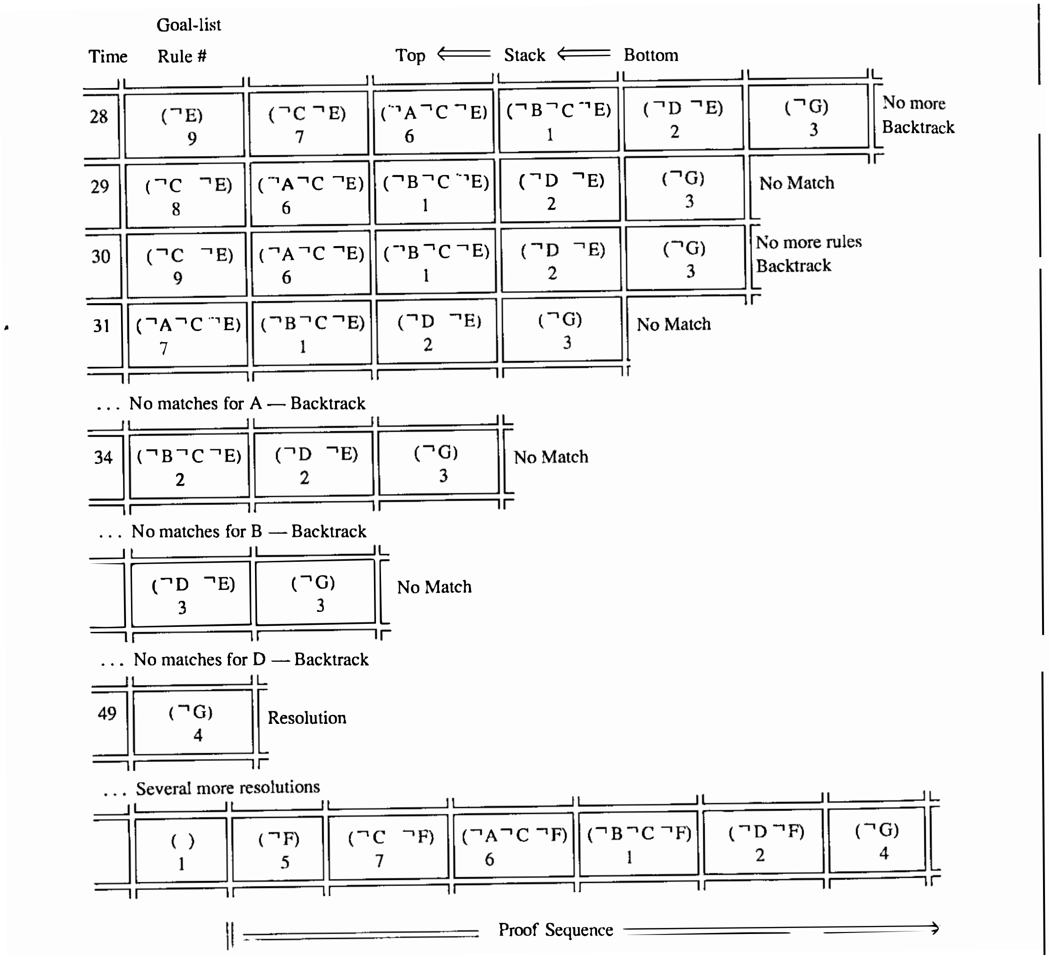


FIGURE 16-10
Stack growth for sample problem (part 2).

cide what to do with the substitution generated by the unification match and the construction of the actual resolvant clause to be used next.

The easiest approach is to assume that we have a ***unify*** routine which takes two literals, unifies them, and returns either the unifying substitution (as an s-expression association list) or an indication (as a nil or False) that the literals are not unifiable. This is compatible with the unification routine described in the last chapter. This routine could then be applied to the goal and the head literal in the selected program statement, and used in place of the "≠" test.

After finding that the two literals unify successfully, the routine must then compute the resolvant clause. As before, this involves appending the resolvant (clause body) to the old *goal-list*, but now we must also apply the unifying substitution to the whole thing. This is necessary because there may be variables in the original *goal-list* which are to receive substitutions as a result of the unification.

As an example, consider the *goal-list* (**p**(3,*x*,*y*,*z*) **q**(*x*,*y*) **r**(*y*,*z*)) and the clause

p(*w*,4,*t*,*t*):−**s**(*w*,*t*).

Resolving the two creates the substitution [3/*w*,4/*x*,*y*/*t*,*y*/*z*], which when applied to the body and remaining goals results in the new *goal-list*:

(**s**(3,*y*) **q**(4,*y*) **r**(*y*,*y*)).

Note that a new *goal-list* created in this fashion must use a physically separate copy of the original *goal-list* so that any changes made to variables in it do not affect prior resolutions still in the stack. This is to permit backtracking to erase the new substitutions if using this clause dead-ends.

Note also that the same rule might be used many times in different contexts in a proof sequence, and consequently each use must assume a "different" set of variable names from the last. This requires that some kind of ***renaming*** process be applied to a statement before it can be used in even the unification test. This renaming will change all variable names in the statement to a consistent set guaranteed not to have been seen before in the current proof sequence.

A common approach to this renaming is to "tag" each variable name with an invisible subscript which equals the current depth of the proof stack. This guarantees that each use of a rule will exhibit distinct variable names, since each use will have a different number of choice points stacked up.

Figure 16-11 uses these techniques in an abstract program version of **prove** adapted to first-order logic. The **unify** and **substitute** routines are the ones from earlier chapters. The additional argument *depth* for **prove** and **prove2** is initialized to 0 on the initial call to **prove,** and is incremented internally each time a new choice point is built. This *depth* is used in **rename-vars** to rename each variable name in each clause to something of the form (⟨***symbol***⟩.*depth*).

As before, it is a relatively simple matter to replace the tail-recursion cases in this program by iteration, and stack only the essential choice points.

A later figure (see Figure 16-18) diagrams execution for a simple case.

16.4.1 An Environment-Stacking Inference Engine

A major problem of the above approach is the explicit copying of partial clauses when resolvants are built. This copying costs both storage for the copies and computation time for applying the substitution, and is partic-

```
prove(goal-list.) = prove2(goal-list,1,0)

prove2(goal-list, rule#, depth) =
  if null(goal-list) then T ;success
  elseif rule#>size-of-program then F
  else let rule = rename-vars(program[rule#],depth)
       and goal = get-leftmost-goal(goal-list) in
           let subs = unify(goal, get-head(rule)) in
              if subs=F
              then prove2(goal-list,rule# + 1,depth)
              else let new-goal-list= append(get-body(rule),
                                      drop-leftmost-goal(goal-list))) in
                   let resolvant=substitute(new-goal-list,subs)
                   in let temp=prove2(resolvant,1,depth+1) in
                      if temp then T
                      else prove2(goal-list, rule# + 1, depth)

  rename-vars(clause, depth) = "returns a copy of clause where all
  the variables have been replaced by (<name>.depth)"
```

FIGURE 16-11
Goal-stacking Horn clause inference engine.

ularly wasteful if a later backtrack throws it all away (see problem 4 at the end of this chapter). Consequently, many real implementations tend to keep the substitutions in separate ***association list***-like structures called ***environments,*** which can be pushed and popped like the *goal-lists* and referred to only when a variable is encountered during the unification process. This is very similar to that used previously for function-based computing, particularly the SECD Machine.

This alternative implementation is also useful when the empty clause is found, since then the inference engine can look up the current values assigned to all variables in the original query and print them out as answers to the computation.

Figure 16-12 diagrams one version of an inference engine based on the use of environments. The main function **prove3** accepts an association list as its fourth argument *env,* which is then passed to **unify** to look up values. In turn, upon a successful unification, **unify** returns a new association list.

To make this function work properly, we must know the depth at which each variable appeared in a goal for the first time. Rather than using the **rename-vars** function, this procedure assumes that each goal in the *goal-list* is a dotted pair of the form ((***literal***).“depth”). The **map** function in the definition of **prove** does this for the initial query (at depth 0). Inside **prove3** the same thing happens when the body of a new rule is appended to the old *goal-list* (and bound to *resolvant*). The depth tagged onto these new goals is the current depth of the computation.

Figure 16-13 diagrams a unification routine to go along with this inference engine. This **unify** has three arguments beyond the two literals to be unified. Two of them are the depths of the literals (used for renaming variables); the last is the current environment as an association list. Variable names inside the association list have the form (⟨***identifier***⟩.*depth*) to permit different versions of the same name to be properly identified.

```
prove(query) = prove3(map((λx|(x.0), query), 1, 1, nil)

prove3(goal-list, rule#, current-depth, env) =
  if null(goal-list) then T ;success
  elseif rule#>size-of-program then F
  else let rule = program[rule#]
       and goal = caar(goal-list) ;leftmost goal
       and goal-depth = cdar(goal-list) ;depth of that goal
       and subs = unify(goal, get-head(rule), goal-depth,
                  current-depth, env) in
         if subs=F
         then prove3(goal-list,rule# + 1, current-depth, env)
         else let resolvant=append(
            map((λx|(x.current-depth), get-body(rule)),
            cdr(goal-list))
              and temp=prove3(resolvant, 1, current-depth+1,
              subs) in
                 if temp then T
                 else prove3(goal-list, rule# + 1, current-depth, env)
```

Assume: All goals in goal-list for prove3 of form (<literal>.depth), where depth is stack depth at time of goal creation.

FIGURE 16-12
Environment-stacking Horn clause inference engine.

The function **unify-args** actually does all the work of unifying the arguments one by one recursively.

16.4.2 Capturing the Solution Substitution

The inference engines described above stop as soon as the first proof sequence is found, flush their internal call/return stacks of all entries, and return success to the original caller. While this indicates that the query logically follows from the original program (i.e., is a theorem), it does not give the substitutions that were made for the variables in the query that were needed to complete the proof. In most cases this information is the reason why the user constructed the query in the first place.

Depending on the exact implementation for the inference engine, there are a variety of methods for saving this information. For a goal-stacking version as pictured in Figure 16-11, we need to make sure that any substitutions generated are applied to the variables in the original goal. There are at least two ways of doing this:

```
unify(goal-lit, rule-lit, goal-depth, rule-depth, env) =
  if get-pred-symbol(goal-lit) = get-pred-symbol(rule-lit)
  then unify-args(get-args(goal-lit), get-args(rule-lit),
  goal-depth, rule-depth, env)
  else F

unify-args(goal-args, rule-args, goal-depth, rule-depth, env) =
  if null(goal-args)
  then env
  else let g = translate(car(goal-args), goal-depth, env)
       and r = translate(car(rule-args), rule-depth, env) in
           if is-a-constant(g)∧is-a-constant(r)
           then if g=r
                then unify-args(cdr(goal-args), cdr(rule-args),
                                goal-depth, rule-depth, env)
                else F
           elseif is-a-var(g)
           then unify-args(cdr(goal-args), cdr(rule-args),
                           goal-depth, rule-depth, ((g.r).env)
           elseif is-a-var(r)
           then unify-args(cdr(goal-args), cdr(rule-args),
                                goal-depth, rule-depth, ((r.g).env)
           elseif get-function-symbol(g)=get-function-symbol(r)
           then let s1=unify-args(get-args(g), get-args(r),
                                   goal-depth, rule-depth, env) in
                if s1=F
                then F
                else unify-args(cdr(goal-args), cdr(rule-args),
                                goal-depth, rule-depth, s1)
           else F

translate(x, depth, e) = if is-a-variable(x)
  then let s=assoc((x.depth), e) in
       if null(x) then x else translate(cdr(s),depth, e)
  else x
```

Note: env=((<identifier>.depth).value)*)

FIGURE 16-13
Unification using depths and association lists.

1. Adding an extra argument to **prove2** which consists of a list of the original variables in the query
2. Adding a dummy goal to the end of the original query which consists of some predicate such as **end-goal** with a list of arguments again corresponding to all the variables in the query

In the first case, each time we find a clause which unifies with the front of the current *goal-list*, we also apply the substitution to the current copy of the extra argument. The first time the *goal-list* empties (*resolvant* becomes null), the value of this extra argument contains the desired substitutions.

In the second approach, the dummy goal receives all the necessary substitutions as a normal matter of course. Consequently, the only change needed is detecting when the actual *goal-list* has run out and the dummy terminating goal has been reached. This could be done by adding a test to **prove** of the form if **get-predicate**(*goal*)=end-goal then "print argument values and return true."

In the case of an environment-stacking inference engine, the answer is much easier. Again, if we include as an argument to **prove3** a list of the query variables, then upon first detecting an empty *goal-list* we need only evaluate each of these variables using the current environment list. The resulting value is what should be printed out along with the variable name.

16.4.3 Multiple Answers

In real situations there are many cases where the user may actually want not just the first but perhaps all the possible proof sequences for the given goal.

A simple modification to **prove** in any of the above forms provides such a capability. Basically, when the end of a proof sequence is discovered (a null *goal-list*), instead of simply printing the answer and returning T to the caller (triggering the chain of stack pops), a small procedure could ask the user if more answers are desired. If they are, instead of returning T to its recursive caller, the procedure could return F, causing a backtrack to the top choice point on the stack, and pick up from there as if the last resolution (which gave the null clause) had not been possible. The normal inferencing procedure would then continue until the next possible solution was found, at which point the process could be repeated.

If there are in fact no other solutions, the inference engine will eventually backtrack to an empty stack, and stop with a returned value of F.

In many PROLOG implementations, "asking the user" consists of stopping the cursor just after printing out the last query variable and its substitution. If the user types in a semicolon (";"), the system causes a backtrack as describes above, and continues to the next solution. Typing in anything else causes the stack to be cleared and a new query to be requested.

16.4.4 Relationship to Deduction Trees

Another useful way of viewing the PROLOG inference engine is in terms of how it builds the equivalent of a ***deduction tree*** for its original query. To do this we assume that the deduction tree starts with a single node which expands out to N open nodes, one for each literal in the query. PROLOG then extends this tree in the following fashion:

1. Always pick for expansion the leftmost, lowest unexpanded node. If there are no such nodes, the tree is closed, and a solution is found.
2. Starting with the first clause in the program, try to find a clause which can successfully extend the node just chosen. This requires renaming all variables in the clause, and having a successful unification between the chosen open node and the head of the clause.
3. If a successful unification is found, propagate any value bindings to variables in the original node up and through the rest of the tree, and to variables in the clause body. Mark the node just extended with the number of the statement which expanded that node. Then go back to step 1 and pick another node for expansion.
4. If no clause is available to extend the node chosen in step 1, remove the last extension made to the tree, and retract all bindings made to variables as a result of that extension. Note that the extension retracted is not always a parent of the node which was unexpandable.
5. Then try to reextend the node just revealed by the above removal, picking up from whatever clause was just removed. If there are no such clauses available, step 4 is repeated on the next previous extension.
6. If we ever find ourselves unable to extend in any way the leftmost node in the original query, then there is no solution to the query.

There are several direct connections between this model and the storage used by the inference engines described above, particularly the flowchart form of the goal-stacking one. First, at any point in the computation, each closed node in the tree corresponds directly to a choice point pushed on the stack. Further, if one numbers these closed nodes in the following fashion, then the numbers correspond directly to the relative position of the associated choice point in the stack.

1. Number the leftmost node of the query 1.
2. If a node has just been labeled N, then if it has any children that are extended, label the leftmost child N+1, and then repeat the process on this child.
3. If a just-labeled node has no children, select the next rightmost child of the parent.
4. If there are no more children at this level, back up and down the tree until a node is reached which has some unnumbered node, and select the leftmost such node.
5. If all nodes are labeled, quit.

In graph theory such a labelling is called a ***graph preorder traversal.***

As an example, Figure 16-14 diagrams the deduction tree for the

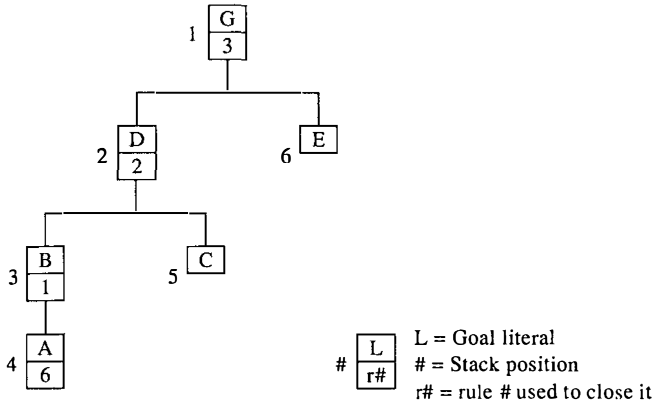


FIGURE 16-14
Deduction tree for Figure 16-8.

problem of Figure 16-8 just after resolution #5 (at time 19 of Figure 16-9). The reader should compare the numbering assigned by the above process to the position of choice points in the stack at that time.

Finally, at any point in the computation (particularly before the final solution has been reached), the open nodes of the deduction tree, as read from left to right, correspond directly to the *goal-list* at that point, in exactly the same order. In this case this is ¬C,¬E, just as in the earlier figure.

16.5 SPECIAL FEATURES OF PROLOG

There are at least four categories of special features in PROLOG that were added to make it a complete programming language but that do not correspond to anything from mathematical logic. This section overviews some of the items from each category, why they are useful, and generally how their implementation affects our model of PROLOG semantics. For more specific details the reader is referred to the appropriate programming manuals available on the language.

16.5.1 Special Terms

The first category of features are what one might call "special terms," and provide ways of specifying unique objects and data structures. Their major implementation impact is in the unification algorithm used by PROLOG, which now must monitor the expressions being matched for their occurrence.

The first of these is the ***anonymous variable,*** written as "_." This variable can occur anywhere a normal variable can, but it has the property that it matches anything (like a "wild card") and adds nothing to the unifying substitution. This aids efficiency in cases where the value of an argument is not needed.

The second feature in this category is the ability to define, create, and dissect what are basically standard s-expressions anywhere a term is permitted as an argument. Thus we can write literals such as ***append***((1 2),(3 4 5),*x*) and interpret them exactly as we might expect, i.e., as a relation of tuples where each component is a list, and the third component is a list that represents the appending of the first list to the front of the second. (Actually, in Clocksin and Mellish syntax, lists are surrounded by [ ], with the **cons** operator "." written as "|"). Unification of such terms works exactly as if we had written them as function applications involving the function ***cons,*** as in:

append(**cons**(1,**cons**(2,nil)),**cons**(3,**cons**(4,**cons**(5,nil))),*x*).

The normal PROLOG unification algorithm would handle this as any other expression involving functions. No predefined meaning would be given to **cons,** but by appropriate construction of expressions involving **cons,** variables, and constants, any normal list structure could be duplicated. The primary advantage of the "built-in" notation is ease in reading and in supporting some other built-in predicates which treat clauses as lists.

Figure 16-15 lists some sample unifications. Figure 16-16 gives both the abstract program form of some standard list-processing functions described in prior chapters and some equivalent PROLOG statements that implement them. Note that functions which give boolean results translate directly to PROLOG predicates with the same number of arguments, while functions that return nonboolean objects translate to predicates with one more argument. In order for a tuple of arguments to be a member of that predicate, the last argument in such cases must correspond to the value that the functional form of the predicate would have given when presented by all but the last argument. This is a standard trick in logic programming.

List 1	List 2	Unifying Substitution
(1 2 3)	(x y z)	[1/x,2/y,3/z]
(1 2 3)	(x.y)	[1/x,(2 3)/y]
(1 2 3)	(x.(y.z))	[1/x,2/y,(3)/z]
(1 2 3)	(x 3 z)	IMPOSSIBLE
(1 2 3)	(_ y 3)	[2/y]
(1 2 3)	(_.(_.(_.x)))	[( )/x]
((1.2).3)	x	[((1.2).3)/x]
((1.2).3)	(x y)	IMPOSSIBLE
((1.2).3)	(x.y)	[(1.2)/x,3/y]
((1 2) q 3)	(x x z)	[(1 2)/x,(1 2)/q,3/z]

FIGURE 16-15
Unifying list structures.

```
Functional Definition: member(x,y) = ; Returns T or F
  if null(y) then F
  elseif x=car(y) then T
  else member(x,cdr(y))

Prolog Form:
  member(x, (x._)).
  member(x, (_.y)):-member(x,y).
```

(*a*) Set membership.

```
Functional Definition: append(x,y) =
  if null(x) then y
  else cons(car(x), append(cdr(x),y)))

Prolog Form: (third argument is result of append)
  append(( ),y,y).
  append((h.t),y,(h.z)):-append(t,y,z).
```

(*b*) List append function.

```
Functional Definition: reverse(x) = rev(x,nil)
wherereс rev(x,y) =
  if null(x) then y
  else rev(cdr(x),cons(car(x), y)))

Prolog Form: (second argument is reversed list)
  reverse(x,y):-rev(x,( ),y).
  rev(( ),y,y).
  rev((h.t),y,z):-rev(t, (h.y),z).
```

(*c*) Basic reverse function.

FIGURE 16-16
Standard list functions in Prolog.

In particular, Figure 16-17 gives some examples of the use of the **append** procedure. The first case corresponds to the normal function invocation; the second, however, corresponds to running **append** "backwards," i.e., giving a list that might have resulted from an **append** and asking what list(s) might have formed it. Note that multiple answers are possible, and, as described above, the user can request all these solutions from most real PROLOG systems, and will receive them in the order given. The reader should review the interpreter model of PROLOG to guarantee an understanding of how and why this comes about.

In the literature this **append** definition is often called the ***concatenate*** procedure, with calculations of the above types called ***deterministic concatenate*** and ***nondeterministic concatenate,*** respectively. In the former case we are running the computation like a function evaluation; there is exactly one, and only one, answer. In the latter case multiple answers are possible, and it is "indeterminate" without a good deal of analysis to predict which comes first.

Figure 16-18 gives a partial stack picture of a particularly complex

Query: ?− append((1 2), (3 4 5), x).

→ append((2), (3 4 5), z_1)
where [1/h_1, (2)/t_1, (3 4 5)/y_1, (1.z_1)/x]

→ append(( ), (3 4 5), z_2)
where [2/h_2, ( )/t_2, (3 4 5)/y_2, (2.z_2)/z_1]

→ nil
where [(3 4 5)/y_3, y_3/z_2]

Thus x=(1.(2.(3 4 5))) = (1 2 3 4 5)

(*a*) Normal direction (deterministic computation).

Query: ?− append(x, y, (1 2 3 4 5)).

Result: T when x=( ), y=(1 2 3 4 5)
or when x=(1.( )), y=(2 3 4 5)
or when x=(1 2), y=(3 4 5)
or when x=(1 2 3), y=(4 5)
or when x=(1 2 3 4), y=(5.( ))
or when x=(1 2 3 4 5), y=( )

(*b*) Reverse computation (nondeterministic computation).

FIGURE 16-17
Sample derivations from **append.**

nondeterministic computation. Note that this figure explicitly lists variable names "renamed" as described earlier by tagging them with the depth of the call.

16.5.2 Built-in Computational Predicates

PROLOG includes a variety of predicates that are "built-in," i.e., which have a special and predefined "meaning" assigned to them. The first group of these might be classified as "computational," since they actually do things normally expected of a conventional computer program. Since they have a built-in meaning, they can be used only in a rule body (antecedent), are detected by the decision procedure, and are executed directly (via code in the inference engine) rather than used in a search process through the other statements.

Because they do not trigger searches, such predicates usually establish no choice points, and thus if backtracking is triggered by the failure of a predicate to their right in some rule, the choice point returned to is the last one established to their left.

Further, because their meaning is "built-in," many of these predicates require some or all of their terms to be ***instantiated*** by the time PROLOG executes them. This means that if there are any variables in the predicate's arguments, prior predicates in this rule must have assigned them values before reaching the predicate. If values have not been assigned, an error condition is flagged.

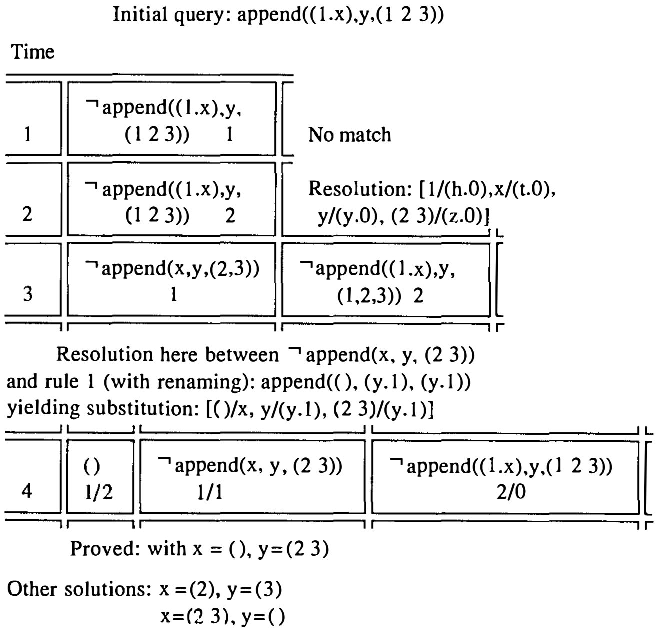

Initial query: append((1.x),y,(1 2 3))
Time
1
¬append((1.x),y, (1 2 3)) 1
No match
2
¬append((1.x),y, (1 2 3)) 2
Resolution: [1/(h.0),x/(t.0), y/(y.0), (2 3)/(z.0)]
3
¬append(x,y,(2,3)) 1
¬append((1.x),y, (1,2,3)) 2
Resolution here between ¬append(x, y, (2 3))
and rule 1 (with renaming): append((), (y.1), (y.1))
yielding substitution: [()/x, y/(y.1), (2 3)/(y.1)]
4
() 1/2
¬append(x, y, (2 3)) 1/1
¬append((1.x),y,(1 2 3)) 2/0
Proved: with x = (), y=(2 3)
Other solutions: x =(2), y=(3)
x=(2 3), y=()

FIGURE 16-18
Stack growth for simple **append.**

The first of these computational predicates is "=." A predicate of the form "**A**=**B**" will be true only if the terms **A** and **B** can be unified. In this case neither term need be instantiated, and the result of the unification can assign values to variables for use in later predicates.

A slightly different set of predicates are the numeric comparators such as ">" and "<." These require both arguments to be instantiated to numbers, and simply execute the specified "yes/no" test on their values. Thus "x>7" will not try to assign a value to x if it does not have one already; instead, it will error out.

The arithmetic predicates +, −,...are combinations of the above two. These take three arguments, two for the inputs and one for the result. If all three arguments are instantiated, these predicates will verify if the numbers are consistent. Thus "+(3,4,7)" will succeed, while "×(5,x,30)" will not if x has been assigned the value 7. If, however, two of the arguments are instantiated to numbers and the third (in any position) is a variable, the predicate will return a true value along with an update to the substitution list that assigns the appropriate value to the variable. Thus if x has no value before "×(5,x,30)," it will have the value 6 afterward.

Another predicate of this form is ***is***. A predicate of the form "**A** is **B**" will succeed if **A** is an uninstantiated variable and **B** is an expression which can be evaluated in the normal arithmetic fashion to yield a number. Thus "x is 2×(y−1)" will pass as true, with the assignment x=14 if

y had the initial assignment y=8. Figure 16-19 uses this predicate in one form of a PROLOG factorial routine.

Finally, the user can perform conventional I/O using predicates such as ***read*** and ***write.*** The former expects a single uninstantiated variable, and when encountered in a rule, stops and waits for input from the user's terminal. When that arrives, it is placed as a value for the argument variable, and the next predicate on the right is started. Conversely, the argument for **write** must be instantiated and is printed out to the user's console when the predicate is encountered. Note that in neither case can we "back up" the I/O operation if a later predicate fails.

Although none of these predicates build choice points, PROLOG does remember if any variables were assigned values during their execution, so that if later backtracking is required, those variables can be unbound back to their original state.

16.5.3 Control Predicates

PROLOG also includes predicates that permit direct user control over the backtracking process. ***Fail,*** for example, is a predicate of no arguments that always causes a backtrack to the prior choice point. While not strictly needed, it is often convenient when used with some of the other control predicates. The predicate ***true*** is similar to **fail** except that it always works.

NOT. The predicate ***not*** accepts as its single argument some other predicate and its arguments. **Not** succeeds only if this other goal fails, that is, there is no sequence of resolutions and substitutions which will result in the empty clause for the predicate given its particular arguments. Because of this, **not** will never add to the substitution for the rule that it is in.

Note that this is radically different from the **not** in logic; **not(E)** in logic is true only if **E** is unsatisfiable for any substitution. This is different from simply failing to find one from the set of given clauses. For this reason, PROLOG's **not** is often called ***negation by failure.*** Naish (1985) gives a thorough description of these differences.

As an example of this, consider the following queries:

?−**not(member**(4,(1 2 3)))
?−**member**(z,(1 2 3)),**write**(z).
?−**not(not(member**(z,(1 2 3)))),**write**(z).

All three of these queries succeed, but they do so in different fashions.

```
factorial(0,1).
factorial(x,y):−z is x−1, factorial(z,q),y is x×q.
```

FIGURE 16-19
A PROLOG factorial routine.

The first succeeds because it is false that 4 is a member of the list (1 2 3). The second succeeds and prints out 1, because z is bound to the first possible answer which makes the predicate true, namely, 1. (The reader should determine what happens if the user of this query asks for additional solutions.)

Finally, the third query also succeeds, but no value is printed out for z. The **member** inside the double **not** succeeds with x=1, which means that the inner **not** fails (there is a way of making its goal true). The substitution that made it fail is tossed away, and only the fact of failure is passed on. This, however, satisfies the outer **not,** causing the **write**(z) goal to be attempted with no value bound to z.

DISJUNCTION. Another control predicate is ";", which is also called ***disjunction.*** As a goal of the form (**q**;**r**), for example, this predicate will be true if either **q** evaluates to true, or **q** fails but **r** succeeds. These ; can be cascaded in as many places as desired, with the either-or interpretation extended naturally. For all practical purposes, a clause of the form **p**:−**u,(q;r;t),w.,** for example, is equivalent to the clauses:

p:−**u,q,w.**
p:−**u,r,w.**
p:−**u,t,w.**

CUT. Perhaps the most important control predicate is the ***cut*** (symbolized as "!"). When first encountered in an antecedent, it acts just like **true.** PROLOG immediately goes to the predicate on its right. If, however, the predicate on the right ever fails and backtracking returns to the !, normal backtracking is suspended. Instead of returning to the predicate on the immediate left of the !, the inference engine goes back through the stack of choice points to the one corresponding to the invocation of the consequent for the rule containing the !. This choice point is failed, and backtracking returns to the choice point in back of it.

Essentially, ! sets up a "fence" that "freezes" this rule as the only one which can possibly satisfy the most recent goal matching the rule's head (cf. Figure 16-20). Executing it has the effect of deleting (for the current goal) the rest of the clauses with the same predicate name and of "erasing" the choice points to all literals to the left of the ! in the current clause. If the predicates on the right of the ! fail, then there is no solution to that goal, and none of the other possible rules with the same predicate but below the current rule in program order is tried.

Figure 16-21 diagrams a sample computation using the cut predicate for our simple propositional form. In this case the cut is equivalent to replacing the rules inferring **A** by the set

A:−B,C.
A:−not(B),D,E.
A:−not(B),E,Q.

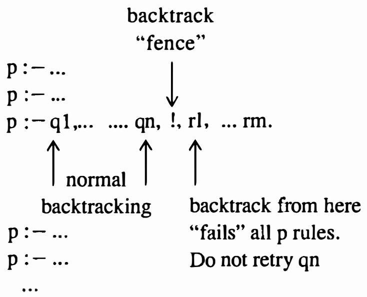


FIGURE 16-20
Cut predicate.

Note the effect of this replacement. If **B** is proved true, then the truth of **A** rests solely on the first rule, for the later rules will reprove that **B** is true and fail because of the **not** in front of it.

The first-order logic version is not quite so simple because of the need to keep track of the substitutions that occur in the first rule's execution, but the net effect is similar.

The use of ! is primarily for efficiency in cases where the programmer either wants only the first answer, or knows that if certain conditions occur then there is no answer, and thus there is no point in pursuing alternative rules.

OTHER CONTROL PREDICATES. Next, as described earlier, finding all solutions to a query is such a useful feature that many PROLOG implementations include built-in predicates that perform this directly. A common form of this is:

bagof(⟨***identifier***⟩, ⟨***query***⟩, ⟨***identifier***⟩)

Such a literal repeatedly evaluates the ⟨*query*⟩ expression until no more solutions of it are found, at which time the next rightmost goal is tried. Each time a successful solution is found, the value given to the first variable is appended to a list bound to the second variable.

Note that the name **bagof** is deliberate. The resulting list of arguments is not a set; duplicate solutions are not removed. If the user wishes duplicates removed, the next goal literal can be one that does the filtering.

Finally, the ***repeat*** predicate, defined as follows, permits iterative loops to be executed:

repeat.
repeat:–repeat.

The first time through the predicate, the result is always true. If a backtrack occurs to a **repeat,** the second clause form restarts the goals to the right one more time.

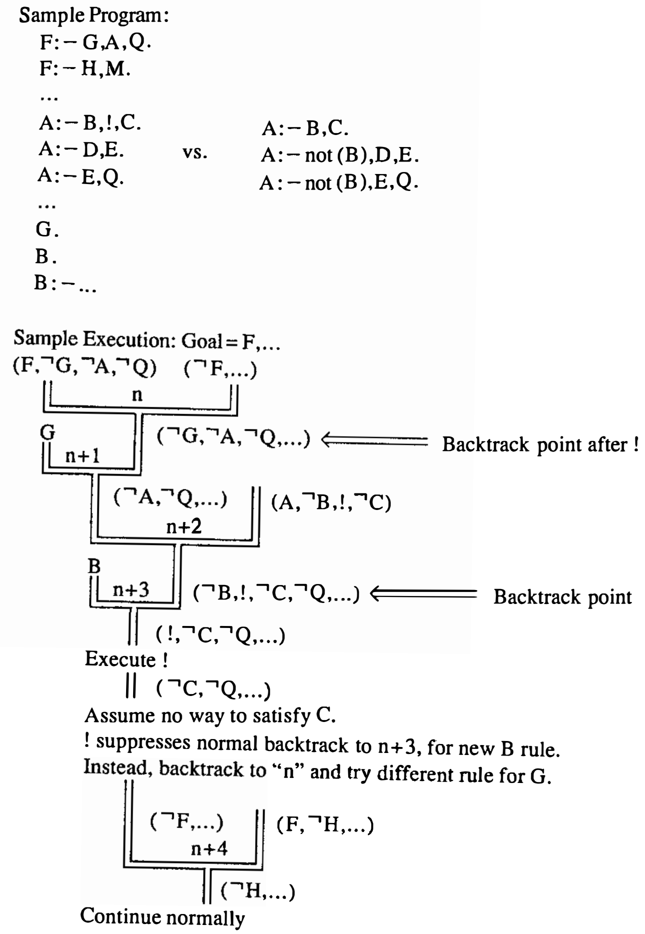


FIGURE 16-21
Sample operation of cut predicate.

A typical use of this predicate is in the form

p(...):–q(...),...,repeat,r(...),s(...).

where **r(...)** represents the body of some loop that we want executed until **s(...)** is true. If **r(...)** fails, **repeat** forces it to start over again. We will never backtrack into the **q**'s. If **r(...)** succeeds, and **s(...)** fails, then a backtrack restarts **r.** If both succeed, the clause completes successfully.

16.5.4 Extralogical Predicates

Finally, most PROLOG implementations include predicates which are decidedly nonlogical, but which provide the language with enough features to be self-modifying. These predicates assume that individual statements in the program can be transformed into clauses and written as expressions of some sort. For example, one technique might represent a clause of the form

p(*x*,*y*,**Roy**):−**q**(*x*,3),**r**(**Tim**,*z*),**s**(*x*,*y*,*z*).

as the term :−(**p**(*x*,*y*),(**q**(*x*,3) **r**(**Tim**,*z*) **s**(*x*,*y*,*z*))). ":−" is a functor symbol for a function of two arguments. The first is a term representing the head literal; the second is a list of terms representing the goals.

Such a representation permits the contents of the current program to be examined, added to, and modified arbitrarily. It is the basis for many of the extensions of PROLOG whose interpreters are themselves written in PROLOG.

The first of these predicates is ***clause***. This takes two arguments and succeeds if it can find a clause in the current program whose head unifies with the first argument and whose body matches with the second. If the matching statement in the program has no body, the second argument is unified with the predicate **true.**

Thus, with the above example, **clause**(**p**(*a*,*b*,*c*),*q*)) would succeed with the substitution [*x*/*a*,*y*/*b*,**Roy**/*c*,(**p**(*x*,*y*),(**q**(*x*,3) **r**(**Tim**,*z*) **s**(*x*,*y*,*z*))/*q*].

The first time **clause** is encountered in a goal-list, PROLOG will start the search through its program from top to bottom, in exactly the order that it uses when searching for a statement that matches a goal. It returns the first such entry it finds. When asked to backtrack, it finds the next clause that matches the given arguments.

The next special predicate is ***assert*** (of which there are several forms). This predicate takes one argument which must be instantiated to a valid clause (either a rule or fact). This clause is added to the program for immediate use. Note that if backtracking occurs to the **assert,** the added clause is not retracted.

The ***retract*** predicate is the opposite. It will search the program for the first clause that unifies with its argument, and remove that clause from the program. Leaving uninstantiated variables in the argument permits later predicates to examine the clause that was removed. Again, backtracking does not reinsert the removed clause but picks up from where it left off, looking for the next matching clause and deleting it.

16.6 SOME EXAMPLES

The following subsections describe some sample PROLOG programs chosen to demonstrate some aspect of the PROLOG inference engine. The first example, symbolic arithmetic, shows the power of PROLOG's unifier when dealing with arguments involving function applications. The second example, the Towers of Hanoi, demonstrates recursion and s-

expressions. The third example demonstrates how a fairly complete PROLOG interpreter can be written and executed in PROLOG itself.

16.6.1 Symbolic Arithmetic

Section 4.10 included a discussion of how the positive integers can be described by simple syntatic terms in lambda calculus. A 0 is represented by the constant symbol ***zero***, which has no "meaning" per se to the system. The integer k is represented as the term built up by applying some function symbol **s** to **zero** k times. Thus 5 is represented as **s(s(s(s(s(zero))))).** Further, with this syntax, some very simple lambda calculus functions (which only know how to apply substitutions into expressions) can duplicate the operations of normal arithmetic functions such as "+," "×,"....

The same kind of thing can be done in logic in general and with PROLOG in particular. Figure 16-22 diagrams the PROLOG equivalent of the lambda calculus functions. The reader should mentally execute some sample goals, such as "**factorial(s(s(s(zero))),***y***)**" and determine that the correct answer is derived (*Hint:* it's 6). Also note that this representation is perfectly amenable to reverse computations, such as:

?−**factorial(***x***,s(s(s(s(s(s(zero))))))).**

16.6.2 The Towers of Hanoi

(Clocksin and Mellish, 1984, p. 147)

The Towers of Hanoi problem was introduced in Section 3.3.3 as an example of a problem that is nicely solvable by a recursive function. Figure 16-23 gives a short PROLOG program that implements this function and uses many of the features described previously.

The main predicate is **hanoi,** and it accepts a single argument representing the number of disks on the leftmost pole and returns as a second argument a list consisting of "(*from.to*)" terms which indicate the

```
add(zero,y,y).
add(s(zero),y,s(y)).
add(s(x),y,s(z)):− add(x,y,z).

mult(zero,y,zero).
mult(s(x),y,z):− mult(x,y,zl),add(y,z1,z).

divide(x,y,q,r):− mult(q,y,r1),add(z1,r,x).

factorial(zero,y,zero).
factorial(s(x),y):− factorial(x,z),mult(s(x),z,y).
```

FIGURE 16-22
Integer arithmetic via unification.

```
hanoi(n,z):− move(n,left,center,right,nil,x), reverse(x,z).

move(0,_,_,_,z,z):−!.
move(n,a,b,c,z,z2) :− m is n−1,
                      move(m,a,c,b,z,z1),
                      move(m,c,b,a,((a.b).z1),z2).
```

FIGURE 16-23
A solution to the Towers of Hanoi.

sequence of moves to make. To "prove" this predicate, we use a predicate **move,** which has six arguments. The first is the number of disks left to move, the second is the name of the pole where these disks are found, the third argument is the name of the pole where the disks should end up, and the fourth argument is the name of the spare pole. In the rule for **hanoi,** these poles are given the constants **left, center,** and **right,** accordingly.

The fifth argument in **move** is a list (in reverse order) of whatever moves were necessary to get to the current state (as specified by the first four arguments). In the original query this is nil, because no moves have been made yet. The last argument is a list of moves (again in reverse order) which contains all the moves needed to solve the puzzle, assuming that the fifth argument's moves are part of them.

The implementation for **move** involves two rules, the first of which says that if you ever get to the state of 0 disks left to move, stop. The cut says that this is the only solution the program is designed to find, so asking for another will return false.

The second rule says that to move n disks, you move $n-1$ disks to the spare pole, move the bottom disk to the destination, and then move the $n-1$ disks from the spare to the destination. Note the use of an intermediate variable *z1* to represent the solution to the first $n-1$ disk problem, and the construction of the initial move list for the second $n-1$ problem by appending on the single disk move.

16.6.3 Self-Interpretation

One of the unique characteristics of abstract programs in general (and variations such as pure LISP in particular) is that it is possible to express an interpreter for the language in itself very elegantly. Observing such programs can give important insight into how the underlying computing systems actually work and how they could be extended to handle new features. Chapter 6 covered such self-interpreters for functional languages in detail.

With some exceptions, it is possible to write such an interpreter for a large subset of PROLOG in PROLOG. The exceptions usually have to do with the handling of predicates that have side effects affecting the order of execution.

Figure 16-24 gives one form of this program. The predicate ***system*** is a built-in that is satisfied if its argument is a goal having a predicate name that is built into the system.

The execution of the interpreter is fairly direct. Given a query of the form **interpret(⟨*body*⟩)**, PROLOG will look for the unique rule handling the argument. If it is the single predicate **true** with no arguments, the goal is satisfied. The goal is also satisfied if its argument is an empty list of goals. If the argument is a nonempty list of goals, each goal must be shown true, in order. If the argument is a goal having a system predicate, the goal is executed directly, with the truth or falsity coming from that execution. Finally, if the argument is none of the above, the argument must then be a single literal defined as the head of some clause in the program. The built-in **clause** then looks for the first statement whose head matches that of the goal. If one exists, a copy of the body (with all substitutions from the head unification appropriately made) is bound to the variable *body,* and the interpreter is reentered with the body as the new goal or goals.

Note that all the statements but the last include cuts to control execution. If a goal matches one of these rules, the cut indicates that that is the only such rule. If the body to the right of the cut fails, the goal must fail.

This is not true of the last rule. Here we want backtracking to **clause** to occur if necessary and to sequence through the program's statements just as PROLOG would have done.

We will use modifications of this interpreter to explain other logic languages later on such as in Chapter 21.

16.7 PROBLEMS

1. Consider defining the relation **is-uncle-of.** Assume that the only relations available are **is-father-of, is-mother-of, is-sister-of, is-brother-of, is-wife-of,** and **is-husband-of.** Assume that the spouse of an aunt also qualifies. Do the following:

```
; interpret(goal-list) interprets a list of goals
interpret(nil):−!. ; Stop on no goals
interpret(true):−!. ; Stop on a single true predicate

interpret((goal1.goal-list)):−!, ; To interpret multiple goals
  interpret(goal1), ; Interpret leftmost one first,
  interpret(goal-list). ; Then the rest
interpret(goal):−system(goal),!,goal. ; system does builtins

interpret(goal):− ; To interpret a single goal
  clause(goal,body), ; Find a clause with unifying head
  interpret(body). ; Then interpret body
```

FIGURE 16-24
PROLOG interpreter in PROLOG.

a. Write a description of this relation in English.
b. Convert the description to a single wff in first-order predicate logic, showing all quantifiers, and which variables are bound by which quantifiers.
c. Convert into clausal form, with each clause having unique variables from other clauses. Show all quantifiers and intermediate steps.
d. Convert into a PROLOG program.

2. Repeat Problem 1(d) assuming that you develop and use the auxiliary relations **is-parent-of** and **is-spouse-of.**
3. Write a set of PROLOG rules to implement the **equal** function between two lists, i.e., true if the lists are equal and false otherwise.
4. Redo Figure 16-8 assuming that the program was reordered in the following fashions. Count the number of inferences made and the maximum depth of the stack. How many times does shallow and deep backtracking occur?
   a. 3, 2, 1, 6, 7, 4, 5, 8
   b. 8, 7, 6, 5, 4, 3, 2, 1
5. There is a small problem with Figure 16-5 when *rule* is a fact. Fix it.
6. Repeat Figure 16-8 assuming that we add a ninth clause **E:−G.**
7. Write a PROLOG program to compute the union of two lists of atoms, where no atom shows up more than once in the result. Feel free to use the **member** rules given in the text. Assume that each of the lists has no duplicates within itself and that the order in which items appear in the final list is unimportant.

   What makes lifting either of the latter two simplifications complicated?
8. Write a PROLOG program to define a predicate **pairlist**(x,y,a,z) that is true when z is a list consisting of appending to a the reverse of a list of pairs from matching elements of lists x and y. No error checks are necessary.

   For example, for x=(1 2), y=(F G), a=((3.T)(4.N)),

   z=((2.G)(1.F)(3.T)(4.N))
9. For Figure 16-10, determine at what time the stack achieves the last state shown, that is, when the answer is available.

   If we asked for the next solution, at what time would the system respond, and with what?
10. Construct the deduction tree corresponding to the state of the computation when the first solution in Figure 16-10 is obtained (time 67). Number the closed nodes.
11. Consider a small propositional-based logic language that acts like PROLOG in its proof derivations. Assume that a program in this language is a list (s-expression format) in any order of facts or rules, where a fact is a list with one atom in it—the atom to assume true—and a rule is a list of two or more atoms where the first is the consequence and the rest are the conditions. All condition atoms must be proved true before inferring that the consequence is true.

    Write in abstract syntax a function **provelist**(*goal-list,kb*) which returns TRUE if all the atoms in the list *goal-list* can be proved true using the rules and facts in the "knowledge base" list *kb*. (*Hint:* It may be useful to write a companion function **proveone**(*goal,kb,kbx*), which proves that the single atom *goal* is true using whatever rules are left in *kbx*.)

    As an example, consider *kb*=((A)(D B C)(B A)(C)) corresponding to the wff **A ∧ (B ∧ C→D) ∧ (A→B) ∧ C.** Verify that your functions prove **A ∧ D** is true, i.e., **provelist((A D),KB)** returns TRUE.
12. Modify Figure 16-11 to return either nil or a list where each element is an association list that binds variables from the queries to the values assigned by one of the solutions. Thus the length of the object returned by **prove** corresponds to the number of solutions to the original query.

    Assume that an auxiliary function **free-vars** can take a query and return a list of the variables used in the query.
13. Repeat Problem 12 for the environment-stacking inference engine of Figure 16-12.
14. Construct a flowchart form of the environment-stacking inference engine.
15. Describe briefly the commonalities and differences between functional languages such as SASL and HOPE with logic languages like PROLOG. (Consider their basic computational model and how they might be implemented.)
16. Write a PROLOG program permitting queries of the form

    ?−**sentence**((the student solved the problem)).

    where **sentence**(x) is true if the elements of the list x represent a valid English sentence. For this problem assume an English grammar of the form:

    ⟨sentence⟩:=⟨nounphrase⟩⟨verbphrase⟩
    ⟨nounphrase⟩:=⟨adjective⟩⟨noun⟩
    ⟨verbphrase⟩:=⟨verb⟩|⟨verb⟩⟨nounphrase⟩
    ⟨noun⟩:=student|problem|test|apple
    ⟨verb⟩:=solved|ate
    ⟨adjective⟩:=the|a|an

    Hint: Remember that a statement of the form **p**((x.y)):− . . . will match any goal where the predicate symbol is **p** and the argument is a list, where the car is bound to x and the cdr is bound to y.

    Show the execution of your program for the specified query.
17. How many answers would your program from Problem 16 return (and in what order) for the query:

    ?−**sentence**((x y ate x apple)),**member**(x,(the a)).

CHAPTER 17

THE WARREN ABSTRACT MACHINE

In Chapter 1 we discussed the idea of an ***abstract machine*** to use as a semantic model of how a program carries on some computation. For functional programming we saw one famous example of this, the ***SECD Machine***. Such a model also exists for logic programming: The ***Warren Abstract Machine*** or ***WAM*** has in fact grown to the point where it serves as the basis for most high-performance implementations of PROLOG and similar languages. This includes use as an intermediate language for a compiler from PROLOG to conventional machines, as the basis for entirely new computer architectures that support PROLOG features directly, and as a starting point for implementations of non-PROLOG logic languages. Figure 17-1 diagrams some of these combinations.

Because of this widespread use, any reader interested in the implementation of logic languages is advised to study this chapter carefully.

The primary origins of the WAM model come from David H. D. Warren's Ph.D. thesis work (Warren, 1977), which he and others (such as David S. Warren—no relationship) have expanded on since then to the point where both real compilers and real machines have been built, and detailed measurements taken. The original references to these descriptions include Warren (1983), Tick (1983), Dobry et al. (1985), and Dobry (1987a).

In brief, this architecture is von Neumann-like, with sequentially executed instructions modifying data found in a randomly accessible memory. Like the SECD Machine, there is a strong reliance on stacks and a memory structure that supports tagged cells containing pointers. A variety of procedure call instructions implement directly the sequencing philosophy assumed by the PROLOG inference engine as it cycles through the goal lists and program clauses. Other instructions provide a wide variety of unification tests in ways that permit automatic compilation of highly optimized code. Arguments in goal literals are passed through a set of ***argument registers*** accessible to most instructions, which in turn are very similar to the ***general registers*** found in many modern architectures. Other specialized machine registers give access to the major data structures in memory.

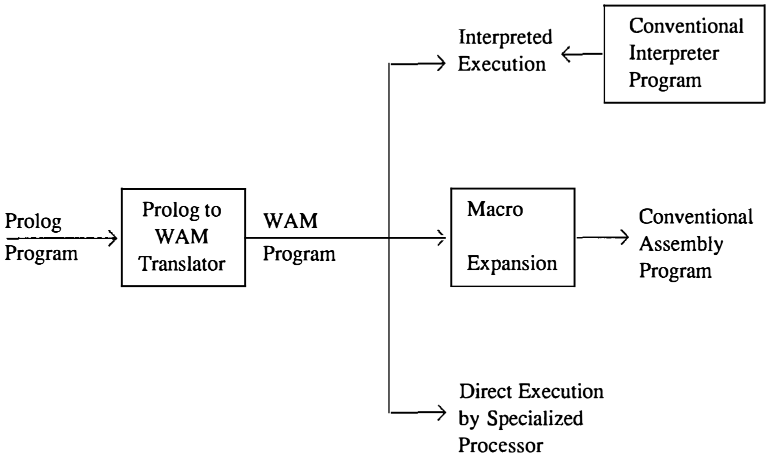

Prolog Program
Prolog to WAM Translator
WAM Program
Interpreted Execution
Conventional Interpreter Program
Macro Expansion
Conventional Assembly Program
Direct Execution by Specialized Processor

FIGURE 17-1
PROLOG execution using the WAM model.

The general model of execution assumes that the argument registers mentioned above contain the actual arguments for the current goal literal, and these values are successively unified against the formal argument expressions (as described by short program code sequences) in the appropriate program clause heads. When a match occurs, new argument values are built (in the same argument registers) for the first literal in that clause's body, and the process is repeated for that goal. Success in solving that goal causes code to be executed that builds the arguments for the next goal on the right-hand side. A linking mechanism keeps track of which literals are left to be treated as goals, and in what order, after the current goal is proven successfully. Saving copies of the argument and other registers in a memory stack permits failure and backtracking operations to restart with a previous goal as required.

The rest of this chapter describes a simplified WAM architecture by starting with a correspondence between real PROLOG programs and the general structure of resulting compiled code. We then discuss the key data structures assumed by the architecture, and then a deliberately simplified version of the WAM instruction set itself. Chapter 18 analyzes opportunities for improvement in this simplified WAM, and uses these to introduce features and modifications that appear in most real WAM models. A detailed example is used throughout to show how a real PROLOG program would be compiled into WAM code (both simplified and optimized) and how that program would execute on a real machine.

To the reader interested in understanding the implementation of logic programming systems, this chapter is perhaps the most important of the book, and should be read carefully.

17.1 PROGRAM STRUCTURE

The general structure of a compiled program in the Warren architecture mirrors closely the original PROLOG program (see Figure 17-2). For each of the original PROLOG statements there is a corresponding section of WAM instructions which handles the head unification for that clause and the sequencing through the goals called out on the statement's right-hand side.

All such code sections for clauses having the same predicate name in their head literal are chained directly together in the order in which the programmer entered the original text. The chaining is via instructions at the beginning of each section. This permits the computer to rapidly find the next clause to try if one clause fails. Some instructions at the beginning of the chain set up for the chain and handle other housekeeping chores.

Together, each linkage of sections acts as a single ***procedure,*** tailored specifically to handle any goal whose predicate symbol matches that for its internal clauses. The internals of the procedure step through the appropriate statements in the expected PROLOG order, with calls to other such procedures as needed as goals are processed. All such calls are to the entry code of a procedure segment where the necessary initialization is performed.

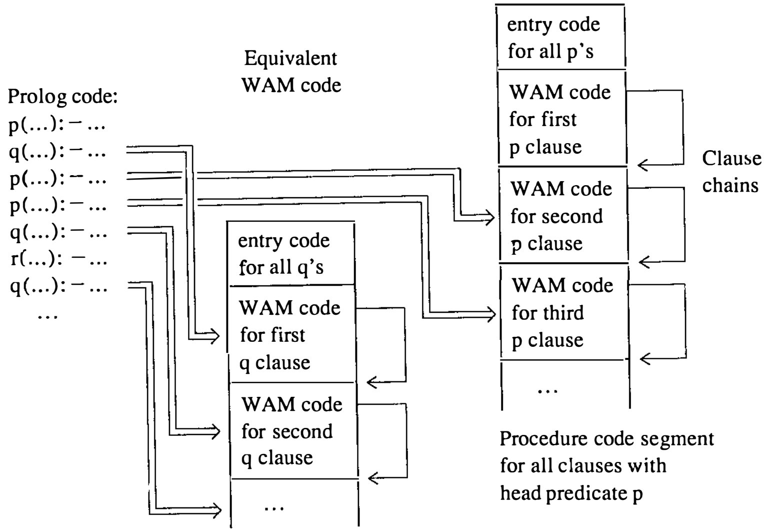

Prolog code:
p(...): – ...
q(...): – ...
p(...): – ...
p(...): – ...
q(...): – ...
r(...): – ...
q(...): – ...
...
Equivalent WAM code
entry code for all p's
WAM code for first p clause
WAM code for second p clause
WAM code for third p clause
...
Clause chains
entry code for all q's
WAM code for first q clause
WAM code for second q clause
...
Procedure code segment for all clauses with head predicate p

FIGURE 17-2
Simplified Prolog-to-WAM translation.

17.1.1 Code for One Clause

Within a procedure it is possible to link the individual code sections in orders other than the simplistic way shown here. In many circumstances this can significantly improve performance by avoiding entirely clauses which are known beforehand not to work for certain goals. Some of the initial entry code handles this, and will be discussed later under optimization techniques.

17.1.2 An Individual Code Section

Figure 17-3 looks a little deeper at the structure of a code section corresponding to one clause. Starting out the segment is initialization code which sets up the machine's major data structures to permit trying the

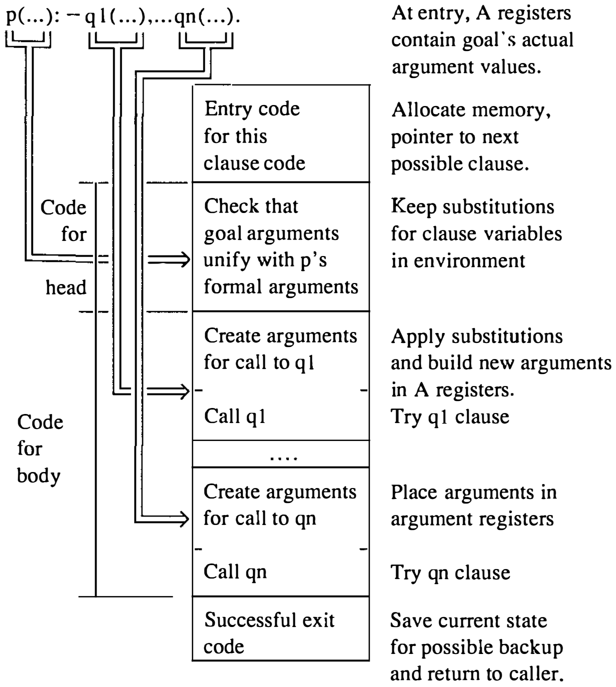

p(...): – q1(...),...qn(...).
At entry, A registers contain goal's actual argument values.
Entry code for this clause code
Allocate memory, pointer to next possible clause.
Code for head
Check that goal arguments unify with p's formal arguments
Keep substitutions for clause variables in environment
Create arguments for call to q1
Apply substitutions and build new arguments in A registers.
Call q1
Try q1 clause
Code for body
....
Create arguments for call to qn
Place arguments in argument registers
Call qn
Try qn clause
Successful exit code
Save current state for possible backup and return to caller.

FIGURE 17-3
General structure of clause code.

clause. This includes saving a pointer to the next clause with the same predicate name if backtracking occurs out of this one.

Following this is code that checks that the formal argument expressions of the clause are in fact unifiable with the actual arguments in the current goal. These actual arguments are always found in the ***argument registers,*** denoted A1 through An, with Ak containing the k-th argument. The unification checks are performed one argument at a time by separate instructions that test the A registers. A mismatch in any of these tests causes this code sequence to be aborted and control transferred to the next appropriate code segment (as set up by the entry code). This is sometimes termed ***shallow backtracking,*** because the amount of information that must be reset is small.

In the process of doing these unification checks, it may be necessary to assign values to formal variables in the clause. This is done by allocating a memory location to each variable in the clause and storing a value as appropriate during the unification checks. The set of memory locations covering the variables for a clause is called its ***environment*** and is kept on an internal stack. Since rules can be recursive, each attempted use of a clause as a goal has its own distinct environment.

Storing values bound to clause variables in the new environment represents only part of a potential substitution generated by a unification. Bindings to variables found in the goal's arguments (as indicated by the argument registers) is performed by storing backward into the environment that was active at the time the goal was built.

Once all arguments have unified successfully, control in the WAM program passes to a series of instructions that mirror the body of the original statement. There is a series of instructions for each goal, in the order in which the goals were written. These instructions are of two kinds. The first type takes the current substitutions for clause variables (as recorded in the environment) and creates the actual arguments (in the argument registers) to be presented to the code for the new goal's predicate symbol.

The second type actually performs the transfer of control to the entry code for the goal's predicate. That code saves any machine state information that might have to be reloaded if a return is necessary to the current code, plus initialization for the new predicate's clauses.

Note also that during the unification instructions, substitutions may be made into variables in various goals' arguments. Because it may be necessary to "retract" these substitutions if the current clause fails, the program keeps a list of all uninstantiated (unbound) variables which received values during each clause. A failure in a clause causes all variables on this list to be relabeled back to "uninstantiated," and their assigned values removed.

Successful execution of the new code for some goal will eventually result in control being passed back to the original code that called it, with the original machine state largely restored (except, of course, for appropriate changes to the environment). Here, the whole process of argument construction and control passing is duplicated for the next goal. Note that if any of this clause's arguments were given substitutions during the prior-called code's execution, these substitutions will be available in the environment.

A failure in the called code to find any matching clauses at all will cause a ***backtrack*** into the caller's code to look for another alternative for a prior goal. Because of the amount of state information that must be reset, this is often called a ***deep backtrack.***

Successful completion of the code segment for the last goal in a clause causes return to the section's exit. This code does whatever storage housekeeping is necessary before returning to the code that called the procedure in which this section is embedded.

The resemblance between this execution sequence and that of a typical procedure call mechanism in a conventional programming language is more than superficial. Transferring control to the code for a new goal is very much like a conventional subroutine call. The current state of the machine (primarily registers and return address) is saved in a single unit on a stacklike data structure, and control is transferred to the start of a new procedure body. Arguments to this new procedure are available by convention in predefined places (argument registers in this case). Storage for local variables is made available on the stack as part of the topmost choice point. Upon completion of the new routine, the information needed to return to the calling code is available on the top of the stack. The primary difference is that, unlike a conventional procedure return, a return in the WAM does not free up the stack space allocated to the call. This is because of PROLOG's backtrack mechanism, which may require restarting the procedure later if a failure is detected in some other clause.

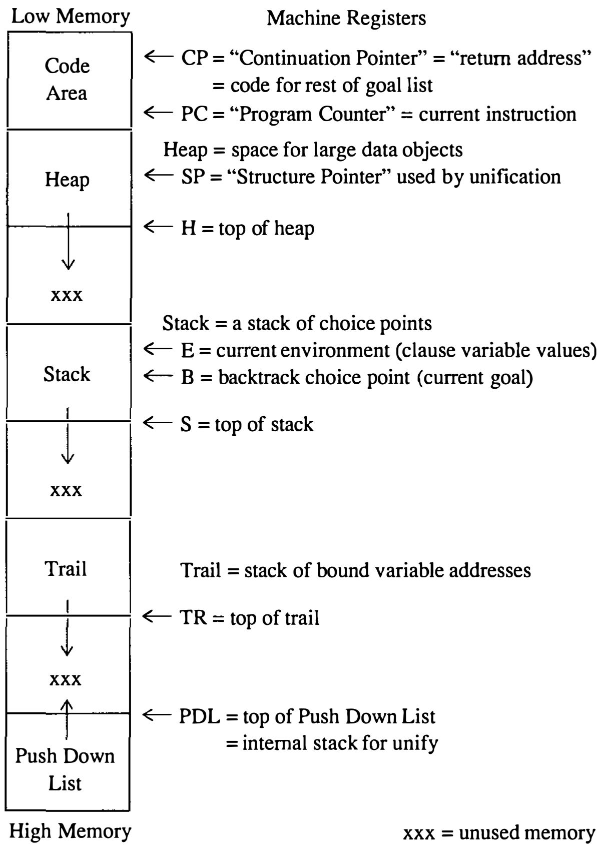
17.2 MAJOR DATA STRUCTURES AND STATE REGISTERS

The WAM model matches fairly directly a conventional von Neumann computer. There is a memory array of individually addressed locations, and a CPU which contains registers for addressing the memory and which executes instructions one by one out of this memory.

The typical WAM program views memory as divided into several major areas, with hardware registers and instructions available to manipulate each in specialized fashions. Figure 17-4 lists these areas and the basic machine registers that address them. The ordering of these areas from low to high addresses matches Warren's 1983 description and is chosen to simplify some of the optimizations described later.

The full WAM model has a few other registers and subdivisions of memory not shown here. The WAM model we will discuss first uses this simplified set to aid in clarity of understanding. In the next chapter we introduce modifications as appropriate.

The first memory area is devoted to program code. Instructions are


Low Memory
Machine Registers
Code Area
← CP = "Continuation Pointer" = "return address" = code for rest of goal list
← PC = "Program Counter" = current instruction
Heap
Heap = space for large data objects
← SP = "Structure Pointer" used by unification
← H = top of heap
xxx
Stack
Stack = a stack of choice points
← E = current environment (clause variable values)
← B = backtrack choice point (current goal)
← S = top of stack
xxx
Trail
Trail = stack of bound variable addresses
← TR = top of trail
xxx
← PDL = top of Push Down List = internal stack for unify
Push Down List
High Memory
xxx = unused memory

Other Registers:
A1,...An = Argument registers used to hold actual arguments.
Mode Flag = Flag used by unify instructions.

FIGURE 17-4
Major memory areas for the simplified WAM.

fetched from this area one at a time as indicated by the ***Program Counter*** or ***PC register***. These instructions are part of the code section for a clause for some procedure.

Successful completion of the code section for some clause means that a goal built by some other clause's right-hand-side code has been successful, and the machine should "return" to that point in the right-hand side and resume execution. The ***Continuation Pointer*** or ***CP register*** points to this location. When combined with the appropriate environment of bindings, this is virtually identical to the concept of a ***continuation*** described earlier for functional languages.

The typical WAM instruction calling a procedure sets the CP to one instruction after the call, permitting a return to the caller upon successful completion of the called procedure. Thus, the CP also corresponds almost exactly to a subroutine return linkage register in a conventional computer.

The next major data area is the ***heap,*** which contains structures and lists built during the unification process. These objects are typically too large to fit in either an argument register or a single environment cell. The ***Structure Pointer*** or ***SP register*** steps through the components of such objects during unification. Storage here is allocated dynamically as needed, with pointers to them left where needed. The ***H register*** indicates the top of the allocated part of the heap.

Perhaps the most important data structure is the ***stack,*** which holds call/return and environment information for sequencing through the code segments corresponding to the clauses. The information for each attempt to solve a goal is called a ***choice point,*** with the ***B register*** (B for backtrack) pointing to the most recently created one and the ***E register*** (E for environment) pointing to the one created when the clause code currently pointed to by the PC was entered. The ***S register*** points to the current stack top from which new choice points will be built. (Note that the original literature called this the A register; we changed it here to avoid confusion with the argument registers.)

The relationship among the PC, B, and E registers is worth repeating. At any point in time the PC points into the code for some clause, and the E register gives access to the current values for variables in that clause. The B register points to the most recently created choice point and may be equal to or greater than (i.e., earlier than) E. It is equal to E just as the code for the right-hand side of some clause is entered and is greater as goals in that clause's body are solved successfully. The stacked choice points between E and B in the latter case reflect goals that have been solved in the process of handling the right-hand side of E's code. The example later in Section 17.8 will demonstrate this more fully.

The ***trail*** is a stack of locations containing references to variables that have received values at some point during execution (i.e., are ***bound*** or ***instantiated***), and may have to be "unbound" of these values at some point in the backtrack process when a goal is found that is not solvable. The ***TR register*** points to the top of this area where new trails can be pushed.

In some more modern versions of the WAM, the trail is positioned as shown in memory but grows down toward the top of the stack.

The last memory area, the ***PDL*** or ***push-down list,*** is a small stack used by the unification instructions to save information during the unification of complex objects. The ***PDL register*** points to the top of this stack.

Finally, a one-bit flag register, the ***mode flag,*** is also included in the WAM CPU model. This flag can signal that the WAM is in one of two modes, ***read mode*** or ***write mode.*** These modes are set and used by certain instructions during head-goal unification.

For convenience, the S, TR, H, and PDL registers are all assumed to point to the next available (unused) cell in their areas. Clearly, choosing to make them point to the most recently used locations is a perfectly viable alternative.

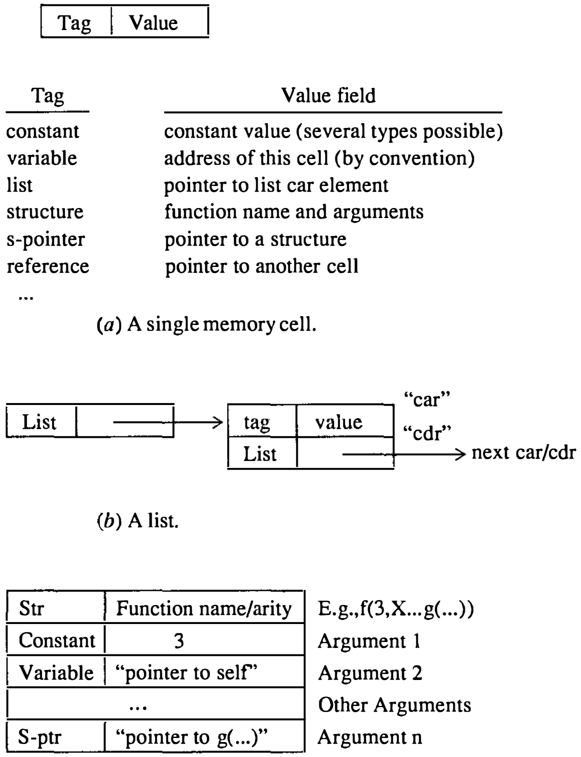
17.3 MEMORY WORD FORMAT

Memory in the WAM model is a linear array of individually addressable locations. The only real variation from a conventional computer is that the format of each location looks somewhat like that from the SECD Machine. The bits associated with a location are divided into two parts, a ***tag*** and a ***value***. The tag identifies how to interpret the value and may have several different combinations as listed in Figure 17-5.

The argument registers mentioned earlier are also big enough to hold a single tag and matching value, and are assumed to be in the same format.

A ***constant*** tag means that the value field holds some number, character, or other real value.

A ***variable*** tag indicates that this cell corresponds to a logical variable that has not yet been given a value by a unification. For implementation convenience, its value field often equals its own address.



Tag	Value field
constant	constant value (several types possible)
variable	address of this cell (by convention)
list	pointer to list car element
structure	function name and arguments
s-pointer	pointer to a structure
reference	pointer to another cell
...	

(*a*) A single memory cell.

(*b*) A list.

Str	Function name/arity	E.g.,f(3,X...g(...))
Constant	3	Argument 1
Variable	"pointer to self"	Argument 2
	...	Other Arguments
S-ptr	"pointer to g(...)"	Argument n

(*c*) A structure.

FIGURE 17-5
Representation of objects in memory.

A ***list*** is virtually identical to that supporting general s-expressions as discussed in an Chapter 00. The value of a word tagged as a list is a pointer to a two-entry ***list cell*** block in memory, the list's ***car*** and ***cdr*** components, respectively. In the WAM this takes up two consecutive memory locations, each of which can be an arbitrary object. In particular, a component with a tag of type list means that the entry is itself a pointer to another two-word block containing another list cell. Such an entry is typically found in an argument register to signify a list, and as the cdr of a list cell that is part of a list, and points to the next list element.

In the WAM model, lists are built out of pairs of words allocated from the heap.

A ***structure*** corresponds to a syntatic term involving a function symbol and its arguments [e.g., g(x,31)]. Since it is usually impossible to encode all such information into a single word, such an object also takes up multiple consecutive words in memory. The first word has a tag of type ***structure,*** and contains an encoding of the function symbol and the number of arguments (the function's ***arity***). Following this are a series of words, one for each argument in the expression. Thus, for g(x,31) there are three words, the first describing the name of the function g and denoting that it has two arguments, the second corresponding to the first argument x, and the third corresponding to the second argument 31.

As with lists, neither an environment variable value cell nor an argument register can contain all these words. Consequently, there is a tag of type ***structure pointer,*** which indicates that the object in question is a structure that can be found starting at the designated memory location. This fits conveniently in an argument register or in a word reserved for a component of a list (or other structure) [cf. Figure 17-5(*c*)].

Also as with lists, structures are usually found in storage allocated from the heap area of memory.

Finally, a ***reference*** tag indicates an indirect pointer to some other cell. This is used to chain objects together (e.g., by giving variable x a reference to the location for some variable y, we are saying that "variable x will have the same value as variable y whenever y is assigned a value"). In many ways it is like the ***invisible pointer*** type used earlier to support parameter passing and garbage collection.

In some implementations the tag for reference is the same as that for variable, with the contents of the value field distinguishing between the two. A value field that points to itself is an unbound variable; otherwise it is a reference. Figure 17-14, later on in this chapter, gives some other complex examples.

For simplicity, the notation "x#y" will denote the contents of some cell, where the tag is "x" and the value is "y." A value field of "*" denotes the address of that cell. Thus "var#*" stands for a cell representing an unbound variable.

17.4 SIMPLIFIED CHOICE POINT

The major data structure controlling program execution is the ***choice point***—a set of contiguous locations on the main stack. These objects are built during the execution of the entry instruction in a code sequence for a predicate symbol, modified by entry code for each clause in that chain, and discarded by failing the last clause for that predicate. They closely resemble a ***frame*** built by a conventional call instruction in an architecture oriented toward classical languages such as Pascal, and contain copies of the various machine registers needed to restart a clause's code under various conditions.

At any point in a program's execution there is one choice point for each goal currently still active, piled up in linear order on the stack. Thus, when a successful solution is found (the equivalent of the emptying of the goal-list), the stack contains in the choice points a complete history of the individual resolutions (and matching unification substitutions) used to derive it.

Figure 17-6 diagrams this structure, along with indications of when each piece is created, modified, or deleted from the stack.

In particular, the information found in a simplified choice point includes the following:

- A copy of the argument registers A1,...,An as they were when the choice point was constructed. This corresponds to the arguments of the goal literal that the choice point represents and permits the argument registers to be reloaded to their original values if one clause fails after changing some of them and a new clause is to be tried against the same goal.

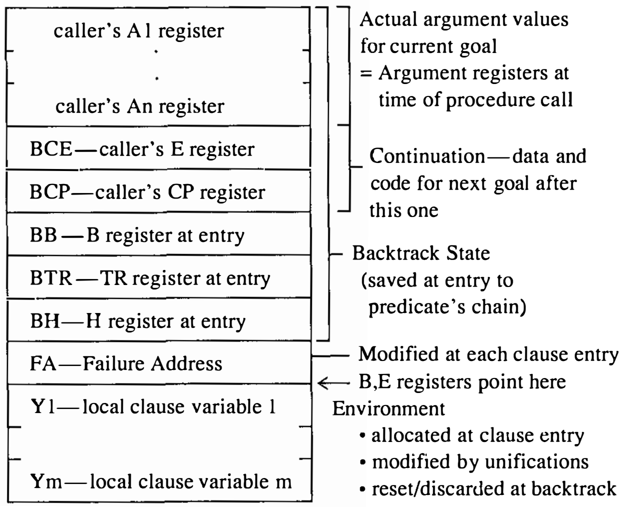


FIGURE 17-6
The choice point and environment.

- Where to return to if the goal represented by this choice point is solved successfully. As with functional languages, this is called the ***continuation*** of the current goal, and consists of
  - The instruction to return to in the caller's program (called the ***Backtrack Continuation Pointer or BCP*** entry because it is initialized to the CP register value when the choice point was built). The instruction addressed by this value is the beginning of the code for the next goal to solve in left-to-right order.
  - The environment to use with the caller's program (the ***Backtrack Continuation Environment or BCE*** entry). This is the address of the choice point that was built when the caller's program was started, and contains the values associated with that clause's variables (the values needed to build the set of arguments for the next goal at BCP).
- The address of the code for the next clause to try if the current clause fails. This is the ***FA*** (for ***failure address***) entry, and is the primary information needed by the ***shallow backtrack*** process.
- The state of the main memory data structures at the time the choice point was built, namely, the top of the trail and heap, and the choice point in effect before this one [as stored in the ***BTR*** (***backtrack trail***), ***BH*** (***backtrack heap***), and ***BB*** (***backtrack B***) entries, respectively]. This is the primary information needed to perform a ***deep backtrack*** if no clause exists which satisfies the current goal. It represents how to get back to the last goal solved before the current one.
- The ***environment,*** or space (Y1...Ym) for the values to be bound to local variables for the clause currently being used against the goal represented by this choice point. There is one such cell for each variable in the clause. Also, when executing a particular procedure, the size of the topmost environment varies as different clauses are tried.

Finally, the notation ⟨***entry-name***⟩[X] refers to the contents of a specific entry in the choice point designated by X (usually B or E). Thus, BTR[B] is the memory cell labeled BTR in the choice point selected by B. It holds a copy of the TR register at the time the most recent choice point (B's) was built.

As a side note, the optimized WAM of the next chapter splits the data structure of Figure 17-6 into two pieces: one for goal and backtrack information, and one for environment variables.

17.5 SIMPLIFIED WAM INSTRUCTION SET

With the above introductions, we can now discuss a simplified WAM instruction set. A more complex, but higher-performance, version is the topic of Chapter 18.

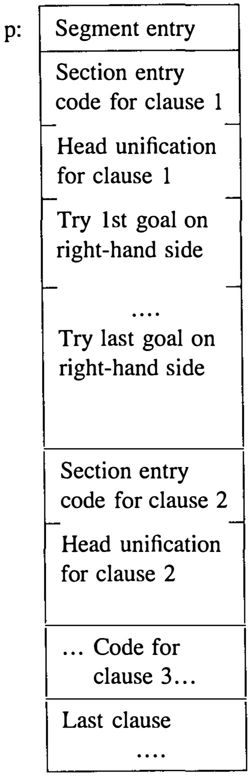
Based on their intended functions, both the simplified and the full set of WAM instructions partition naturally into five classes:

- ***Indexing instructions*** to control sequencing through the chain of code sections associated with one procedure (one head predicate symbol)

- ***Procedural instructions*** to control choice point and environment setup, and transfer of control from one chain to another
- ***Get instructions*** to verify that the formal arguments in a clause unify with current actual arguments (as recorded in the argument registers), and to record the appropriate unifying substitutions
- ***Put instructions*** to load the argument registers for the next goal on the right-hand side of some clause
- ***Unify instructions*** to handle gets and puts of complex objects such as lists and structures

The following subsections discuss these classes. To aid in understanding, Figure 17-7 diagrams the general way in which these instructions are strung together when used to form a code segment for an arbitrary PROLOG procedure. Also, Figure 17-8 gives a set of code compiled from this architecture for a PROLOG program to perform a classical ***quick sort*** on a list of numbers.

```
Assume: p(...):-q1(...),...qn(...). ; 1st clause a rule.
        p(...).                     ; 2nd clause a fact.
```


p:
Segment entry
Section entry code for clause 1
Head unification for clause 1
Try 1st goal on right-hand side
....
Try last goal on right-hand side
Section entry code for clause 2
Head unification for clause 2
... Code for clause 3...
Last clause
....

```
p:  mark           ;Build choice point
p1: retry-me-else p2 ;chain to 2nd clause
    allocate n     ;Space for clause variables
    get/unify      ;Do unification checks
      ....         ;against current A registers
    put/unify      ;Build arguments for 1st goal
      ....
    call q1        ;Call code for 1st goal
      ....
    put/unify      ;Build arguments for 2nd goal
      ....
    call qn        ;Call code for last goal
    return         ;Successful return to caller
p2: retry-me-else p3 ;Chain to 3rd clause
    allocate n     ;Space for clause variables
    get/unify      ;Do unification checks
      ....         ;against current A registers
    return         ;Successful return to caller
p3: retry-me-else p4 ;Chain again
      ....         ;Similar code for 3rd clause
pn: retry-me-else px ;Chain again
      ....         ;Similar code for last clause
px: backtrack ;No more clauses—backtrack
```

FIGURE 17-7
A generic WAM procedure code segment.

```
; qsort(left,right,array) is true when array=append(sort(left),right)
1 qsort(nil,right,right).
2 qsort((x.left),right0,right):- partition(left,x,left1,left2),
        qsort(left1,right0,(x.rightl)),
        qsort(left2,right1,right).
3 partition(nil,_,nil,nil).
4 partition((x.left),pivot,(x.left1),left2):- x<pivot,!,
        partition(left,pivot,leftl,left2).
5 partition((x.left),pivot,left1,( x.left2)):-
        partition(left,pivot,leftl,left2).
; Note Yi stands for the i'th local variable in environment at E.
qsort: mark                       partition: mark
  retry-me-else qs2                 retry-me-else pr2
  allocate 1                        allocate 1
  get-constant nil,A1               get-constant nil,A1
  getv Y1,A2                        getv Y1,A2
  getv Y1,A3                        get-constant nil,A3
  return                            get-constant nil,A4
qs2: retry-me-else exit             return
  allocate 7                   pr2: retry-me-else pr3
; Y1=X,Y2=L,Y3=R0,Y4=R              allocate 5
;Y4=R,Y5=L1,Y6=L2,Y7=R1             ; Y1=X,Y2=L,Y3=Pivot,
  get-list A1                       ; Y4=L1,Y5=L2
  unifyv Y1                         get-list A1     pr3: retry-me-else exit
  unifyv Y2                         unifyv Y1            allocate 5
  getv Y3,A2                        unifyv Y2            get-list A1
  getv Y4,A3                        getv Y3,A2           unifyv Y1
  putv Y2,A1                        get-list A3          unifyv Y2
  putv Y1,A2                        unifyv Y1            getv Y3,A2
  putv Y5,A3                        unifyv Y4            getv Y4,A3
  putv Y6,A4                        getv Y5,A4           get-list A4
  call partition                    putv Y1,A1           unifyv Y1
  putv Y5,A1                        putv Y3,A2           unifyv Y5
  putv Y3,A2                        escape less          putv Y2,A1
  put-list A3                       cut                  putv Y3,A2
  unifyv Y1                         putv Y2,A1           putv Y4,A3
  unifyv Y7                         putv Y3,A2           putv Y5,A4
  call qsort                        putv Y4,A3           call partition
  putv Y6,A1                        putv Y5,A4           return
  putv Y7,A2                        call partition
  putv Y4,A3                        return
  call qsort
  return
exit: backtrack
```

FIGURE 17-8
Simplified WAM program for **quick sort.**

In general, the instructions described below treat the WAM machine registers in a certain formalized fashion, namely:

- The B register always points to the topmost choice point on the stack.
- Once inside the code for some particular clause, the E register points to the choice point that was created when the procedure containing that clause was entered.

  The only time this may be the same as B is just as the code for a particular clause is entered, and before any goals on the right-hand side of that clause are tried. After that the E choice point is "buried" under choice points used to solve the right-hand-side goals.
- At the entry to a code segment for a predicate symbol, the CP register contains the instruction address to return to if a clause is found in the new procedure that unifies with the current goal, and has all of its right-hand-side goals fully satisfiable. The E register at this time points to the environment needed to continue execution at CP.
- Unless otherwise specified, each instruction increments the PC register to point to the next sequential instruction.

17.5.1 Indexing Instructions

Indexing instructions chain together and control the code sections for different clauses that have the same predicate symbol in their head. Of the instructions shown in Figure 17-9, the ***mark*** instruction is the first instruction encountered in a procedure (i.e., right after a call); it builds a choice point from the current contents of the machine's registers. (In the full WAM architecture this instruction's function is combined with variations of the retry). After execution, the B register is set to point to the new choice point, and the HB register set to the heap as it exists right now.

Instruction	Operation	Comments
mark	push Aregs to stack push E,CP,B,TR,H,PC + 1 B←S − 1	Build a new choice point up to the environment. Make it current one.
retry-me-else L	FA[B]←L	Set next clause to L.
backtrack	B←BB[B] fail	Discard current choice point and restart its caller.
Fail sequence: (part of backtrack and unification failure)		
	A1...An ← A1...An[B]	Reload argument registers.
	H←BH[B], S←B + 1	Reset heap, stack.

	while TR≠BTR[B] do
	TR←TR − 1
	V←memory[TR]
	memory[V]←V
		Reset all variables instantiated since choice point was built.

	PC←FA[B]	Branch to next rule.

FIGURE 17-9
Simplified indexing instructions.

This choice point will be the one to contain the environment for any clause code in the procedure.

The ***retry-me-else*** instruction is the first instruction for each clause code section. It modifies the FA entry in B's choice point to indicate the start of the code section for the next possible clause with the same predicate symbol. This address is provided as an argument to the instruction. Execution then continues with the next instruction after the retry. Note that at the time this instruction is executed, the B register always points to the same choice point built by the mark instruction which started the procedure code in which this section is imbedded. Also note that the full WAM architecture contains several variations of this instruction for use in different circumstances.

The address planted by a retry-me-else instruction in a choice point is used if the clause corresponding to the code following it fails for any reason (either there is a unification failure or the clause's right-hand side is not satisfiable). The failure mechanism to be discussed shortly will pick up this new address as a pointer to the next clause code to try. Usually the first instruction at this address is itself another retry-me-else, priming the same choice point for yet another clause if this new one does not work either.

Finally, the ***backtrack*** instruction is used as the target of the retry for the last clause of a chain. If control reaches the backtrack, then none of the clauses satisfied the current goal, and deep backtracking to the predecessor of the current choice point is necessary.

The implementation of this process consists of backing up one choice point (i.e., reloading B from the current choice point) and then doing the ***fail sequence:***

1. Reload the argument registers from B's choice point.
2. Reset the heap top to what it was when B's choice point was built (as indicated by the BH field in the choice point).
3. Use B to recompute the top of the main stack.
4. ***Unwind*** the trail stack by popping off entries (corresponding to addresses of clause variables) until the TR register reaches the value stored in B's choice point. For each entry popped off, the memory location corresponding to that variable is reset to a **variable** entry, indicating that it no longer has a value.

   How and why the trail stack gets "wound" to begin with is discussed later.
5. Branch to the code specified by the FA entry in the restored choice point. This is the next possible clause which might satisfy the goal signified by the choice point.

The net result of a backtrack is that we have totally deleted all storage associated with the failing goal (namely, the choice point and any val ues stored on the heap as a result of executing the failing procedure's code), ''undone'' any variable assignments made by that code to variables in other choice points, and reset the machine back to the next deeper choice point on the stack. From this point it can pick up and try a different clause for the prior goal.

17.5.2 Procedural Instructions

Procedural instructions (Figure 17-10) handle the management of environments and the transfer of control between chains of clauses.

The ***allocate*** instruction is typically the first instruction of the code section for a clause, and allocates space for all the clause's variables (as indicated by its single argument). In the version shown here, this allocation is on the stack right after the current choice point (as designated by B). It also initializes all N locations in the environment to entries with tag **variable** and value equaling the address of its own memory location. This gives this execution of the following clause its own set of variables, all initialized to ''unbound variable.'' Again, in the full WAM this initialization is avoided and shifted to specialized versions of other instructions.

Also, this instruction sets up the E register to point to the choice point where the new environment has just been created. Note that the current substitution associated with the i-th variable in the clause can be obtained by looking at memory location E+i.

The ***call*** instruction is used just after loading the argument registers with argument values for a goal literal in the body of the current clause. In operation, it saves the address of the next instruction in the CP register and branches off to the entry point of the clause code that corresponds to the predicate symbol in that goal. For all intents this is virtually identical to a subroutine call instruction found in many conventional computer instruction sets.

The ***return*** instruction is the last instruction in a clause segment, and if executed, it indicates successful satisfaction of all goals in the body of the clause. Control is passed back to the continuation address in the caller (as indicated by the BCP[E] field) with the caller's environment (reset E to BCE[E]) *without* deallocating any choice points or environments. This saves any assignments bound to the caller's variables, and permits proper PROLOG backtracking if necessary. Note also that the BCP and BCE in E were initialized to CP and E, respectively, by the mark instruction at the beginning of the procedure's code. CP, in particular, was set to the caller's return address by the call instruction that triggered the mark to begin with.

allocate N	For I = 1 to N do push ''var''#* to stack E←B	Allocate new environment to current choice point. (Initialize all variables.) Set E to this choice point.

call L	CP←PC + 1
	PC←L
		Branch to L, with return
		address in CP.

return	PC←BCP[E]
	E←BCE[E]
		Caller's return address.
		Caller's environment.

FIGURE 17-10
Simplified procedural instructions.

Note that, unlike a return in many conventional computers, this return does not pop anything off the stack. The choice points built by the procedure being returned from are left intact pending a possible backtrack that might restart one of them.

Figure 17-11 diagrams for a simple case how all the above instructions would have built a stack of choice points. The reader should follow carefully how this was built and where B and E point to just at, and just after, the indicated return instruction executes.

In the full WAM architecture, the functions of this instruction are split among several others.

17.5.3 Get Instructions

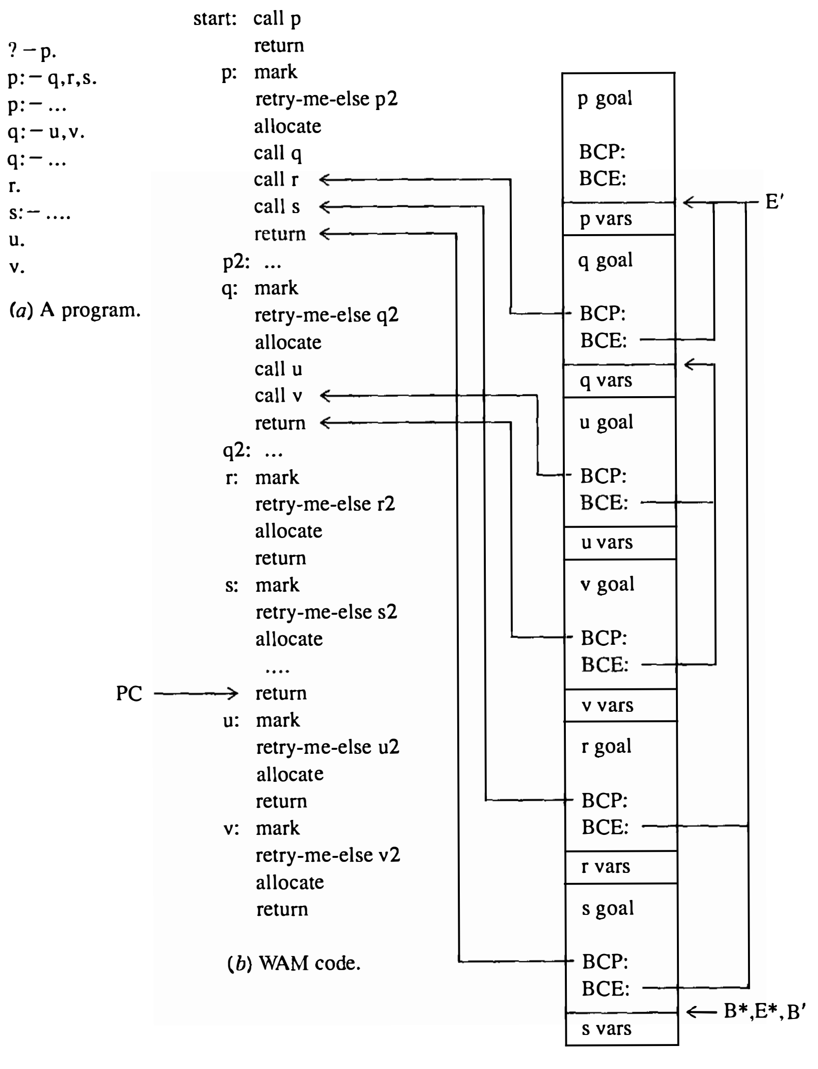
Get instructions (Figures 17-12 and 17-13) perform the initial unification checks between the actual arguments for the current goal (as found in the A registers) and the formal arguments in the head of a potentially applicable clause (as compiled into the code). They are used right after an ***allocate*** in the code section for that clause. At this point both B and E point to the same location in the same choice point. For consistency with later usage, we will use E to reference clause variables in the choice point.

There is typically one get per formal argument in the clause's head literal, with the code for the k-th formal argument referencing the actual argument in register Ak. The form of the get depends on the type of the formal argument. If the formal argument is defined by the programmer as some sort of object (i.e., a constant, list, or structure), then an instruction from Figure 17-12 is chosen. Such an instruction specifies the object's value and an argument register which should match.

If the formal argument is a clause variable, then a more generic instruction (***getv*** in Figure 17-13) is necessary. This instruction specifies both the clause variable (as an offset in the environment) and an argument register, and determines at runtime what value (if any) is bound to the clause variable and what specialized test should be used to compare it to the actual argument.

The following subsections describe each of these classes of get.

COMMON OPERATIONS. There are several operations common to both classes. First, because of prior unifications, it is possible for either an actual argument variable or a clause variable to have been bound to some other variable or object, meaning that the value of the first variable will track whatever is bound to the second. In such cases a **reference** tag is



```
?-p.
p:-q,r,s.
p:-...
q:-u,v.
q:-...
r.
s:-....
u.
v.
```

(*a*) A program.

```
start: call p
       return
   p:  mark
       retry-me-else p2
       allocate
       call q
       call r
       call s
       return
  p2:  ...
   q:  mark
       retry-me-else q2
       allocate
       call u
       call v
       return
  q2:  ...
   r:  mark
       retry-me-else r2
       allocate
       return
   s:  mark
       retry-me-else s2
       allocate
       ....
       return
   u:  mark
       retry-me-else u2
       allocate
       return
   v:  mark
       retry-me-else v2
       allocate
       return
```

(*b*) WAM code.

(*c*) Stack.

Note: B*,E* before return executes. B',E' after.

FIGURE 17-11
Sample stack of choice points.

found in the first variable's memory cell, with the value field of the cell indicating the address of the variable it is supposed to match. When trying to unify such an object to some other object, we must first ***dereference*** the object by chasing down the reference pointer(s) until something other than a **reference** tag is found. Figure 17-12 includes a description of this operation.

The second common operation occurs when a get instruction has found that some object is being matched to a currently unbound variable.

Instruction	Operation
get-constant C,Aj	Dereference Aj. If a variable: trail. Aj←"cons"#C If a constant and ≠C, fail. If anything else, fail.
get-list Aj	Dereference Aj. If a variable: trail. Aj←"list"#H. set write mode. Else if a list: SP←value[Aj]. set read mode. Else fail.
get-structure f/n,Aj	Dereference Aj. If a variable: trail. bind var to "strptr"#H. push "str"#f/n to heap. and set write mode. Else if a structure pointer then if functors match: set read mode, set SP to value of Aj + 1. Otherwise fail.

* To trail a variable: push its address to trail stack.
* dereference(X) = if tag(X)=reference
then dereference(memory(value(X)))
else X

FIGURE 17-12
Simplified get instructions.

In this case, unification always works, with a ***substitution*** generated which records the binding of the object's value to the variable. In the WAM this substitution is recorded by writing into the memory cell associated with the variable the tag and value of the object. This is fine if the rest of the head literal/goal unification goes through, but the machine must be able to "unwrite" the substitution if a later get finds a contradiction.

The process of recording the information necessary to repeal the substitution if necessary is termed ***trailing*** and consists of pushing onto the ***trail stack*** the address of the variable being bound. As mentioned before, we picked the format of a cell containing an unbound variable to consist of a tag **variable** and a value equaling the address of that memory cell. Consequently, to ***trail*** a variable we simply push a copy of its memory cell to the trail stack.

Failure of a get instruction to find a match between its argument and the corresponding A register causes a ***shallow backtrack*** through the current choice point (B) to try some other clause. The fail sequence from Figure 17-9 lists the operations required to do this. Unlike a deep back-

Instruction: getv Yj,Ai ; Yj an index into environment.

```
Operation: Clear PDL
  X←Ai
  Y←memory[E+Yj]
loop: use the tags of X and Y to select a case from:
```

			Y tag		
X tag	ref.	var.	con.	list	str.ptr.
reference	1	1	1	1	1
variable	2	3	5	5	5
constant	2	4	6	F	F
list	2	4	F	7	F
str.pointer	2	4	F	F	8

```
  Then, if PDL empty then instruction complete
       else X←mem(XA←PDL[top,1]).
            Y←mem(YA←PDL[top,2]).
            N←PDL[top,3] − 1.
            Pop PDL.
            If N>0 then push {XA + 1,YA + 1,N} to PDL.
            Repeat the loop.
Cases:
  1: X←dereference(X) and repeat loop.
  2. Y←dereference(Y) and repeat loop.
  3. Trail both X and Y,
     bind a reference to earlier one from the other.
  4. Trail Y, bind a copy of X to Y.
  5. Trail X, bind a copy of Y to X.
  6. If values not equal, fail.
  7. Push {value(X),value(Y),2} to PDL.
  8. If mem(X) ≠ mem(Y) then fail
     else push {value(X) + 1,value(Y) + 1,arity(mem(X))} to
     PDL.
  F. Fail.
```

FIGURE 17-13
A get instruction for formal variables.

track, this requires only reloading the A registers and resetting some stacks; the current choice point is left intact. Repealing the bindings listed on the trail stack then consists simply of popping objects off the trail stack and writing them into the address contained in their value field. This is often called ***unwinding*** the bindings.

Success in the match permits continued execution and possible assignment of values to variables in the clause's environment.

EXPLICIT GET INSTRUCTIONS. Figure 17-12 diagrams the get instructions used when the formal argument of a head literal is a well-defined object. For example, if the programmer wrote a clause of the form:

p(11,(Pete.Mike),**age**(Pete,40)):=...

the code representing this clause would use a ***get-constant*** instruction to check the first actual argument (in A1), a ***get-list*** instruction to check the second (in A2), and a ***get-structure*** instruction to check the third (in A3). In the latter two cases, the get instructions simply verify that the actual argument is an object of the appropriate type and leave checking of the components (such as Pete, Mike, and 40) to the ***unify*** instructions discussed later.

The first of these instructions, the ***get-constant,*** is simplest. The actual argument found in the specified A register is dereferenced. If the result has a tag of **constant,** then the value fields are compared. A match means that the unification is successful, and execution continues with the next instruction. A mismatch means that the actual and formal arguments are not unifiable, and thus this clause cannot be used for the current goal. The failure sequence described above is then invoked to start up the code for some other potential clause.

If the result of the dereference has a tag of **variable,** then that variable is trailed as described above, and a copy of the constant is stored into the variable's corresponding memory cell (wiping out the **variable** tag). This corresponds to a successful unification where a substitution is necessary.

If the tag of the dereferenced object is anything other than **constant** or **variable,** a mismatch is declared and the failure mechanism invoked.

Depending on the underlying machine and the range of data types supported, there may be several versions of get-constant, such as for integers, floating-point numbers, booleans,....

The next get instruction is the ***get-list.*** This instruction expects to see if the corresponding actual argument (after dereferencing) is either another list or an unbound variable. Any other actual argument causes a unification failure.

If the tag of the actual argument is a **list,** then at least it is the right kind of object. That is all this get instruction checks. A later pair of unify instructions will check that the car and cdr of the actual list match what is expected by the formal arguments. To set up for this test, the get-list will set the ***mode flag*** status bit in the CPU to ***read mode,*** and set the SP (structure pointer) register to point to the memory location containing the car of the actual goal's list.

If the tag of the actual argument is a **variable,** then we still have a successful unification, but we must now generate a substitution (in another environment) for that variable where the variable's new value is the formal list. In this case the get-list sets the tag of the variable cell to **list** and gives the cell's value field a copy of the current H register. This points the cell to a currently unused location on the heap where the following unify instructions will build a copy of the desired list. In addition, this instruction sets the ***mode flag*** to ***write mode,*** telling these unify instructions to build such a list.

The ***get-structure*** instruction is similar. After dereferencing the actual argument, this instruction expects to see either a variable or a structure pointer. Anything else causes a failure. In the latter case, the structure pointer is followed to what should be in memory a cell with a tag ***structure***, from which the value for the functor name and arity is extracted and compared to that stored in the instruction. A mismatch causes a failure.

A match means that the actual and formal arguments have at least the same function symbol and the same number of arguments. Again following unify instructions will check the components for matches. This is signaled by setting the ***mode flag*** to ***read mode*** and SP to point to one memory cell beyond the actual argument's **structure** cell.

An actual argument that is an unbound variable causes that variable to be trailed and the corresponding cell overwritten by a tag of ***structure pointer*** and a value equaling the current H register value. The cell at memory[H] receives a tag of ***structure*** and a value equaling the functor/arity code from the get-structure instruction. H is incremented to indicate that a cell has been assigned, and the ***mode flag*** is set to ***write mode.***

VARIABLE GET INSTRUCTIONS. The getv instruction handles the case where the formal argument is a clause variable (Figure 17-13). This is complex because in many cases the compiler cannot always know beforehand whether or not this clause variable might have a value at a particular point, or even what kind of value (in terms of data structure) that might be. The internal execution of the instruction must be able to handle all possible cases, including ones where multiple subcomponents of an object must be checked.

The individual cases in Figure 17-13 should be self-explanatory except perhaps for part of case 3, the binding of one variable to another. Conceptually either one could be bound to the other, while the approach given here always binds a reference to the one with the numerically smaller address to the one with the larger one. This corresponds to binding a reference to the variable that was built first to the one that was built later. It was chosen to avoid problems with later optimization techniques where the storage associated with intermediate variables is sometimes discarded.

As an example, consider the case where the clause head is **p**(x,x), generating code:

```
getv 1,A1; Assume 1 is offset in environment for x.
getv 1,A2; See if first argument matches the second.
```

For the case where the goal is **p**(2,2), the first getv binds 2 to x and the second one does a simple constant-to-constant test.

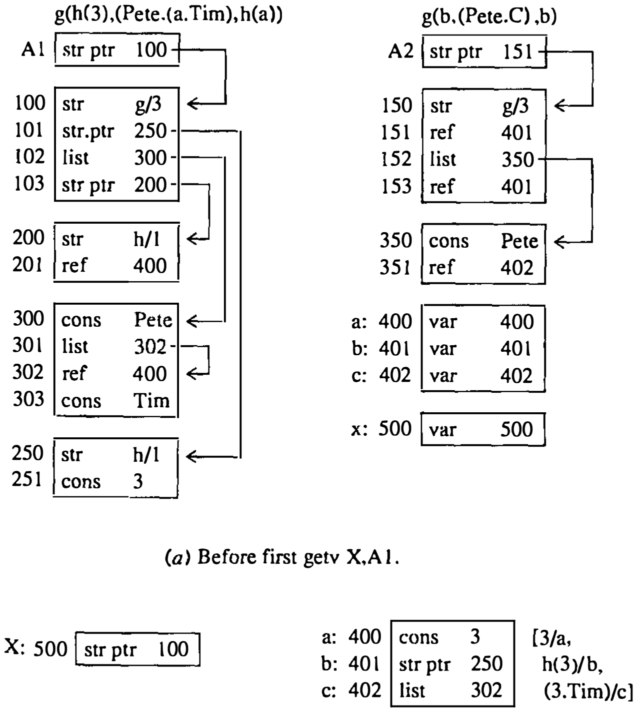
Now consider a goal of the form:

p(**g**(**h**(3),(Pete.(a.Tim)),**h**(a)), **g**(b,(Pete.c),b))).

In this goal **p** has two actual arguments, both complex structures. Of course, the formal code is the same as above. The first getv recognizes that x is unbound, and binds to it a structure representing **g**(**h**(3), (Pete.(a.Tim)),**h**(a)). The second getv must recognize that x is now bound to a structure which does have a matching functor and whose arguments can be unified by binding **h**(3) to b, 3 to a, and (3.Tim) to c. Further, several of the arguments are themselves complex objects requiring checks of their components.

Such a process is potentially recursive, requiring some sort of internal stack to keep track of complex objects that are not yet bound. In the WAM this is the purpose of the ***PDL*** (or ***push-down list***). This stack is emptied at the start of each getv, and as complex structures are found that must be matched, a triple consisting of the starting addresses of the actual and formal objects and the number of consecutive cells to compare is pushed onto the PDL.

Figure 17-14 diagrams storage before and after these two getvs. The reader should trace through the instructions' execution to verify the operation.



(a) Before first getv X,A1.

(b) After getv X,A1.

(c) After getv X,A2.

FIGURE 17-14
Sample sequence of getv instructions.

17.5.4 Put Instructions

Put instructions (Figure 17-15) are used to load the A registers with the actual arguments to be passed on to predicates found in the body of a clause. They occur in bunches, one bunch per literal on the right-hand side, with one put in each bunch for each top-level argument in the corresponding literal. For the most part their operation consists of simply copying something and involves no possibility of a fail or backtrack. They correspond to a "load register" in many conventional ISAs.

For complex argument objects such as lists or structures, these instructions handle only the start of the object, and start allocating space on the heap for the rest of the object. What goes into the argument register is a pointer to the first component on the heap. Later unify instructions then add to the object on the heap. The setting of the ***mode flag*** to ***write mode*** by the appropriate puts indicates to the unifys that objects are to be built.

17.5.5 Unify Instructions

A sequence of ***Unify instructions*** (Figure 17-16) are used after a get-xxx or put-xxx instruction to handle components of lists (xxx=**list**) or structures (xxx=**structure**). They operate in one of two modes signaled by the current value of the ***mode flag,*** either "read" or "write," which in turn was set by the prior get or put. In ***read mode*** they simply attempt to unify the next component of the object (as pointed to by the SP register) with the variable or constant specified in the instruction. A successful match may cause variables to be trailed and bound as in get, and increments SP to point to the next component. Also as before, a mismatch causes a ***fail sequence*** to back the processor up to try the next clause. Only get-xxx instructions can set the machine in a read mode.

Instruction	Operation
Put-constant C,Aj	Load Aj with tag = constant,value = C.
Putv Yi,Aj	Dereference Yi (at mem(E + Yi)) If a variable, Aj←"ref"#(E + Yi) Else Aj←mem[E + Yi].
Put-list Aj	Aj←"list"#H. Set write mode.
Put-structure f/n,Aj	Aj←"str-ptr"#H. Push "structure"#f/n to heap. Set write mode.

FIGURE 17-15
Simplified put instructions.

Instruction	Operation
unify-constant C	If write mode: push "cons"#C to heap. If read mode: dereference next cell from SP, and increment SP. If a variable: trail. Bind to "cons"#C. If a constant and equal: continue else fail.
unifyv Yi	Dereference Yi from current environment. If write mode: push copy of Yi to heap. If read mode: dereference next cell from SP, unify, increment SP.

FIGURE 17-16
Simplified unify instructions.

Also, in read mode the unify operation used by unifyv is essentially the same as that described for getv earlier.

In ***write mode*** these instructions copy the specified constant or variable to the object being built up on the heap. The initial get-xxx or put-xxx has earlier given either a register or a variable a reference to the start of this object. The SP register is not needed. Note that pushing something to the heap automatically increments the H register.

The astute reader will note that there are no unify-list or unify-structure instructions to handle the cases where a formal argument is either a list or a structure and where one or more components are themselves lists or structures. Such instructions could be added; however, they end up being more complex than the get-list and get-structure instructions described earlier (the SP register must now be stacked and unstacked from somewhere).

A simpler approach to handling such situations is as follows:

1. For each list or structure used as a component of a complex formal argument in the head of a clause, allocate an extra local clause variable cell not to be used anywhere else in the clause.
2. When the place in the code is reached where a unify-list or unify-structure would be used, replace it by a unifyv with an argument that specifies the new variable defined in step 1.
3. After completion of the top-level code for that formal argument, generate a putv to load some unneeded argument register with the contents of one of these new variables.
4. Follow this by a get-list or get-structure as appropriate against this argument register.
5. Use unify instructions as above to complete the components of this new structure.

Figure 17-17 includes an example of this process.

```
get-str    f/3,A1 ; f(
unify-con  70     ; 70,
unifyv     X      ; X,
unify-con  roy    ; roy)
```

(*a*) Unify A1 with f(70,X,roy).

```
get-list   A2     ; (
unify-con  Mike   ; Mike.
unifyv     Y      ; Y)
```

(*b*) Unify A2 with (Mike.Y).

```
get-list   A3     ; (
unify-con  1      ; 1.
unifyv     T1     ; (...))
putv       T1,A6
get-list   A6     ; (
unify-con  2      ; 2.
unifyv     T2     ; (...))
putv       T2,A6
get-list   A6     ; (
unify-con  3      ; 3.
unify-con  nil    ; nil)
```

(*c*) Unify A3 with (1 2 3).

```
get-str    g/3,A4 ; g(
unifyv     T1     ; (...),
unify-con  9      ; 9,
unify-con  mary   ; mary)
putv       T1,A6
get-list   A6     ; (
unifyv     T2     ; (...).
unifyv     T3     ; h(...))
putv       T2,A6
get-list   A6     ; (
unify-con  1      ; 1.
unify-con  2      ; 2)
putv       T3,A6  ; h(
get-str    h/1,A6
unifyv     X      ; X)
```

(*d*) Unify A4 with g(((1.2).h(X)),9,mary).

FIGURE 17-17
Some complex objects and their WAM code.

17.6 PROLOG-TO-WAM COMPILER OVERVIEW

This section summarizes briefly what an elementary compiler from PROLOG to our simplified WAM code might look like. Real compilers will be more complex because of some of the optimization techniques to be mentioned later, but the general structure is similar.

Figure 17-18 diagrams an example of the application of this compiler to a procedure from Figure 16-22.

There are two major sections to this compiler. First is an outer loop that cycles through the clauses and chains them into procedures. Second is the compilation of a single clause into a code section for the above procedure chains.

The approach taken here assumes a syntatically perfect PROLOG program; no error checking is included.

17.6.1 Procedure-Level Compilation

The following is a high-level description of the steps that might be involved in cycling through the clauses and linking sections of code compiled from them together. It assumes that we maintain a ***symbol table*** containing an entry for each symbol used as the predicate symbol of the head of some clause. At a minimum, an entry in this table has:

```
add(0,y,y).
add(s(0),y,s(y)).
add(s(x),y,s(z)) :- add(x,y,z).

; V1=y
add:       mark
           retry-me-else add2
           allocate 1
           get-constant 0,A1
           get V1,A2
           get V1,A3
           return
; V1=y
add2:      retry-me-else add3
           allocate 1
           get-structure "s",A1
           unify-constant 0
           get V1,A2
           get-structure "s",A3
           unify V1
           return
; V1=x,    V2=y, V3=z.
add3:      retry-me-else add4
           allocate 3
           get-structure "s",A1
           unify V1
           get V2,A2
           get-structure "s",A3
           unify V3
           put V1,A1
           put V2,A2
           put V3,A3
           call add
           return
add4:      backtrack
```

FIGURE 17-18
Compilation of the symbolic **add** procedure.

- The name of a symbol
- The memory address of the initial mark instruction for the symbol
- The memory address of the last retry-me-else instruction compiled for that symbol
- Space for a list of places in the program where this predicate symbol has been referenced as a right-hand-side goal literal (i.e., where it shows up as the argument to a call)

- A flag indicating whether or not any code has been generated yet for the predicate symbol

We assume below that this table is built first, with all entries but the first initialized to appropriate nulls.

The program clauses are then processed in the order they were entered by the programmer, using the following algorithm:

1. Select the next unprocessed clause from the program and get the predicate symbol used in the head literal.
2. If the symbol table entry for this symbol indicates that no code has been generated for it yet:
   - a. Mark the entry as having had code generated;
   - b. Record as the initial address the next available memory location;
   - c. Compile into this location a mark instruction, followed by a retry-me-else instruction (with the label field left empty);
   - d. Save in the symbol table entry the address of the retry's label.
3. If the symbol table entry indicated that some code has already been generated:
   - a. Store into the memory word designated by the last retry field the address of the next available word in memory;
   - b. Compile into this location a retry-me-else instruction (with the label field left empty);
   - c. Save in the symbol table entry the address of the retry's label.
4. Generate a code section for the clause as described in the next section.
5. If there are more clauses in the program, go back to step 1.
6. After compiling all clauses:
   - a. Compile into the next available location a backtrack instruction, remembering the address where it went;
   - b. For each symbol table entry, fix up the label field of the last retry instruction to point to this backtrack;
   - c. For each symbol table entry, go through the list of addresses of call instructions that used that symbol, and write into those locations the address of the mark instruction.
7. Compile the query using a variant of the clause code generator (no head unification code is needed).
8. The first instruction of the query's code is the program's starting point.

17.6.2 Clause-Level Compilation

This section assumes that a clause has been selected for compilation by the outer compiler routine, and that all the appropriate interclause link addresses are set up properly in the symbol table.

1. Calculate the number of local variables needed, including allowances for temporaries used for complex objects buried inside other complex objects. If the number is not zero, generate an allocate instruction. Also build a list pairing local variable names and their offsets.
2. Process the k-th formal argument of the head as follows (k=1,2,...):
   - a. If it is a constant, generate a get-constant instruction, encoding in the value of the constant and Ak.
   - b. If it is a variable, generate a getv instruction, encoding in the offset to that variable from the above-computed pairings and Ak.
   - c. If it is a list, generate a get-list instruction which references Ak. Then for the car and cdr of this list generate either a unify-constant or unifyv instruction, as appropriate. If either car or cdr is a complex object, generate a unifyv instruction which refers to one of the allocated temporaries (see the discussion on unify instructions).
   - d. If it is a structure, generate a get-structure instruction, encoding in the name of the functor and its arity. Then do exactly as described for lists for each argument of this expression.
3. For each complex object that was a component of some other object:
   - a. Generate a putv instruction which refers to the allocated local variable and to some argument register that is not needed any more.
   - b. Generate either a get-list or a get-structure instruction as appropriate against this register.
   - c. For each component of this object, generate code as was done above.
4. Process the goal literals on the right-hand side one at a time, from left to right.
   - a. Process the i-th top-level argument of the next literal as follows (i=1,2,...):
     - (1) If it is a constant, generate a put-constant instruction, encoding in the constant's value and Ak.
     - (2) If it is a variable, generate a putv instruction, encoding in Ak and the offset to the variable from the pairings developed in the first step.
     - (3) If it is a list or a structure all of whose components are either variables or constants, generate either a put-list or a put-structure as appropriate (with Ak encoded), and then generate a unify-constant or unifyv as required for each argument.
     - (4) If it is a list or structure that includes an embedded list or structure:
       - (a) Take the most deeply nested such component.
       - (b) Select a currently unused argument register Au.
       - (c) Generate an instruction sequence as in the prior step (put-list or put-structure), but target the result to Au.
       - (d) Generate a getv to place Au in a specially allocated clause variable (as was done for nested objects in the clause head).
       - (e) Repeat the above process for the next most nested compo-

nent, except that for components that refer to nested structures that have already been processed, use a unifyv instruction with the offset of the clause variable into which they were compiled earlier.

(f) Mark the register Au as free again.

b. Generate a call instruction, leaving the label field empty.

c. Link the address of this field into the symbol table entry for the predicate being called.

5. Complete the code by generating a return instruction.

Note that the processing order for complex objects as they are built for right-hand-side goals is just the opposite of that for head unification, namely, inside out versus outside in. They do, however, use the same idea of temporarily saving a partially processed object in an extra clause variable until it is needed.

Note also that one of the final steps in the outer loop (as described in Section 17.6.1) uses information from the symbol table to fix up all the labels for the call instructions to point to the correct procedure entry points.

17.7 SUPPORTING BUILT-INS

The WAM as described so far supports implementations of "pure" PROLOG, that is, PROLOG without special built-in predicates that have "side effects" such as **cut,** I/O, and the various predicates to read and modify the program dynamically. Such predicates fall into three categories: those requiring some new, but not difficult, WAM instructions; those that can be built up out of current WAM instructions; and those presenting really difficult challenges. The following subsections give some examples of each.

17.7.1 Some Simple New Instructions

Given the resources described so far, several of the standard PROLOG built-in predicates can be easily implemented as simple new instructions, or use some simple instructions in short sequences to support them. Figure 17-19 gives several of these instructions.

Fail, for example, simply invokes the failure sequence. This is is exactly what is needed for the PROLOG predicate **fail,** meaning that whenever a **fail** occurs on the right-hand side of a clause, the code to be compiled consists of a single WAM fail.

Escape is an instruction which permits a WAM machine to communicate with some other processor, perhaps one that is capable of arithmetic functions, I/O, or some user program written in some other language such as Pascal or C. The instruction simply takes the current argument registers, places them somewhere the other processor can access them, signals the other processor, and waits for a completion in return. In many implementations this instruction might also want to reload the registers from the communication area to recover answers from the coprocessor.

Instruction	Operation
fail	Initiate the failure sequence.
escape	Pass registers to attached coprocessors, wait for coprocessor complete, then restart.
swx-on-type	Ax,Lvar,Lcon,Llist,Lstr Dereference Ax and branch one of 4 ways depending on tag.
cut	B←BB[E].

FIGURE 17-19
New WAM instructions to support built-ins.

The ***switch-on-type*** instruction specifies an argument register and four addresses to branch to. The argument register's contents are dereferenced, and the tag of the result is tested. Depending on the tag, one of the four branch addresses is placed in the PC. Very often only the last three addresses are specified, with the first case causing execution to continue with the next instruction after the switch.

This instruction is handy for testing the type of an argument or variable.

The cut instruction is designed to completely implement the ***cut*** predicate in PROLOG. To understand it, consider a clause of the form:

p(...):=q1(...),...,qn(...),!,r1(...),...,rm(...).

When converted into WAM code, the code to support the cut operation should come immediately after the call **qn** instruction. Following this code should then be the normal puts in support of **r1.**

If program execution ever reaches the cut code, B points to the most recent choice point built in support of **qn,** while E points to the earlier one established for **p.** In between these two choice points are at least n−1 other choice points (at least one for each **qi**). If ! were a normal predicate, a failure in it (or one beyond it that ripples back to it) would cause the topmost choice point (for **qn** or a goal subsidiary to that) to be restarted.

The semantics of a cut, however, dictate that on its first execution it will always succeed, but will have the effect that any backtracks through it should result in a backtrack through the **p** choice point. It should be as if the choice points for **q1** through **qn** never existed, and that this clause is the last one possible for the **p** predicate.

One way to do this is to have the cut code set B to point to the same choice point as E does, and to load FA[E] with the address of some location known to hold a backtrack instruction. Now if the code for **r1** backtracks, it will restart **p**'s choice point, which will branch to a backtrack instruction, which in turn will restart the prior choice point as desired.

Another way to achieve the same effect is simply to reload B with BB[E]. This is the choice point directly in back of that for **p**. Now a failure in **r1** will cause a backtrack directly to the desired choice point. This is the method shown in Figure 17-19.

Once a cut is executed, for all practical purposes the storage for the intermediate choice points will never be accessed again, and will be recovered if a backtrack from **r1** occurs. If **r1** never backtracks, however, this storage is simply unrecoverable dead storage. Consequently, in a memory-efficient implementation one might want to consider recovering it (and associated heap storage) as soon as the cut code is executed. Most modern PROLOG systems try to do at least some of this, but it is tricky, and a discussion of the techniques is left for later (see Section 18.4).

17.7.2 Multiinstruction Built-ins

Many of the common PROLOG built-in predicates can be implemented directly as sequences of WAM instructions. Figure 17-20 gives some samples.

The code for the ***disjunction*** ";" is worth discussion. What it does is build a separate choice point for the goals involved that will try the second if the first does not succeed, and so on. Note that the code generated for these literals is the same as if there had been no extra choice point; the put instructions still work because the mark did not change the E register, which points to the proper environment. This also implies that any local variables needed by these goals were counted in when the clause containing the disjunction was processed.

Other classes of built-in predicates are also implementable by appropriate sequences. The comparison predicates (">," "<,"...), for example, could consist of a pair of swx-on-type instructions that continue execution only if both arguments dereference to constants, and fail otherwise. These could be followed by an escape instruction to go off to a coprocessor to do the comparison (if the WAM itself does not support arithmetic comparison instructions).

The ***is*** built-in can be handled similarly. A tree of swx-on-type instructions can test the arguments, determine whether two or three of the arguments are constants (and which ones), and call the appropriate escape sequence.

Finally, given a method for implementing ";", several of the other PROLOG built-ins can be implemented by recasting them as disjunctions and compiling the results as above. Figure 17-21 gives some examples.

```
        Prolog Builtin   Code Sequence
------------------------------------------------------------------
true                       ; requires no WAM code at all

; var(x)
                             ... get x into A1
                             swx-on-type A1,Ok,Notok,Notok,Notok
                        Ok:  ... continue
; x=y        note x is in    A1, y in A2
                             getv T1,A1 ; T1 a new clause variable
                             getv T1,A2
; (L1;L2; ...;Ln) Lk all goal literals
                             mark
                             retry-me-else tryL2
                             ... puts for L1
                             call L1
                             jump Ok
                   tryL2:    retry-me-else tryL3
                             ... puts for L2
                             call L2
                             jump OK
                             ...
                             call Ln
                       Ok:   ... continue
```

*At location Notok there is a fail instruction.

FIGURE 17-20
Some multi-WAM instruction built-ins.

Prolog Predicate	Equivalent Disjunction Form
not(L)	((L,!,fail);true)
(L1→L2;L3)	((L1,!,L2);L3)

FIGURE 17-21
Compiling built-ins using disjunctions.

17.7.3 Tough Predicates

Some of the PROLOG built-in predicates represent a very tough challenge to implementation with WAM code. In particular, these include the predicates that read and modify the set of PROLOG statements during execution, such as ***clause, assert,*** and ***retract.*** The reason is, of course, that by the time the PROLOG compiler has completed its processing, the original source code has been lost, and it is often difficult to deduce from the compiled code what clause or statement is represented. This is particularly true of optimizing PROLOG compilers that go even further than we have discussed here.

Various schemes have been discussed in the literature (cf. Dobry, 1987a), such as leaving to an interpreter all clauses whose head predicate symbol might be the target of such built-ins, limiting such predicates to programmer-defined ones where all such clauses are in the form of facts, or keeping both source and compiled code and doing on-the-fly recompilation as required. None of these is very effective.

17.8 DETAILED EXAMPLE

As an example, Figure 17-22 repeats the two-clause definition of ***append***, and what it would translate into in our simplified WAM code. In the literature this PROLOG program is often used as a benchmark for performance studies, but sometimes with the predicate name ***concatenate*** instead of ***append***.

The WAM code for this program follows exactly the guidelines discussed above. Any time the **append** predicate symbol shows up in a goal, control is transferred to the first instruction, the mark, with three arguments in argument registers A1, A2, and A3. This instruction builds the initial choice point and drops down to the entry **append1** for the first clause. The entry code here consists of a retry-me-else, which designates **append2** as the starting location for the next possible rule for **append** and allocates space for the single clause variable *x*. The body of the clause code consists of checking that the first argument is nil, and then that the second and third arguments are the same. The final instruction, return, returns control to the caller if this all worked.

```
append(nil,x,x).
append((H.L1),L2,(H.L3)):-append(L1,L2,L3).

;At entry registers A1, A2, A3 hold the three arguments.
append: mark                  ;build the choice point
append1:retry-me-else append2;try the first clause first.
; code for the first clause
        allocate 1            ;only one clause variable x
        get-constant nil,A1   ;first arg must be nil
        getv y1,A2            ;bind 2nd arg to x=y1
        getv y1,A3            ;see if it matches 3rd
        return                ;if so, we're done
append2:retry-me-else quit    ;try the second clause.
; code for the second clause
        allocate 4            ;4 clause variables H,L1,L2,L3
        get-list A1           ;first argument must be list
        unifyv y1             ; with car put into H
        unifyv y2             ; and cdr in L1
        getv y3,A2            ;save second argument
        get-list A3           ;3rd arg must be list
        unifyv y1             ; with car H
        unifyv y4             ; and cdr L3
; successful unification—build arguments for RHS
        putv y2,A1            ;build 1st arg=L1
        putv y3,A2            ;build 2nd arg=L2
        putv y4,A3            ;build 3rd arg=L3
        call append           ;call the new goal
quit:   return                ;if it works, we're done
; come here if none of above clauses work.
quit:   backtrack             ;Backtrack to caller.
```

FIGURE 17-22
Simplified WAM code for **append**.

The second clause is a little more complex. Here the first argument is supposed to be a list, so the unification code for the first argument starts with a get-list. If, after dereferencing, A1 is a list, this will set the SP register to point to its car cell, and set the machine to read mode. The following two unifyv instructions then try to unify the car of the actual list with *H* (as stored in *y1*), and the cdr with *L*1 (as stored in *y2*). Both of these have no current value, so the net effect is a copying of the car and cdr of the actual list into these two variable locations in the environment.

If the actual argument in A1 is an unbound variable to begin with (tag=var, value=own address), a copy of the variable's cell is pushed to the trail, the cell itself is loaded with a tag of list and a value pointing to the top of the heap, and the machine is set to write mode. We will build for the variable a new list on the heap. The two unifyv instructions handle the specifications for the car and cdr of this list, and will create the appropriate entries on the heap.

Another getv and then get-list, unifyv, unifyv combination follows to process the other two arguments.

Following this code, three putvs create the new argument values needed to call append from the body. Note that copies of the original A registers are in the choice point where they get reloaded if a backtrack occurs.

The recursive call to **append** repeats this whole process over again with the new arguments.

Upon a successful return from one call, the return instruction will successfully return from either code sequence.

The final instruction is a backtrack, and is positioned as a new clause if the second one fails. This signals that there are no more rules for this goal, and deep backtracking must occur.

17.8.1 Determinate Execution Trace

As an example of the execution of this program, consider what would happen if it were called with a goal of the form **append**((1 2),(3 4 5),*x*).

The result of the execution should bind *x* to (1 2 3 4 5). In the PROLOG benchmarking literature this is called a ***determinate concatenate*** because there is exactly one answer, and asking for more returns nothing.

Figure 17-23 diagrams storage as it might exist just after **append** has been called, and the instructions mark, retry-me-else, and allocate executed. We assume here that the heap locations 100 through 111 were loaded by earlier parts of the program as the arguments were constructed. The reader should trace from the A registers through these locations and verify that the desired arguments have been properly represented.

The locations 200–208 represent the choice point built by mark and modified in the FA component by retry-me-else to point to the start of the second rule.

The allocate has initialized one location after this choice point to hold the value for clause variable *yl* (corresponding to *x* in the text).

Now consider what happens when the get-constant is executed. A1 has a **list** tag which cannot possibly match the constant nil, so a fail sequence is initiated. The heap is reset to 111 (nothing was added, so this has no effect), the stack is reset to 209, no unwinding is necessary (nothing was added to the trail), and control is transferred to FA[B], or location **append2.** Here FA[B] is reset to the location labeled **quit,** and space for a four-clause variable is allocated and initialized to "var"#×. Note that this allocation occurs right over the previous allocation, since its contents are no longer needed.

Now the get-list instruction executes. A1 is a list, so SP is set to 100 (the address of the car of the list), the machine set to read mode, and the first unifyv is executed. This instruction finds *yl* an unbound variable ("var"#209), so it is trailed ("var"#209 is pushed to trail), location 209 is set to a copy of location 100 (namely, "cons"#1), and SP is incremented to the list cdr. The same process repeats in the next instruction for the cdr (2).

The getv *y3*,A2 finds an object in A2 and *y3* unbound, so *y3* is trailed ("var"#211 is pushed to trail), and location 211 is set to a copy of A2.

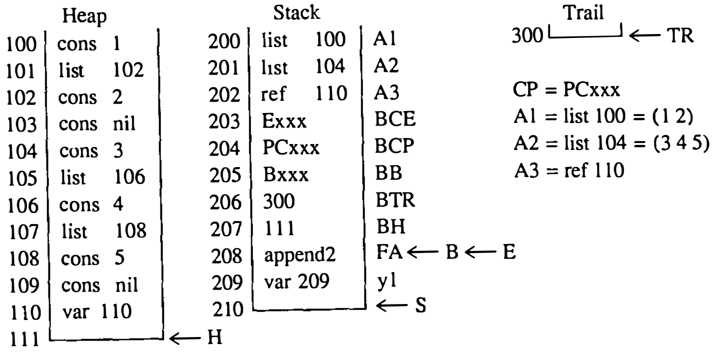


FIGURE 17-23
Memory just before get-constant nil, A1.

The next three instructions are similar to the first, but execute in write mode because here the argument A3 is an unbound variable at location 110. It is trailed ("var"#110 is pushed to trail) and reset to point to a list to be constructed on the heap (at locations 111 and 112). The two unifyvs construct these components.

Completing these instructions completes the successful unification of the original goal with the head of the second clause. Following this are three putv instructions to load registers A1, A2, and A3 with copies of the appropriate clause variables.

At the call instruction, the CP register is set to the next instruction (location appx), and control is transferred back to **append.** Figure 17-24 diagrams storage just after this. Any memory not shown here is the same as in the previous diagram.

Now the whole process described above is repeated exactly, with a new choice point built, the first clause tried and rejected, and the second clause entered. Again, executing code in this clause will cause various locations to be trailed, and more list structures to be built on the heap. Note also that in this process the variable *y4* from the first choice point ends up getting trailed and bound to a new list on the heap with car=2 and cdr the most recent *y4*.

Figure 17-25 diagrams storage just before the second call. Note in particular the change to location 212 as a result of the second get-list. Again the reader should verify this.

The second call again creates a new choice point and enters the first clause. Here, however, A1 contains the constant nil, so a fail sequence is not initiated. Instead, the second two getvs are executed, with the net result of assigning the variable at location 225 to the list at 104 [namely, (3 4 5)]. The return is then executed, which returns control to the return in the second clause (via the second choice point), which in turn branches to itself (via the first choice point), and then back to the original caller. Note that none of the additions to the heap, stack, or trail has been changed.

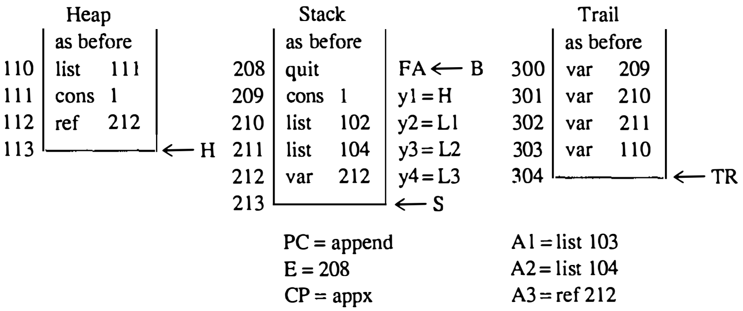


FIGURE 17-24
Memory just after **call append.**

	Heap			Stack			Trail	
111	cons 1		212	list 113	y4=L3	304	var 222	
112	ref 212		213	list 102	A1	305	var 223	
113	cons 2		214	list 104	A2	306	var 224	
114	ref 225		215	ref 212	A3	307	var 212	
115		←H	216	208	BCE	308		←TR
			217	appx	BCP			
			218	208	BB			
			219	304	BTR			
			220	113	BH			
			221	quit	FA←B←E			
			222	cons 2	y1=H		CP = appx	
			223	cons nil	y2=L1		A1 = cons nil	
			224	list 104	y3=L2		A2 = list 104	
			225	var 225	y4=L3		A3 = ref 225	
					←S			

FIGURE 17-25
Memory just before second **call append**.

Figure 17-26 diagrams storage at this point.

To verify that the proper value has been returned, look into location 110. This has a value "list"#111, which means that it is a list object whose car is the object at 111 (a constant 1) and whose cdr is at 112 (a reference to 212), which in turn is the list at 113. The car of this list is the object 2, and the cdr at 114 (a reference to 225) is itself a list at 104 [which was the original (3 4 5) list]. The net result is the list (1 2 3 4 5), as expected.

To demonstrate backtracking, consider what would have happened if the initial call had been **append**((1 2.7),(3 4 5),*x*). Initially, the only difference would have been that memory location 109 would have contained "constant"#7 rather than nil. Execution would have proceeded exactly as through Figure 17-25, and into Figure 17-26, except that the get-constant would have failed, as would have the get-list when the second clause was tried. This would have caused the backtrack instruction at quit to be executed. The heap, stack, and trail would have been reset to 115, 226, and 308; the A registers would have been reloaded to the values in 226–228, the choice point would have been backed up to location 221, location 225 would have been reset to "var"#225, i.e., a variable with no value, and control would have been transferred back to the contents indicated by location 221—quit. The memory at this point is exactly the same as Figure 17-25.

	Heap			Stack			Trail	
	as before			as before			as before	
115		←H	225	list 104	y4=L3	308	var 235	
			226	cons nil	A1	309		←TR
			227	list 104	A2			
			228	ref 225	A3			
			229	221	BCE			
			230	appx	BCP		B=234	
			231	221	BB		E=Exxx	
			232	308	BTR		PC=PCxxx	
			233	115	BH		A1=cons nil	
			234	append2	FA		A2=list 104	
			235	list 104	y1=x		A3=ref 225	
					←S			

FIGURE 17-26
Memory after final return.

Executing the code at quit causes another backtrack to the storage indicated by Figure 17-24, with control passed to that indicated in location 208, namely, quit again. This third backtrack will reset storage to exactly what it was at the beginning.

17.9 PROBLEMS

1. Convert the set of PROLOG statements for each of the following predicate symbols as found in the text into a sequence of simplified WAM instructions:
   a. **member**(*x*,*y*) [Use the variable *z* in place of the anonymous variable]
   b. **reverse**(*x*,*y*)
2. Develop a complete WAM program for the PROLOG program of Figure 16-22. Note that Figure 17-18 gives the section of code for **add.**
3. Add the query ?−**factorial**(s(s(s(0))),*y*) to the program for Problem 2.
4. Consider the literal **p**((3.(*x*.(s(0))))). Generate WAM code when:
   a. This is the head of a clause
   b. This is a goal on the right-hand side of a clause. Assume that *x* is mapped to local variable Y2.
5. Generate simplified WAM code for the following set of clauses (assume that you have a "less A,B,L" instruction, which compares registers A and B and branches to L if A, *but which requires that both A and B have constants in them*):

   ssplit((*x*.*L*),*y*,*L1*,*L2*):−**nsplit**(*L*,*y*,*L1*,*L2*,*x*).
   ssplit(nil,__,nil,nil).
   nsplit(*L*,*x*,*L1*,*L2*,*x*):− !, **ssplit**(*L*,*x*,*L1*,*L2*).
   nsplit(*L*,*y*,(*x*.*L1*),*L2*,*x*):−*x*<*y*,!, **ssplit**(*L*,*y*,*L1*,*L2*).
   nsplit(*L*,*y*,*L1*,(*x*.*L2*),*x*):−*y*<*x*,!, **ssplit**(*L*,*y*,*L1*,*L2*).
6. Write in PROLOG a WAM compiler for the simple propositional logic language Proplog from Figure 16-4. Generate WAM code for it.
7. Compare the WAM to the SECD. Where are the major differences; the major similarities? Develop an abstract architecture that would encompass both.

CHAPTER 18

OPTIMIZATIONS AND EXTENSIONS

Chapter 17 introduced the Warren Abstract Machine and how PROLOG programs compile into it. Despite its seeming sophistication, however, the WAM described there is not quite the one used in many modern PROLOG (and other logic language) systems. A variety of optimizations are often used.

This chapter describes the more common of these architectural extensions, and indicates when they are useful and how a PROLOG compiler might locate them. Basically the techniques fall into three categories:

- Optimizing code for individual clauses
- Optimizing code for multiple clauses
- Other techniques

The order of the presentation here was chosen to simplify the overall flow and does not necessarily match the expected performance gains.

The first section summarizes a revised WAM that reflects all the optimizations from this chapter that seem to be most used today. A listing of our standard procedure **append** in this architecture is also included.

These WAM/compiler optimizations, however, are not the only ones possible for logic languages such as PROLOG. Many language extensions permit a programmer to provide the compiler with information (called ***annotations***) about the direction of substitutions for identifiers inclause heads. Such information permits a compiler to specialize the code generated for goal-head unifications regardless of the WAM architecture employed. This can greatly reduce execution time for many real programs, where the nature of arguments to goals is known in advance. Related techniques can also tremendously reduce the expense of backtracking by predicting which prior goal in a clause should be restarted. Section 18.5 describes such optimizations.

Although most of the topics here are of an advanced nature, and are not strictly necessary for at least a top-level understanding of later chapters, there are at least two the reader should pay attention to. First, the indexing techniques of Section 18.3.1 are almost universally used in real systems, and have a very significant effect on their performance. In contrast, the annotations and backtracking controls of Section 18.5 represent one of the most promising, and intensely studied, directions of future research.

18.1 AN OPTIMIZED WAM SUMMARY

(Touati and Despain, 1987; Dobry, 1987a; Dobry et al., 1985; Turk, 1986)

Figure 18-1 summarizes one version of the WAM architecture after most of the modifications from the following sections have been factored in. Figure 18-2 gives the results of using that ISA for the **append** predicate of Chapter 17. Notice that the entry point for this code, at "append," is in the middle of the code sequence.

Although all the techniques described here appear to be useful, quantifying them is often difficult. Variations in benchmarks, underlying machines, etc., all complicate the analyses. For specific examples, the reader is referred to the references.

18.2 OPTIMIZING SINGLE CLAUSES

This section describes techniques that can be used to optimize the code generated for a single clause. This includes:

- Minimizing the need to trail variables
- Eliminating unnecessary variable cell initializations
- Reducing local environments by keeping variables in registers
- Minimizing the need for choice points by breaking the environment away from the choice point
- Eliminating unnecessary branches from call/return sequences.

18.2.1 Trail Optimization

The simplified WAM architecture ***trail***ed a variable by pushing its address on the ***trail stack*** any time the variable was given a value. When a unification failure occurred, such cells were unwound back to their original unbound state. This is a simple, but time-consuming, process.

---------- Indexing Instructions ----------	
switch-on-type Ax,Lv,Lc,Ll,Ls	—Dereference Ax. Branch on tag.
switch-on-constant Ax	—Compute hash from Ax. Branch through table.
switch-on-structure Ax	—Compute functor hash. Branch through table.
try-me-else L	—Build Choice Point,B←S − 1,HB←H,FA[B]←L.
try L	—Build Choice Point,B←S − 1,HB←H,FA←PC + 1,PC←L.
retry-me-else L	—FA[B]←L.
retry L	—FA[B]←PC + 1, PC←L.
trust-me	—B←BB[B], HB←BH[B].
trust L	—B←BB[B], HB←BH[B], PC←L.
---------- Procedural Instructions ----------	
allocate	—Temp←E, E←max(E + 2 + N,B), CE[E]←Temp. CCP[E]←CP.
deallocate	—CP←CCP[E], E←CE[E].
call L,Num	—CP←PC + 1,PC←L,N←Num.
proceed	—PC←CP.
execute L	—PC←L.
---------- Put Instructions ----------	
putTval Ay,Ax	—Ax←Ay.
putPval Y,Ax	—Ax←mem[E + 2 + Y]
putTvar Ay,Ax	—Ax←Ay←"ref"#H, Push "var"#* to H.
putPvar Y,Ax	—mem[E + 2 + Y]←"var"#*, Ax←"ref"#(E + 2 + Y).
put-cons C,Ax	—Ax←"cons"#C.
put-list Ax	—Ax←"list"#H. Set write mode.
put-str f/n,Ax	—Ax←"str-ptr"#H. Push "str"#f/n to H. Set write mode.
put-unsafe-val Y,Ax	—Dereference Y. If variable in current E then push "var"#H to heap. Ax←"ref"#H. else Ax←dereferenced value.
---------- Get Instructions ----------	
getTval Ay,Ax	—As getv in simple WAM.
getPval Y,Ax	—As getv in simple WAM.
getTvar Ay,Ax	—Ay←Ax.
getPvar Y,Ax	—mem[E + 2 + Y]←Ax.
get-cons C,Ax	—As in simple WAM.
get-list Ax	—As in simple WAM.
get-str f/n,Ax	—As in simple WAM.
---------- Unify Instructions ----------	
unifyTval Ax	—As unifyv in simple WAM.
unifyPval Y	—As unifyv in simple WAM.
unifyTvar Ax	—In read mode: Ax←mem[SP], SP←SP + 1. In write mode: Ax←"ref"#H, Push "var"#H to heap.
unifyPvar Y	—In read mode: mem[E + 2 + Y]←mem[SP], SP←SP + 1. In write mode: mem[E + 2 + Y]←"var"#*
unify-void	—In read mode: SP←SP + 1. In write mode: Push "var"#H to heap.
unify-cons C	—As in simple WAM.

Note: "x#y" stands for a cell with tag field x, value field y.

FIGURE 18-1
Modified WAM ISA.

```
append(nil,X,X).
append((H.L1),L2,(H.L3)): − append(L1,L2,L3).

try-all:   try app1
           trust-me
app2:      get-list A1        ;Second append clause
           unifyTvar A4       ;A4 = H
           unifyTvar A1       ;A1 = L1
           get-list A3
           unifyTval A4
           unifyTvar A3       ;A3 = L3, drop through for tail recursion
;
; Come to following instruction at beginning of append
append:  switch-on-type A1,try-all,app1,app2,fail
app1:    get-nil A1         ;First append clause
         getTval A3,A2
         proceed
```

FIGURE 18-2
Optimized **append** program.

To understand how much of this process could be avoided, consider what happens in the failure sequence. Both the heap and the stack are reset to what they had been when the most recent choice point (located by B) was built, and all information added since that point is of no further use. This means that any variables in either the environment that was at the top of the stack or in the part of the heap added by the failed clause need not be reset (see Figure 18-3). Thus, they need not have been trailed to begin with.

In the full WAM architecture, the arrangement of storage was deliberately chosen to simplify such a test. Basically, a variable being given a value need never be trailed if either:

1. It is in an environment beyond the current choice point (if its address is numerically greater than B); or
2. It is in part of the heap created since the most recent choice point was built (if its address is between BH[B] and H).

In summary, if some logical variable (allocated to memory location x) is to be bound to a value, its address x need not be trailed if either $B<x$ or $BH[B]x<H$.

These tests are simple to perform at the hardware level, but they require a memory read of BH[B] each time. To remove this memory read, the full WAM defines an additional hardware register, the ***HB register*** (for "heap at last B"), and modifies some of our simple WAM instructions to guarantee that its contents always correspond to BH[B]:

- During a mark, set HB to H.

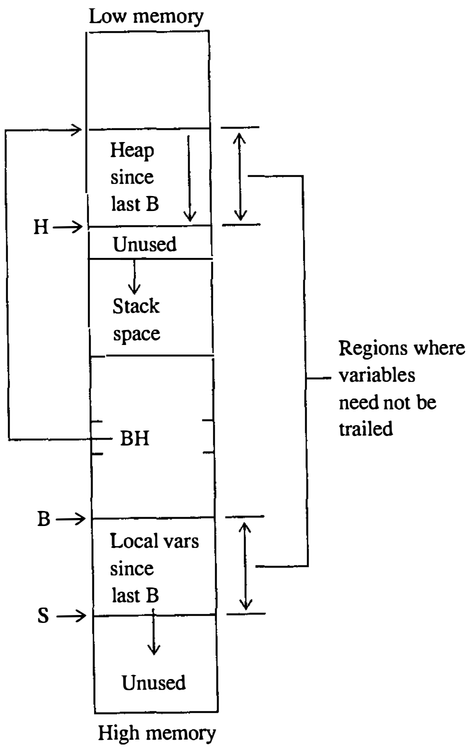


FIGURE 18-3
Variables that do not need to be trailed.

- During a backtrack, reset HB to BH[B], where B is the choice point being returned to.
- During a fail, replace the "H←BH[B]" memory read by the simple H←HB register transfer.

With these modifications, the WAM machine function to test a variable address to see if it need not be trailed becomes $B<x$ or $HB<x<H$.

In high-performance implementations in which hardware support for the WAM can be added, such a test can be done quite rapidly by some additional hardware comparators in parallel with other activities.

18.2.2 Environment Variable Initialization

As defined in Chapter 17, the allocate instruction must initialize each entry in the environment it creates to represent an unbound variable. If this were not done, later get instructions might mistake whatever happened to be in those locations as "values," and the program would run randomly amok. Consider, however, what happens if we let the compiler identify the point in the code for a clause where each variable is used for the first time (either in a get, a put, or a unify), and invent some instructions that bury the initialization there. The initialization can then be deleted from the allocate and spread around to just those spots where it is needed. Further, in many cases we can simplify the work done by these new instructions if we take into account our knowledge that they will be used only for uninitialized variables. This can actually save a measurable amount of computation.

Warren's approach to this replaces the getv, putv, and unifyv instructions described above by pairs of the form ***get-var*** and ***get-val, put-var*** and ***put-val,*** and ***unify-var*** and ***unify-val.*** The suffix -var is short for variable; likewise, -val is short for value.

The xxx-val instructions function almost exactly like the prior xxxv ones. They assume that the variables have been initialized, and thus must be read and tested. However, the xxx-var instructions assume that when they are executed, this is the first time in the current clause that the indicated variable has been referenced, and thus it should be considered to have a **variable** tag regardless of the current contents.

In particular, the get-var instructions now need perform no unification checks, but need only copy the specified argument register into the designated environment cell. This saves both a read and a tag check over the original instructions and avoids altogether the initializations that were done by the original definition of allocate. The instruction is now equivalent to a simple "store" instruction in a conventional ISA. Further, using some of the techniques to be described later, we can limit the use of get-var instructions to the head of the clause, meaning that we are guaranteed that the environment being written into is on top of the topmost choice point and thus need not be trailed. This saves even making the simplified trail test describe above.

The instruction put-var can likewise be simplified. Since this instruction is used only on the first use of a variable (on the right-hand side), the variable is known to be unbound, no reading or dereferencing is needed, and all that need be done is

1. Initialize the variable's location in the environment to "var"#*.
2. Load the specified argument register with a reference to that location.

The unify-var instruction receives similar simplifications. Again no dereferencing is needed, because we know that the cell is supposed to be an unbound variable. Thus:

- If in read mode, copy the next object from the SP into the specified environment cell
- If in write mode, simply push a variable entry on the heap ("var"#H), and store in the specified environment location a reference to H

The instructions listed in Figure 18-1 actually have two additional forms, based on the distinction between ***permanent variables*** and ***temporary variables*** to be discussed in the next section.

18.2.3 Register and Variable Optimization

A significant amount of processing in our simplified architecture involves initializing, saving, and loading argument registers and variable locations in the environment. The previous section described how to minimize the initialization. This section describes other techniques and some new instructions that remove redundant transfers.

TYPE OF PROLOG VARIABLES. Before describing these techniques, we first give several definitions about different ways of categorizing variables used in a clause. Figure 18-4 gives samples of each. In all cases it is relatively easy to see from the definition how a compiler could determine into which category a variable fits.

- A ***void variable*** is one that is used exactly once in a clause.
- A ***temporary variable*** is one whose useful lifetime does not cross the possible creation of a choice point. This includes:
    - Variables used first in the head of a clause and at most in arguments for the first goal of the body
    - Variables whose first occurrence is in the last goal of a body
    - Variables first used inside a list or structure argument for a goal and then never used outside that goal
- A ***permanent variable*** is a clause variable that is not a temporary or a void; that is, it is used in several different goals of a clause.

EFFECTS ON CODE. Identifying a variable as a ***void variable*** means that the program does not really care what value is associated with it (like an anonymous variable), and the processing of it can be minimized. Thus, if the variable's only occurrence is as a primary formal argument to the head, we need generate no code for it and need allocate no space for it in the environment. If its only occurrence is as the argument to a list or structure, we can invent a ***unify-void instruction*** which simply skips that argument with minimal processing. In read mode we simply increment the SP; in write mode we allocate a cell for the variable on the heap (in case some other procedure needs it). A void's occurrence anywhere else can be treated either normally (i.e., allocate space in the environment for it and do a put-var where required), or by inventing a new tag type, ***anonymous variable,*** and modifying the unification process.

p(t,u,v,n,n):−q(w,u,g(x)), r(v,f(y),y,m), s(x,v,z,z).

void variables: t, m, w

temporary variables: u, n, y, z

permanent variables: v, x,

FIGURE 18-4
Types of clause variables.

A ***temporary variable*** is one that does require processing but that also does not absolutely require storage in the environment. Temporaries are variables whose initial and final uses are quite close together in the code and never extend beyond a call or similar instruction. They serve in the clause only as message passers between formal arguments. Consequently, the compiler can track all uses of them without concern for saving and restoring them across a call to some other goal. In such cases we are perfectly safe in using a spare argument register (if one is available) for the variable instead of a memory cell. This saves both space in memory and time, since instructions that manipulate registers are often faster than ones that manipulate memory.

To signify this distinction, our full WAM architecture will include two forms of the xxx-var and xxx-val instruction discussed above. The xxxPvar and xxxPval are identical to the above forms, with the "P" denoting that the variable referenced is a "permanent" one stored in the environment. The new xxxTvar and xxxTval forms will signify that the variable is a "temporary" one stored in an argument register. The functioning of these are minor modifications to the "P" type, namely, when they must be initialized to an unbound variable, and when an entry is allocated on the heap.

Note that the putTval instruction in particular becomes quite simple: It is essentially a register to register move instruction.

Use of temporary registers reduces the number of memory locations needed for a clause. If this number reaches zero, we can consider deleting the allocate entirely from the clause code. This does, however, require some of the call/return optimizations to be addressed in Section 18.2.5.

Two cases where this almost always occurs are when the clause represents a fact or when there is exactly one right-hand-side goal. In both cases, by the above definition there are no permanent variables, and thus if there are sufficient argument registers available in the machine, we need allocate no environment.

Finally, our notion of a temporary variable can be extended to cross some goals that do not require choice points to be built but which can be constructed out of basic machine instructions—e.g., many built-ins such as ">", *is,* "!". Thus, in

p(*x*,**f**(*x*),*y*,*w*):−*x* **is** *y*+1,*w*>*y*,**q**(*z*,*y*,*x*).

the goals use **is** and > are often converted into simple machine code to do the operations, so there is no need to worry about objects that are in registers being destroyed by a call. Consequently, in such cases we can declare *w, x,* and *y* to be temporary variables, and *z* a void. This means that if there are sufficient registers we need not allocate any local memory for this clause.

Strings of goals for which only the last one actually generates a choice point, or performs some other potentially register-destroying operation, are often called ***chunks.***

REGISTER ALLOCATION FOR TEMPORARIES

(Debray, 1986; Janssens et al., 1988)

Given that certain variables in a clause are denoted as temporaries, they can be allocated to machine registers and the environment shrunk accordingly. This allocation can be very simple; for example, we can keep a list of free registers, and when a temporary variable is encountered for the first time, it can be allocated to one of the currently free registers. After the variable's last use, we free the associated register back to the list.

As simple as this is, it is nowhere near optimal. A variety of modifications can greatly improve the results. For example, if one of the first uses of a variable is as a top-level variable in the head literal, then it makes sense to allocate the variable to the matching register (it is already there anyway). Another approach is to look at the goal literal at the end of the current chunk and see if the temporary variable in question appears anywhere as a top-level argument. If it does, then one might try to allocate the variable to the argument register corresponding to that position. Consider:

$$\mathbf{p}(x,3){:}-\mathbf{q}(\mathbf{f}(x,y),y).$$

All variables here are temporary, and an assignment that would make sense would be to use A1 for x and A2 for y. A1 already contains x at entry, and y is needed in A2 when **q** is called.

Needless to say, there are often complications when using such strategies. The same temporary may be used in several top-level places, both in the head and in the last goal on a chunk. Or a variable may appear at the top level of both the head and the first goal, but in different positions and in such a fashion that some other argument must be processed first, or even worse, such that neither argument for the goal can be built before the other one is first moved out of the way. In such cases it may be necessary to carefully specify the order in which arguments are unified in the head and/or built for the goal. It may also be necessary sometimes to give up and do a move of one object to a free register or local memory cell, and later move it back to its desired position, just to free up its original register for some other variable.

As an example of this, consider $\mathbf{p}(x,y){:}-\mathbf{q}(y,x)$.

AN EXAMPLE. As an example of the simplifications possible, we refer back to Figure 17-22 [repeated in Figure 18-5(*a*)], where all the variables are temporary. If our target machine has four extra argument registers, A4,...,A7, then we can replace the getv Vi,Aj by a getTvar A(3+i),Aj, the putv Vi,Aj by a putTval A(3+i),Aj, and unifyv Vi by unifyTxxx A(3+i), where xxx is var or val as appropriate. There are now no permanent variables, meaning that we can also eliminate the allocate. Figure 18-5(*b*) lists the resulting code for the same clause.

Further optimizations are also possible. Looking at the original code, we see that the second argument L2 (in register A2) stays in the second argument from the head to the first goal, passing unmodified through A6. This is wasted copying. We can simply let A2 pass unreferenced and unmodified from the head to the body. Figure 18-5(*c*) lists the resulting code. Now only three temporary registers are needed.

```
.. as before        .. as before        .. as before
allocate 4          get-list A1         get-list A1
get-list A1         unifyTvar A4        unifyTvar A4
unifyv V1           unifyTvar A5        unifyTvar A5
unifyv V2           getTvar A6,A2       get-list A3
getv V3,A2          get-list A3         unifyTval A4
get-list A3         unifyTval A4        unifyTvar A6
unifyv V1           unifyTvar A7        putTval A5,A1
unifyv V4           putTval A5,A1       putTval A6,A3
put V2,A1           putTval A6,A2       call append
put V3,A2           putTval A7,A3
put V4,A3           call append
call append
```

(*a*) Original.　(*b*) Temporary optimization.　(*c*) Argument 2 optimization.

```
get-list A1
unifyTvar A4
unifyTvar A1
get-list A3
unifyTval A4
unifyTvar A3
call append
```

(*d*) Register optimization.

FIGURE 18-5
Optimization of **append** clause 2 code.

For the final register optimization, observe that variable A1 is needed only once to initialize the first list unification. Observe also that the temporary A5 is loaded after A1 is used, and then copied back into A1, with no further uses of A5. Thus, by replacing the unifyTvar A5 by unifyTvar A1, we have lost no information and we can now delete the putTval A5,A1. The same holds true for A6, yielding the final code of Figure 18-5(*d*).

Note that with all these optimizations, we need only one extra register A4, not three. The code is also considerably shorter.

Figure 18-6 gives some other examples.

18.2.4 Single Clause Optimization—Decoupled Environments

Consider what happens if there is exactly one clause that might match some particular goal (ignore for now how we might determine such cases in advance). In the simplified model a mark builds a choice point which

```
p(f(x,y),x,7):-q(x,2,y).
p(x,y,g(z)):-q(5,z,x,f(x,y)).

get-cons   7,A3     putTvar   A2,A5
get-str    f/3,A1   get-str   g/1,A3
unifyTval  A2       unifyTvar A2
unifyTvar  A3       put-str   f/2,A4
putTval    A2,A1    unifyTval A1
put-cons   2,A2     unifyTval A5
execute    q        putTval   A1,A3
                    put-cons  5,A1
                    execute q
```

FIGURE 18-6
Other register optimization examples.

contains copies of all the machine register, and the first retry instruction enters an FA entry that points to a backtrack instruction. If the clause is successful, the information in this choice point is never needed.

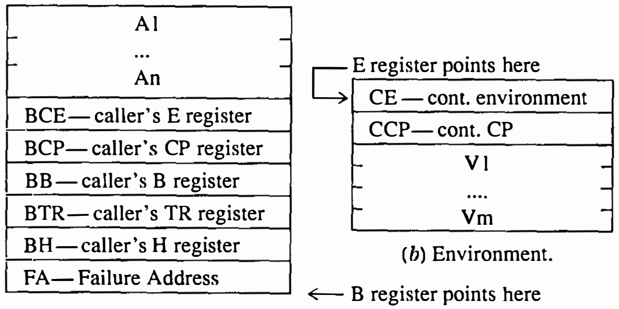
If it fails, however, the failure sequence will take the machine through the FA to the backtrack, and then through the failure sequence back to the prior choice point, from which all the appropriate registers will be reloaded. Again the information in the topmost choice point is irrelevant.

The net result is that if we know there is exactly one specific clause that might match a particular goal, avoiding the mark saves building a choice point and thus all the associated memory activity. This can be a significant savings.

There are, however, two problems. First, even with the register techniques of the last section, the clause may still need local variables which require executing an allocate. This adds an environment to the current choice point—which may cause confusion if that choice point already has an environment. Second, if the clause is successful, it needs to return to the value in the CP register at the time the clause is entered—not the values stored in the top choice point.

The solution used in the full WAM architecture breaks the space allocated to an environment away from any particular choice point and places a copy of the needed return information in this environment. This permits using an environment as either storage for local variables for some prior choice point or as a "mini" choice point when a full one is not necessary.

Figure 18-7 diagrams the new choice point and environment format. The new *CE field* contains the E register at the time the environment was created (i.e., a link to the previous environment). The new *CCP field* contains a copy of the CP register at the time the environment was created (the code to return to upon a success). As before, accessing the i-th permanent variable requires some simple addition to the E register, plus one memory access.



(*b*) Environment.

(*a*) Choice point.

FIGURE 18-7
The split choice point and environment.

Implementing this requires several changes to the prior instruction set:

- allocate now allocates n+2 locations from the top of the stack and must initialize the CE and CCP fields to the current E and CP values before resetting E.
- return should use CE[E] and CCP[E] as sources of its return information.
- The failure sequence must also reload CP from CCP[E] and E from CE[E].
- In addition, for now, all clauses must have an allocate, whether they need it for an environment or not. This restriction will be removed in the next section.

These changes guarantee that at entry to the code for any clause, the CP and E registers have the PC and E values to return to if the clause is successful. For a singleton clause or the first clause of a linked set, this is certainly true. For intermediate clauses of a set linked by retrys, the prior clause's allocate will have saved CP and E, and the failure sequence will have reloaded them.

This statement in turn guarantees correct operation for either singleton or linked clauses—whether they succeed or fail. In the success case, the return will retrieve the correct CP and matching E from the environment built by the allocate. Failures in the linked case work just as before, with the choice point built by the mark sequencing through the list of clauses to try. The final backtrack instruction then works as before.

In the single-clause case, there is no linked backtrack. Instead, the caller's choice point FA is used, with the E retrieved from the environment being the proper one. All the extra backtrack time is avoided.

18.2.5 Call/Return Optimization

The above instructions guarantee that at entry to any clause code, both CP and E give the primary information needed to return to the caller that built the goal. This, when combined with the prior definition of a *tempo-*

rary variable, can also simplify the way in which procedures are called and returned from.

First consider a *fact,* a clause with no right-hand side. From the above definition all the variables in such a clause are ***temporary,*** and thus do not need memory space in the environment. There is therefore no need to allocate storage to an environment. Further, the code for a fact consists of gets and unifys, which change neither CP nor E. Thus at the end of the code section all that is needed for a proper return is a "branch" to CP.

Inventing a ***proceed*** instruction which simply places a copy of CP into PC permits deletion of both the final return and the initial allocate. We do not bother to save CP and E, and thus we do not need the return to retrieve them.

A similar analysis can be used when a clause has exactly one goal on the right-hand side (e.g., p(...):−q(...).). Again, from our prior definition, all the variables in such a clause are temporaries, and no environment need be built. (The one exception might be if there are more variables than there are physical argument registers).

Further, the final instructions for such a sequence are a call followed by a return, where the place and environment to be returned to by the return are those specified by the CP and E registers just before the call. If we can have call avoid overwriting CP when it executes, then the procedure being called can use CP and E to return directly to the caller of the single-goal clause, without a time-wasting intermediate stop back at the single-goal clause.

The solution is to replace the call/return duo in such single-goal clauses by a single ***execute*** instruction, which simply branches to the instruction specified in its single argument with no saving of current PC value. Again, this also obviates the need for an allocate instruction in the calling code.

This technique of replacing the last call/return pair by a simple execute is so appealing that we would like to use it at the end of any nonfact clause, regardless of how many intermediate calls have been made. The problem with this, however, is that the intermediate calls have destroyed the CP value, and the typical allocate needed for permanent variables in those intermediate goals may have similarly wiped out E.

The solution to this is to invent yet another very simple instruction, ***deallocate,*** which is invoked just before calling the final goal of a clause. Its execution retrieves the CP and E registers stored in the current environment (at CCP[E] and CE[E], respectively). A final execute will then work just as with a single-goal clause.

In summary, these architectural modifications change the way a compiler generates code for a clause (cf. Figure 18-8):

- For a fact requiring no permanent variables, do not compile in an allocate, and replace the final return by a proceed.
- For a single-goal clause requiring no permanent variables, do not compile in an allocate, and replace the final call/return by an execute.
- For any clause requiring an allocate, replace the final call return by a deallocate/execute.

```
; p(...).
p: .. get
   proceed
```

(*a*) Unit clause.

```
; p(...):−q(...).
p: ..get
   ..put
   execute q
```

(*b*) Single goal body.

```
; p(..):−q1(..),..qn(..).
p: allocate
   ..get
   ..put
   call q1
   ..put
   call q2
   ...
   put
   deallocate
   execute qn
```

(*c*) Multiple goal body.

FIGURE 18-8
Code sequences.

Note that there is no longer a need for a return instruction, and that the failure sequence need not reload CP and E as suggested by the last subsection.

These three techniques are sometimes referred to in the literature as ***last-call optimizations.***

18.3 OPTIMIZING MULTIPLE CLAUSES

In many cases it is possible for the compiler to predict beforehand that under certain circumstances certain clauses will never be applicable, and thus need not even be entered for the attempt. Sometimes these circumstances can be determined at compile time for individual calls within clause; more generally they are actually determined at runtime by extra compiler-inserted code. In any case, huge time savings are possible, particularly when there are many clauses. The following subsections address such optimizations, including:

- A method for breaking one long chain of clauses into multiple shorter ones, selectable at runtime by simple tests on the argument registers
- Streamlining the code for the last clause in a chain
- Full tail-recursion optimization
- Minimizing choice-point constructions when guarantees can be made that only one clause of a chain will work, even if we cannot predict beforehand which one

18.3.1 Indexing and Hashing

A common property of clause sets that have the same head predicate symbol is that they are often separable in terms of the kinds of goal arguments they handle. One clause, for example, might handle the case where an argument is the empty list; another clause (or clauses) might handle the case where the same argument is a nonempty list; while a third set expects numbers in that position.

In such cases, once the system has determined that a particular goal has an argument of a particular type, there is absolutely no reason why any of the clause sets expecting arguments of other types should be even looked at. They are guaranteed to fail.

The ***switch-on-type*** instruction (often called ***switch-on-term***) is an excellent candidate for doing such a test. Its arguments include an argument register specification and four addresses. The argument register is dereferenced, and the resulting tag is tested. The machine branches to one of the four addresses depending on whether this tag is a variable, a constant, a list, or a structure.

If the compiler positions such an instruction just before the mark for a set of clauses, the computer can at runtime test one of the actual arguments in the actual goal and branch accordingly. If the argument is an unbound variable, then any of the clauses might be applicable, and the code at this target is essentially what was generated above. If, however, the actual argument is a constant, list, or structure, then not all the clauses need be tried, and consequently what should be compiled at the targets of these branches is code for just those subsets of clauses that might match. Further, if some of those subsets are singleton clauses, the single-clause optimization described above can be used, thus avoiding creation of a choice point. Such clauses are often called ***deterministic clauses*** because once the appropriate argument register has been tested and found to be of the proper type, there is absolutely no other clause that need be tried. It is either this one or none. There is no nondeterminism or backtracking needed.

This technique is termed ***indexing,*** and it can often buy factors or 2 or 3 in increased performance over simply trying all possible clauses. As an example, the **append** predicate of the last chapter has the property that the first clause handles cases where the first argument is the constant nil, while the second clause works only for nonempty lists. Thus, when a call or execute is made to **append,** instead of immediately building a choice point via a mark, we could execute a switch-on-type that tests the first argument in A1. The code at the target for a constant need be only that for the first clause, with no mark or retry. Likewise, the code for the list case need only be that for the second clause, also with no mark or retry.

The code for the structure case need only be a simple backtrack, since the predicate cannot handle arguments of that type.

Only for the case for an unbound variable must we perform a mark and chain via retry-me-elses through both clauses.

A potential problem with this technique is that it may require making up something approaching two full copies of the code for each predicate, one as before for the unbound variable case and another one partitioned into three pieces for the other cases. This code replication can be avoided by adding yet another simple instruction of the form:

retry L

which acts just like retry-me-else L except that the uses of L and PC+1 are reversed. The value PC+1 is saved in FA[B] and a branch is made to location L, which should be the start of code for some clause. If that clause fails, control will be passed (via a normal backtrack) back to the instruction following the retry L, which usually is another retry. Thus a string of retry instructions can chain together a subset of the clauses that exist in the full retry-me-else chain.

This solves very elegantly the problem of duplicating code; the code at the "unbound variable" target of an switch-on-type instruction can be the original retry-me-else chain, while the code at the other three labels can be sequences of retrys chaining together individual clause code sequences from this first chain.

HASHING ON VALUES. A further extension of this technique is possible. For both constants and structures, we could use the value of the constant or functor symbol to sort out some particular further subchain of clauses. For example, in the code branched to when a tested argument holds a constant, we could insert a ***switch-on-constant*** instruction which specifies both an argument register to test and the address of a ***hash table*** (usually suppressed by placing the table immediately after the switch-on-constant instruction). This table is a table of addresses, each of which points to a chain. In operation the instruction would use the value part of the argument register (known to be of type constant) to generate an index into the table of addresses. Each of these addresses is the start of a retry chain that handles only clauses whose matching argument is a constant with the same hash code.

A ***switch-on-structure*** instruction could do the same for the functor subchain. Executing such an instruction could take the functor of the structure referenced by the argument, hash it, and branch through a table to an appropriate retry subchain.

It is possible to consider generalizing this technique even further by cascading a whole sequence of switch instruction which test different argument registers, until the compiler has sorted the clause set down into the smallest distinguishable subsets.

Figure 18-9 gives the general format for the code that results from these techniques. If a particular subchain consists of a single clause, then no special chain code is needed; the address in the switch code points directly to the first instruction for that clause in the full chain. Figure 18-2 uses these techniques for the optimized ***append.***

```
; Assume n clauses have same head predicate p.

p: switch-on-type A1,A1var,A1cons,A1list,A1str

A1cons: switch-on-constant A1    A1str: switch-on-structure A1
        address Chain-c1                address Chain-f1
        address Chain-c2                address Chain-f2
        ...                             ...

           The Full Chain          A Typical Subchain
       ----------------------------------------------
       A1var: mark                 Chain-xx: mark
              retry-me-else L2                retry Ci
       C1:    code for clause 1               ...
              ...                             retry Cj
       L2:    retry-me-else L3         quit:  backtrack

       C2:    code for clause 2
              ...

       Ln:    retry-me-else quit
              code for clause n
              ....
```

FIGURE 18-9
Use of switch-on-xxx instructions.

Deciding which argument is optimal to index on is often nontrivial. However, several studies of real code have concluded that the first argument (A1) is usually as good as any. As such, many PROLOG compilers never try to select an optimal position to switch on, and simply always choose A1. This assumption may be a self-fulfilling prophecy, since many PROLOG programmers are aware of this choice and design their predicates so that this first argument is effective at indexing.

Finally, there is nothing to prevent multiple layers of switch-on-type instructions, each picking out separate registers to test. Each such test is relatively simple, and if it can reduce the effective chains, it is worth the extra code.

18.3.2 Last-Clause Optimization

The ***last-call*** optimization described in a prior section streamlined the return process from the last call in a clause's code by setting up a bypass path to skip a needless return. The same kind of technique can be used to simplify the last retry-xxx in a chain of retry-me-elses or retrys by eliminating the double execution of the failure sequence (once for the fail that terminates the last clause, and once for the backtrack instruction which terminates the chain).

The technique invents two related instructions, ***trust-me*** and ***trust.*** These are used in place of the last retry-me-else or retry in a chain, and

operate by resetting B and HB to the choice point prior to the one that is currently active (as addressed by B). The register transfers B←BB[B] and HB←BH[B], in that order, accomplish this.

Unlike retry-xxx, these instructions do not modify the FA entry of any choice point. Thus if the clause code specified by the trust fails, the single execution of the failure sequence will start up the next alternative to the choice point prior to the topmost one, just what the double backtrack is supposed to do, but without the second fail from the backtrack originally used to terminate a chain. In fact, except as a target for a switch-on-xxx for some case for which there are no clauses, we no longer need the backtrack in our revised architecture.

Figure 18-10 is a version of Figure 18-9 that includes these terminating trusts.

In addition to the obvious time savings, there is a more subtle, and often more valuable, attribute of such an optimization. The deletion of the topmost choice point essentially reclaims all the storage associated with it for use by other calls which really need it. For long PROLOG programs this form of built-in ***garbage collection*** can determine whether or not there is sufficient memory to run the program.

18.3.3 Tail-Recursion Optimization

Assume a WAM architecture in which all the above optimizations have been employed. Now a careful review of Figure 18-2 will reveal an inter-

```
; Assume n clauses have same head predicate p.

p: switch-on-type A1,A1var,A1cons,A1list,A1str

A1cons: switch-on-constant A1    A1str: switch-on-structure A1
        address Chain-c1                address Chain-f1
        address Chain-c2                address Chain-f2
        ...                             ...

           The Full Chain          A Typical Subchain
       ----------------------------------------------
       A1var: mark                 Chain-xx: mark
              retry-me-else L2                retry Ci
       C1:    code for clause 1               ...
              ...                             retry Cj
       L2:    retry-me-else L3                trust Ck
       C2:    code for clause 2
              ...

       Ln:    trust-me
              code for clause n
              ....
```

FIGURE 18-10
Use of trust instructions.

esting fact: When initially called with a list in the first two argument positions and an unbound variable in the third, executing the procedure adds *absolutely nothing* to the stack. Neither choice points nor environments are built. Further, with the possible exception of the initial variable to contain the result list, *nothing is added to the trail* (all the "variables" created in this program are in the heap, but before HB). The only change to memory is the creation of a copy of the first list on the heap, with the very last cdr pointing to the second list.

Further, the code for the two clauses is positioned between the entry point switch-on-type, so that:

- No terminating execute append is needed at the end of the second clause.
- There are a minimum of branches when the first argument is a list (basically n−1, branches where n is the length of the list).
- On the last iteration of **append** (when the first argument is nil), the switch-on-type drops directly into the code for the first clause.

The net result is code which looks almost exactly like what a good assembly-level programmer would generate for such a use of **append.** It is a tight loop with no calls or stack growth and a minimum of branches.

This is an example of what is called ***tail-recursion optimization,*** which occurs when the predicate of the last goal of a clause (particularly the last clause of the procedure) is the same as that for the head. Proper arrangement of the code fragments for the different clauses in a procedure, with a switch-on-type in the middle, can create tight loops with minimal storage requirements.

18.4 OTHER OPTIMIZATIONS

The following subsections include a variety of optimization techniques for particular circumstances, including:

- Trimming the environment as the goals are executed right to left in a clause
- Implementing cuts
- A potpourri of minor modifications stemming from all the earlier ones

18.4.1 Environment Trimming

(Warren, 1983)

Up to now the allocate instruction has been bumping the stack top by the value of its argument. This saves a specific location for each local variable used anywhere in the clause. In practice, however, as the goals are executed from left to right, fewer and fewer local variables are needed, raising the possibility of dynamically specifying how much storage to save in an environment at each goal. Figure 18-11, for example, is a clause which nominally needs five local variables (for *y1* through *y5*) but if the compiler is clever in the way they are mapped into cells in the environment, then as the various calls succeed, some of them will no longer be needed. Thus, after preparing the arguments for goal **q2,** *y5* is no longer referenced; nor are *y4* and *y3* after the call to **q5.**

```
p(t,y2):-q1(t,y5), q2(y5,y1),q3(y1,y4,y3),
        q4(y4,y3),q5(y3),q6(y1,y2).

            ; Note: t is a temporary.
            ; Assume Yi stored in environment location i.
            p: allocate
               getPvar Y2,A2
               putPvar Y5,A2
               call q1,5    ; Variables up to Y5 still needed
              *putPval Y5,A1
               putPvar Y1,A2
               call q2,4    ; Y5 never used again
               putPval Y1,A1
               putPvar Y4,A2
               putPvar Y3,A3
               call q3,4
              *putPval Y4,A1
               putPvar Y3,A2
               call q4,3    ; Y4 never used again
              *putPval Y3,A1
               call q5,2    ; Y3 never used again
              *putPval Y1,A1
               putPval Y2,A2
               deallocate   ; none of the variables used again
               execute q6
```

*Change these putPval instructions to put-unsafe-val.

FIGURE 18-11
Sample environment trimming.

A storage savings can be made in such situations if the environment for the clause (**p** in this case) happens to remain on the top of the stack from call to call and is not buried by other choice points. This might be the case if, for example, the various **qi**'s are built-in predicates which consume no stack space.

The process of saving this storage is called ***environment trimming,*** and requires changes to call and allocate. The modified instruction call now has two arguments, the label to branch to and the number of variables still needed in the current environment. There are a variety of ways of capturing this argument, the simplest of which (Dobry, 1987a) has call deposit this argument in a new ***N register.***

The instruction allocate needs more substantial changes. First, it no longer has an argument indicating the number of variables in the current clause. Thus it does not automatically bump the stack by the number of needed locations. Instead it will use the N register to determine how many variables are in the last environment and test to see if this environment is on the top of the stack. To do this, it compares E and B. If E is less than B, then at least one choice point has been built on top of the currently active environment, making it impossible to trim off any memory cells. In this case B is the last-used location on the top of the stack, and a new environment must be built after it.

If E is greater than or equal to B, then the environment to be trimmed to N cells is on top, and consequently the new stack top is found by adding 2 plus the contents of the N register to E. This guarantees that the N variables still needed are protected, while still releasing locations in the prior environment that are no longer needed.

The deallocate instruction need not be changed. It still resets E to that for the caller.

Note that if no call in a clause builds a choice point over the environment for that clause, then at a successful return from the clause, the entire environment is dropped, and any more pushes to the stack will overwrite it. This can save significant stack space.

UNSAFE VARIABLES. While the above technique saves storage on the stack, it does introduce one problem. Consider what happens to environment variables that are used to prepare some goal and then are "trimmed" off. In Figure 18-11 this includes *y5* just before the call to **q2,** *y4* just before the call to **q4,** *y3* just before **q5,** and *y1* and *y2* before **q6.** It is possible that the calls to the indicated predicates will bind values to these cells, and somehow pass references to the cells to other variables. If environment trimming eliminates these cells, it is possible that dereferencing these other variables may produce a ***dangling reference*** to a cell that no longer contains the value it is supposed to.

To solve this problem we must guarantee that such variables keep their values. In some cases, such as *y2,* we are safe because the variable had to be bound to something in some other environment (*y2* is an argument of the clause head). In other cases, however, we must ***globalize*** the variables by shifting them to a location on the heap which will not go away after a trimming. This need only be done at the last possible reference to the variable (the "*"ed locations in Figure 18-11) by inventing a new instruction ***put-unsafe-value***, which works just like ***put-value*** does when the indicated variable dereferences to a value or a variable in some other environment (i.e., an address less than E). However, if the variable is unbound and is in the current environment, then the variable is trailed if necessary, a "var"#H is pushed to the heap, and a "ref"#H is stored in the designated argument register.

Variables that require such treatment are called ***unsafe variables,*** and are formally defined as being permanent variables that did not occur first either in the head's arguments (where they would be bound to something in the caller) or inside some complex structure for some prior goal (where they would have been bound to the heap anyway).

18.4.2 Cut Storage Optimization

Many of the previous optimizations, such as last-clause, last-call, and tail-recursion optimization, all help reduce stack growth for ordinary PROLOG programs. Such reductions in turn increase the opportunities for the environment trimming of the last section.

The advantage of all these is that they can be performed without the programmer being aware of them. Given this, an obvious question to ask is whether or not any programmer-specified activities can also provide the system with information which minimizes stack growth. The answer is yes, particularly when the ***cut*** predicate is used.

Conceptually, a cut signifies that the program will never backtrack to any of the goals between the head of the current clause and the cut, and that all clauses with the same predicate name as the head are to be ignored if the goal to the right of the cut fails.

These two comments mean that the program will never revisit any choice points between the head and the last one just before the cut. This is even better than the last-clause optimization: Not one but a potentially unbounded number of choice points can be considered for discard (see Figure 18-12).

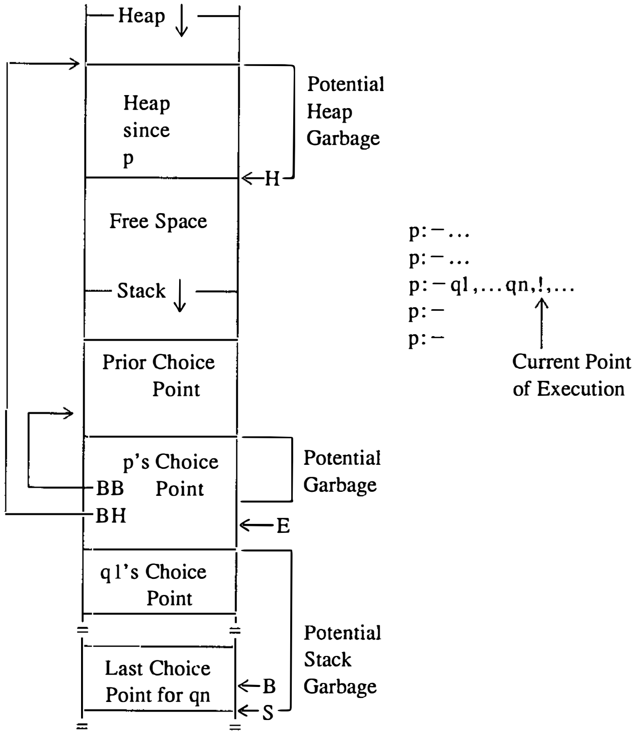


FIGURE 18-12
Garbage created by a cut.

RECOVERING STACK SPACE
(Dobry, 1987a, pp. 72–78)

Recovering the storage associated with the now useless choice points is fairly straightforward. Conceptually, BB[E] points to the topmost choice point to keep. All after it are garbage. We need only reset B to this point and continue.

One problem with this is that given the previous optimizations, we may have already discarded the choice point at E. One solution (Figure 18-13) is to include in the header of each environment an extra cell containing the B at the time the environment was created, along with a flag bit that indicates if there is a choice point associated with the current environment.

Now, when a cut is executed, B is updated from the environment at E. If the flag is set, reset it, and reset B from the choice point at B. This will locate the appropriate choice point.

The observant reader may note that there is still an environment at E above the new B that cannot be removed, and it is unclear how much of the space between them can be reclaimed. One could either:

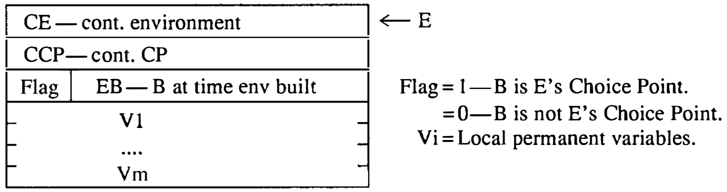
- Wait for the appropriate ***deallocate*** instruction to free this environment and the intervening space
- Copy the environment right after the just-located choice point, fixing all pointers in the process
- Create two separate stacks in memory, one for choice points and one for environments

It is unclear which of these is preferable.

RECOVERING HEAP SPACE
(Barklund and Millroth, 1986)

The prior mechanization of cut recovers stack space, but it does not recover extraneous heap space. This is a much more complex problem, because it is entirely possible that parts of the heap represent structures bound to variables that are still valid in the program (i.e., variables in either the current or earlier environment), even though they were built by procedures whose choice points have now been discarded.

One solution is to invoke a garbage collector as described for functional languages (Chapter 8) to reclaim anything associated with variables still on the stack. This is often very time-consuming, even if done on an incremental basis. It is also not without its problems, because of the need to keep the appropriate pointers to heap boundaries in the stack in appropriate order.



CE — cont. environment
← E
CCP— cont. CP
Flag
EB — B at time env built
V1
....
Vm
Flag = 1 — B is E's Choice Point.
= 0 — B is not E's Choice Point.
Vi = Local permanent variables.

FIGURE 18-13
Revised environment to handle cuts.

A variety of suggestions have been made to overcome this, none of which is terribly appealing. One, for example, would invent a ***garbage cut*** instruction, which functions exactly like the cut defined above to recover stack space but which also invokes a conventional ***mark-sweep*** garbage collector on the portions of the heap that may contain garbage. Careful placement of this in programs, especially deterministic recursive ones, can free up storage at a minimal time cost. Profligate use of it, however, can result in very excessive time penalties.

18.4.3 Other Optimizations

Following are some other modifications to the above architecture that either are in or have been suggested for incorporation in the WAM architecture:

- The ***mode flag*** in the WAM could be eliminated by providing two sets of unify, get-list, and get-structure instructions, one set that works in read mode and one set that works in write mode. The write-mode unifys could be used exclusively after puts, since such instructions always set the flag to read.

  Likewise, a ***switch-on-type*** instruction could be put in place of a get-list or get-structure instruction. In either case, if the indicated argument register holds an unbound variable, a write-mode get and a sequence of write-mode unify's would handle the situation.

  For an argument expected to be a list, finding a list in the actual argument would cause a branch to a read-mode sequence. Either a constant or a structure would branch to a fail instruction.

  An argument expected to be a structure would be handled similarly, with branches to fail if the argument is actually a constant or list.
- If the compiler guarantees that it never modifies an A register holding an argument before the head unification code completes, then reloading the argument registers can be deleted from the fail sequence and moved to the full backtrack sequence. This could save significant register reloading.
- The head unification tests on the argument registers need not go in exact order A1, A2,.... In many cases it may be beneficial to reorder these tests, such as doing all get-constants first, then all get-vals, etc., and leave the complex list and structure tests until last. This would detect all the obvious mismatches before attempting the expensive compares.
- Two new instructions, ***try-me-else*** and ***try,*** can take the place of an initial mark retry-me-else or mark retry sequence. These instructions include the activities of the mark.

  The mark instruction can then be deleted from the WAM. [This is in fact what is done in many definitions of the WAM, including Warren's, (1983).]

- The top of stack register S could be eliminated. Whenever its value is needed, the machine could determine it by looking at the larger of B and E (plus the number of arguments at E). Computing this latter point may require expanding the header of an environment to include the number of words in it. The allocate instruction would be responsible for filling this location.
- Instead of building lists out of two cells, include in each tag an extra ***cdr code bit*** which records whether or not the next cell is a pointer to the next **car** item. [A variation by Dobry (1987a) uses this bit to indicate whether the item is a car or cdr item.] By inventing special unify instructions to process down lists, some savings in both storage and time (tracing cdr pointers) might be possible.

18.5 ANNOTATIONS AND BACKTRACK CONTROL

All the optimizations addressed so far have been low-level ones that do not require much knowledge on the compiler's part of what the program is doing globally. The following subsections address some of the optimizations that can be obtained when such global understanding is given the compiler. In particular, this includes:

- Having the programmer (or a smart analysis program) ***annotate*** a program with information about the actual types of clause arguments and the expected directions of substitutions in head-goal unifications
- Using information from unification failures to be smart about which *choice point* to back up to on a backtrack

Unlike the prior optimizations, the techniques covered here are still topics of intense investigation, and are just beginning to be incorporated into real PROLOG compilers. For that reason, we will not go into as much depth on them as we did for prior techniques. The references should be seen for more detail.

18.5.1 Annotation and Modes

(Warren, 1977; Mellish, 1981; Van Roy et al., 1987)

Conventional programming languages usually permit (or require) programmers to provide additional information about identifiers used in a program. Such information is often called ***annotation,*** and it allows a compiler to optimize the storage representation given objects, and to specialize the code it produces to deal with them. There are usually three kinds of such annotations:

- ***Basic type*** information indicates whether the identifier is to hold an integer, floating-point number, character, etc.

- ***Structure*** information indicates any internal details of the object, such as whether it is an array, a record, a procedure, etc.
- ***Mode*** information indicates how an identifier's bindings are generated and used when the identifier is a formal argument to a procedure.

Examples of the last kind include the in and out notations of Pascal (which signals whether a procedure argument has a value at input—***input substitution***—and whether or not its value will be changed by the procedure—***output substitution***), and the various semantics for parameter passing found in conventional languages, such as ***call-by-value, call-by-result, call-by-name, call-by-reference,*** etc.

In conventional programming the first two kinds of annotation are deemed the most important, because they directly control the kinds of instructions that are generated to carry on the desired computations. Many languages have extensive ***typing systems*** to permit tight user control over their attributes. The third kind (modes) is often considered a "necessary evil" that must be included to satisfy the language design.

As we have seen with PROLOG, logic languages (and many functional languages as well) do not have the same view. Identifiers are often "typeless" and may take on any kind of object during execution of a program. For most functional languages, argument passing is only one-way, so modes are largely irrelevant. For logic languages, however, unification can generate substitutions going in either direction (back to the caller or into the called clause's variables), and any kind of information the compiler can derive about which ways may or may not occur will permit more optimized code. Such information can be derived by explicit programmer annotations, implicit annotations derived by the compiler, or a mixture.

TYPICAL MODE ANNOTATIONS AND THEIR USES. Formally, a ***mode*** is an indication of the kinds of bindings that an argument in a goal will have when a clause (or set of clauses) with a matching head predicate name are checked via unification. As might be guessed from our discussions on the WAM, the most important such information has to do with whether or not the actual argument has a current binding, or is expected to receive a binding if the clause runs to completion. In cases where an argument is known to have a binding, it is also often useful to know what kind of object is bound.

For example, consider the clause p(3,*y*,f(*y*)):-.... If the compiler is told that all actual goals with the predicate p will be ***ground terms*** (terms with no embedded unbound variables), then we can replace the get-con instruction for 3 by a simpler instruction that simply compares the value of the argument register with 3. The code need not worry about the case where the actual argument is a variable that might have to be trailed. Likewise, the code for unifying the third argument could also be simplified to a special get-str and unify that work only in read mode. We need not check for variables in the actual argument, which in turn would require building a structure on the heap.

Similarly, knowing that the actual arguments are unbound and independent variables permits specialized code that runs only in write mode.

Figure 18-14 lists some typical annotations that have proven useful to logic languages. The most common of these are "++," "+," "−," and "?". "++" indicates that an argument is a ground term with no embedded unbound variables. "+" indicates that it is either ground, a list, or a structure where the main functor is ground. There may be unbound variables, but they are subcomponents of the object. "−" indicates that the dereferenced argument is an unbound variable that expects to receive a value as a result of this call. Common variations of this address whether or not the final variable is an ***independent variable*** that is not accessed by any of the other arguments, or a ***coupled variable*** that is shared somehow with other arguments. "?" indicates that the argument could be anything.

The other two annotations are used less frequently, and are specializations of the "+" mode when more information is known about some components of an argument.

Many PROLOGs permit programmer indication of these modes for specific clauses through statements called ***mode declarations,*** with syntax as shown in Figure 18-15. When placed in a program, such a statement indicates to the compiler that it can assume that the arguments for any goal which calls the clause(s) that follow the mode statement will have arguments with modes as indicated. In Figure 18-15(*b*) this means that the append will always be ***determinate***; that is, the first two arguments will always be provided, with the first one ground, and the third will be computed by the clause.

Mode	Typical Symbols	Comments
Ground	+ +,g	Argument expression has no unbound variables in it.
Input	+,i,in	At least main functor is ground.
Structure	s(M1,...,Mn)	Argument is structure arity n, with k'th argument of mode Mk
List	l(M1,M2)	Argument is list, with M1 mode of car, M2 mode of cdr.
Output	−,o,out	Argument always unbound variable.
Input/Output	?,io,inout	All other possibilities.

Note: modes refer to goal arguments after full dereferencing.

FIGURE 18-14
Typical mode annotations.

```
<mode-statement>: = : − mode(<mode-symbol> {,<mode-symbol>}*).
<mode-symbol>: =  + + | + | − | ?
```

(*a*) Typical mode statement syntax.

```
:-mode(+ +, +, −).
append(nil,x,x).
append((h.L1),L2,(h.L3)): − append(L1,L2,L3).
```

(*b*) Sample usage—determinate append.

FIGURE 18-15
Mode declarations in PROLOG programs.

Besides improving the unification part of a clause's code, mode information also permits several other potential optimizations. First, knowing which arguments are ground permits intelligent choice of arguments to index, or hash, on. This can quickly reduce the set of clauses that must be searched. A fringe benefit may be a reduction in storage for programs, since several of the chains may not be necessary.

In a related fashion, knowing modes also often permits early identification of ***deterministic clauses***—clauses which, if the head unifies, are the only possible clause that might match the goal. This permits early recovery of storage from the stack, and potentially the heap.

Given the mode statement of Figure 18-15(*b*), both of the **append** clauses fall in this category. Knowing that the first argument is ground guarantees that any real goal can match at most one of these clauses. No choice point need ever be built, and the associated instructions need never be executed.

MODE-PROPAGATION ANALYSIS

(Debray and Warren, 1986)

Mode statements typically indicate only the state of arguments before a goal is unified with a particular clause. Further optimizations are possible, however, if the mode of clause variables can be tracked as the clause is executed. For example, one might use the mode declarations to predict what kinds of bindings are present in variables in the caller's environment when the clause completes. These *output substitution modes* could then be used in analyzing much more accurate modes for the next goal in the caller's clause. Optimized code for each such call could then be compiled.

Such an analysis could be carried even further. For example, it may often be possible to start with mode declarations for just the original query (or range of queries), and predict the modes of all calls that might exist during the program's execution, without any explicit mode declarations for individual predicates.

The trick for such ***mode analysis*** is to break the analysis in half. For each derived mode declaration (corresponding to a call to a predicate, and called a ***calling pattern***), such an analysis would compute for each potential clause whether or not that clause has a chance of being applicable, and then what the equivalent modes of the original arguments would be after the clause completed successfully. These computed argument modes are termed the ***success pattern.***

Once all possible success patterns have been computed for a particular calling pattern, they are joined together to compute a single "worst-case" success pattern. For example, for some particular predicate a derived calling pattern might take the form (?,++,?,?,+), and the above analysis on three clauses with the same predicate might yield success patterns of (?,++,?,++,++), (++,++,?,++,++), (++,++,++,+,++). Not knowing which of these might work in reality, we would assume the least common denominator, namely (?,++,?,+,++), as the final success pattern.

The usefulness of such a success pattern should be apparent. If one is trying to compute code for a particular call from a particular goal, knowing the minimal modes of the arguments after the call permits derivation of the modes for the containing clause's local variables. This in turn permits development of accurate modes for the next rightmost goal.

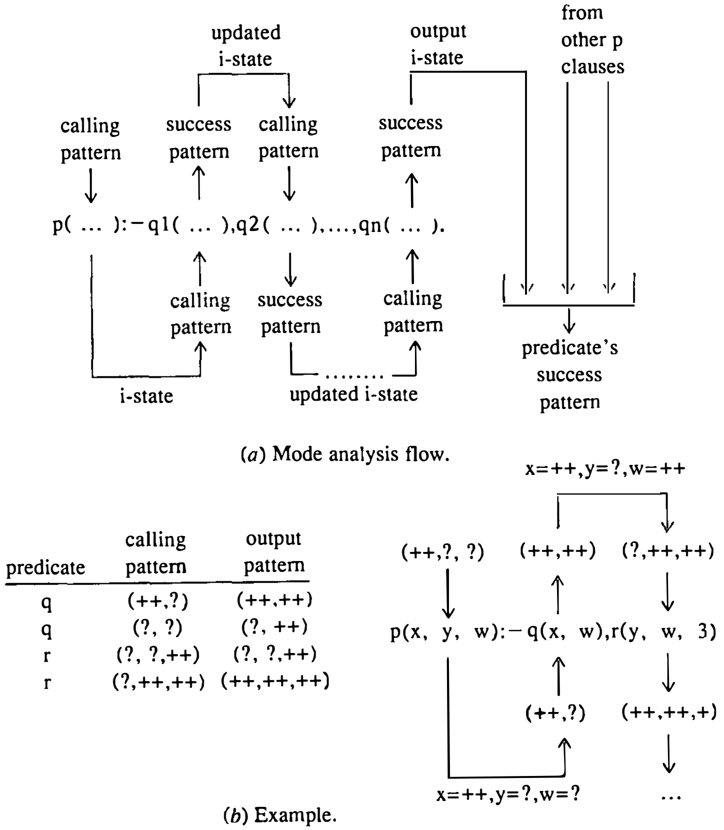
Figure 18-16 diagrams a simple picture for such an analysis, and an equally simple example. In this diagram, the calling pattern of modes is used in conjunction with the formal head arguments to develop an initial estimate of the modes of any local variables that appear in the head. This information is called the ***instantiation state*** or ***i-state.*** It is then used to compute a calling pattern for the first goal on the right-hand side. That call's matching success pattern will then be compared with the goal's formal arguments to see what local variables' modes could be updated. These updates then drive the calling pattern for the next goal, and the cycle repeats until the last goal has been processed. At that point, all other clauses with the same original predicate are processed similarly, and the resulting set of success patterns is combined to form a single result.

In real life there are many complexities that have been glossed over here, such as recursive calls, effects of variables shared among multiple arguments, soundness of the analysis, etc. However, the potential benefits are such that a fair amount of current research is being done in the area.

18.5.2 Backtracking

The normal PROLOG execution model is top down, left to right. This means that in terms of a deduction tree a failure to satisfy a goal literal in the body of some clause causes a backtrack to attempt to resatisfy the rightmost goal associated with the proof of the goal literal to the left of the failing one.

Even with the best of programmers, very often this retried goal has nothing to do with the unsatisfied literal, so a vicious cycle of repeating exactly the same failing computations is followed over and over again until a backtrack is pushed to the point where the value of some variable in the failing goal is changed. Our normal PROLOG model is said to exhibit ***naive backtracking.***



(*a*) Mode analysis flow.

predicate	calling pattern	output pattern
q	(++,?)	(++,++)
q	(?, ?)	(?, ++)
r	(?, ?,++)	(?, ?,++)
r	(?,++,++)	(++,++,++)

(*b*) Example.

FIGURE 18-16
Automated mode analysis and propagation.

Figure 18-17 diagrams some samples of this phenomenon. Consider example (*a*): PROLOG will solve **p** (and give *x* a value), and then attempt to solve **r**(*y*,*y*). It will run through all three clauses for **r**, find no match, and backtrack. This in turn will give *x* a different value, and it will try **r** again. However, nothing has changed, so **r**(*y*,*y*) goes through exactly the same calculations it did the last time, with the same result. In fact, in this case we will repeat the same process 10,000 times before finally declaring no solution.

Example (*b*), query 1, is another example. Assume that the first time through the clause for **p** that the solution for goal **q**(*a*,*b*) assigns values to *a* and *b* (*x* and *y* in the original query), and that **r**(*b*,*c*) uses the *b* value to generate a *c*. Now assume that **s**(*a*) fails—the value given to *a* does not work in **s**. PROLOG will backtrack and get a different value for *c*, perhaps after a long sequence of computations for **r**. However, *c* has

```
p(1).
 ...
p(10000).
r(0,1).
r(1,2).
r(2,3).
?-p(x),r(y,y).
```

(*a*) 30,000 worthless attempts.

```
p(a,b,c):-q(a,b),r(b,c),s(a).
q(x,y):-u(x),v(y).
r(x,y):-m(x),o(y).

?-p(x,y,z). ;Query 1
?-p(x,y,x). ;Query 2
?-p(A,y,z). ;Query 3 assume s(A) fails.
```

(*b*) Other examples.

FIGURE 18-17
Examples of naive backtracking.

no effect on *a*, so we again repeat the computations for **s** *with the same value for a*. As before, this is wasted computation. We repeat over and over until we eventually (we hope) exhaust **r**(*b*,*c*) and force a backtrack to **q**, which might do some good.

The following subsections describe alternatives to PROLOG's naive mechanism that can substantially improve a logic program's execution times by skipping calculations that are known to be redundant. The reader should be cautioned again that such techniques drift away from the simple PROLOG model, and in some cases may actually be unusable in a PROLOG system because of conflicts with extralogical constructs such as **cut** or **assert/retract.**

INTELLIGENT BACKTRACKING

(Bruynooghe and Periera, 1985; Cox, 1985)

One's first reaction to the prior examples should be "Anyone can see when a literal has no effect on a goal that fails—simply look at the variables it uses and compare with the failing one. An ***intelligent backtracker*** will ignore those with no intersection."

In most cases such an analysis is both simple and correct. However, it is not always so. Consider query 2 of Figure 18-17(*b*). Here *a* and *c* are bound to the same variable *x*, and skipping the **r** literal may actually skip some of the possible values that might be bound to *a*, including one that would satisfy **s.** The conclusion from this is that to be correct we must consider the actual dereferenced variables referred to in a failing goal literal and backtrack to the last literal goal which also referenced one or more of those variables, regardless of what the clause called them. In this example the failing literal **s** refers to the original query variable *x*, and the **r** uses the original variables *x* and *y*. The local clause names for these variables (*a*, *b*, and *c*) are irrelevant.

To demonstrate another complicating factor, consider query 3 in example (*b*). Here the constant **A** is bound to the variable *a* by the original query. There is no alternative. Thus, when **s** fails, an intelligent backtracker should recognize that there are no real variables whatsoever associated with **s(A)**, and thus we can backtrack through the entire **p** rule and avoid any alternatives from either **q** or **r**.

Next, if we really want to be optimal, we must look at more than just the literals in the current goal. Consider query 1 for example (*b*), where we backtrack from **s** to **q** directly. Assume further that the way **q**(*a*,*b*) was solved was with a clause of the form:

q(*a*,*b*):-**q1**(*b*),**q2**(*a*),**q3**(*b*).

A normal backtrack to **q** would require restarting **q3** to look for another *b*. Only if that failed would we backtrack to **q2** where a new *a* (or *x* in this case) is possible. A really efficient backtrack mechanism would recognize this and backtrack through **q3** directly to **q2.** We could in fact repeat this process ad infinitum on the clause that was used to solve **q2**, etc.

This means that a really intelligent backtracker would look not just at the goal literals for the clause containing the failing goal, but at the whole string of goals (essentially the entire stack of choice points), and go backward one by one until one is found which binds some of the variables used by the failing goal.

Finally, a ***genius-level backtracker*** would actually watch the patterns of unification between the failing goal and its possible clauses, and attempt to identify if there were any specific variables which caused all the failures. If so, the set of variables used to look backward through the choice points could be reduced, further sharpening the identification of the exact goal literal to retreat to.

Of course, all of these approaches fall apart if any of the literals have side effects, such as a cut operator, an **assert,** or a **retract.** Such cases essentially set up walls over which any kind of superbacktracking must be made very carefully. What usually must happen is that smart backtracking must stop just before (to the right of) such predicates and allow PROLOG's normal naive mechanism to bridge the gap.

DATA DEPENDENCY GRAPHS. The above examples should convince the reader that more than simple observations of the variable usages in a clause are needed to do a complete analysis of how best to backtrack in case of a failure to satisfy a goal. What is needed is a graph of which literals use and need which variables. Such a graph is called a ***data dependency graph,*** and will be useful not only for better backtracking, but also in later discussions on parallel logic systems.

A data dependency graph consists of a set of nodes interconnected by arcs. For each literal involved in a proof tree there is exactly one corresponding node. Coupling these nodes are arcs which indicate data dependencies. Each arc is labeled by a variable name. The node from which such an arc leaves corresponds to a literal whose proof binds a value to the variable named on the arc. Such a node is called the ***generator*** of the specified variable.

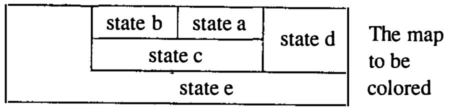
The node pointed to by an arc corresponds to a literal which uses a copy of that variable in it; i.e., when it is executed the prior node had already bound a value to the variable, and this node is simply using that variable in a unification. For obvious reasons, such nodes are called ***consumer*** nodes.

A node may be both a generator and a consumer, although not of the same variable simultaneously. Further, depending on the actual arguments presented to a clause, this pattern of usage may not be constant. In a determinate computation, for example, the head literal of a rule may be the generator for some variables (the inputs) and a consumer for others (the result). In an indeterminate computation, however, we may start off with the head generating a value for the result variable and consuming values for the nominal input variables.

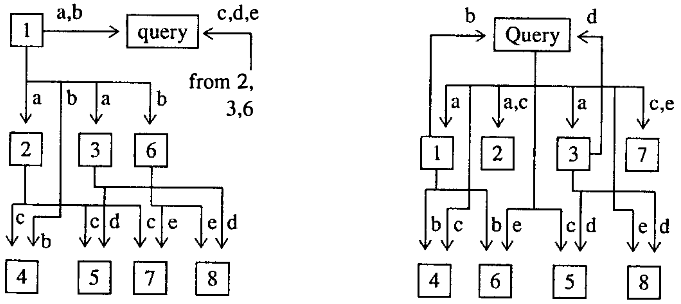
Figure 18-18 diagrams some data dependency graphs for a simple ***map coloring problem.*** The clause given states that there are five states to color, *a* through *e*, and that they are "next" to each other as shown. The predicate **next** has 12 clauses in it, of the form **next**(red,blue),.... Each clause indicates a valid assignment of colors to two states that are next to each other. The only way the original clause can succeed is if there is an assignment of colors that never has two states touching that are of the same color.

Figure 18-18 also includes two data dependency graphs for two queries. In each graph the number in a node corresponds to a goal literal in the clause defining **map.** The first graph corresponds to the case where all the argument variables are initially unbound [i.e., a calling pattern of (−,−,−,−,−)]. The second one corresponds to the case where some of the arguments have been fixed before starting the proof [calling mode (++,−,++,−,++)]. Note also that these graphs assume the **next** predicates are defined exactly as shown previously; regardless of their calling patterns, they always return a success pattern of (++,++), which in turn means that after a call to them both calling arguments are bound to ground terms. Note also the notation that numbers the nodes by their position in the original clause.

The meaning of the graphs is straightforward. In the first example, the first goal, **next**(*a*,*b*), will always assign a value to both *a* and *b*. It is a generator for both. The second goal literal, **next**(*a*,*c*), consumes *a* (it occurs after literal 1, and if literal 1 produces a value, then it will use it) and generates a value for *c*. Literal 3 does the same for *a* and *d:* It consumes *a* and generates *d*. Literals 4 and 5 are consumers only. They test the values generated by the first three literals for consistency. Literal 6 consumes *b* and generates *e*. Literals 7 and 8 then consume *c*, *d*, and *e*, and check for consistency. Finally, in this case the head literal **map**(*a*,*b*,*c*,*d*,*e*) is a consumer for all the variables.



```
map(a,b,c,d,e):-next(a,b),next(a,c),next(a,d),next(b,c),
                    1         2         3         4
                next(c,d),next(b,e),next(c,e),next(d,e).
                    5         6         7         8
next(red,blue).      next(blue,red).      next(yellow,red).      next(green,red).
next(red,yellow).    next(blue,yellow).   next(yellow,blue).     next(green,blue).
next(red,green).     next(blue,green).    next(yellow,green).    next(green,yellow).
```



(*a*) ?−map(a,b.c,d,e).

(*b*) ?−map(red,b,green,d,blue).

FIGURE 18-18
Data dependency graphs.

The second graph is similar except that the head generates *c*.

It should be obvious that a variation of the ***mode analysis*** algorithm discussed above could be used to compute such graphs.

SEMI-INTELLIGENT BACKTRACKING

(Chang and Despain, 1985; Kumar and Lin, 1988)

Really intelligent backtracking requires recomputing the data dependency graph dynamically at runtime. This is often expensive to the point of wiping out the computational savings of minimizing the backtracks.

A ***semi-intelligent backtrack*** algorithm computes a data dependency graph at compile time, and uses that to encode into the WAM code information as to which literal to backtrack to at different points along the execution of each clause. This literal would be in the clause containing the failing literal, and as such would not necessarily be as deep as one derived from a fully runtime analysis.

There are three parts to this encoded information. First is a ***Backtrack Table,*** which is built by the compiler and included with the code for each clause. It indicates for each goal literal in the clause the number of the literal to backtrack to if that literal fails. For reasons to be discussed later, there are actually two such pointers in each entry.

Next is a ***choice point table,*** which is built by the WAM at runtime to record for each clause the choice points (B register values) that correspond to each goal literal attempted so far. Upon a backtrack from goal literal k in some clause, the k-th entry in the backtrack table will determine which literal in the clause to backtrack to (say, the j-th), and then the j-th entry in this choice point table will provide the B register value that points to the matching choice point.

Finally, storage for a variety of status flags and literal numbers in choice points and environments will help determine which column to use in the backtrack table.

Figure 18-19 diagrams how all this information plays together when execution of a WAM program requires a backtrack.

Entries in the Backtrack Table come from an analysis of a data dependency graph for the clause. For each goal literal we determine which literals are generators of variables consumed by the literal that fails. These are the nodes from which arcs lead to the failing node, and are thus candidate literals to backtrack to. Given PROLOG's left-to-right ordering, the one actually chosen for inclusion in the Backtrack Table is the rightmost one of these. A failure out of the leftmost literal must be handled in the usual way.

Figure 18-20 lists such a derivation for Figure 18-18. For example, the fourth literal **next**(b,c) consumes b and c, which were generated by goals 1 and 2, respectively. If it fails, there is no sense in backtracking to literal 3, since it does not change any variable consumed by 6. The choice of the nearest of the two literals 1 and 2—namely, 2—is made to reflect PROLOG operations.

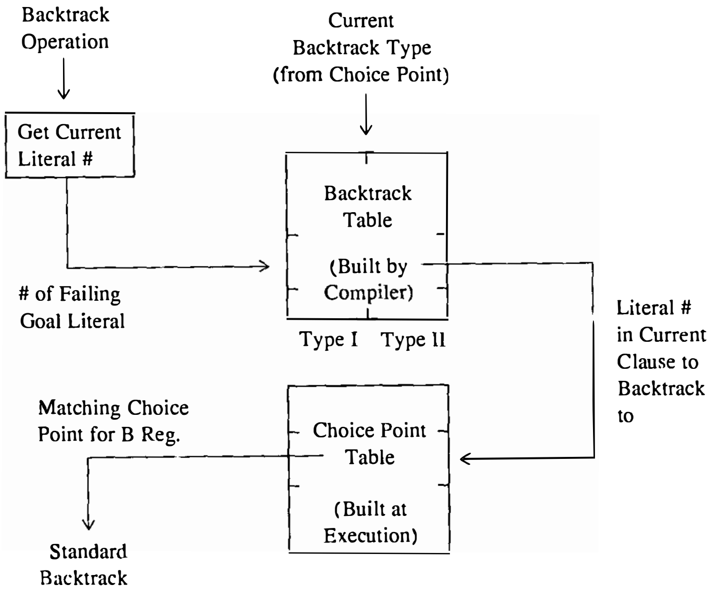


FIGURE 18-19
The semiintelligent backtrack operation.

Literal	Variables Consumed	Generating Literals	Backtrack Table Entry
1	–	–	*
2	a	1	1
3	a	1	1
4	b,c	1,2	2
5	c,d	2,3	3
6	b	1	1
7	c,e	2,6	6
8	d,e	3,6	6

Note: "*" implies a normal Prolog backtrack

FIGURE 18-20
Derivation of normal Backtrack Table entries.

Selection of such a backtrack literal works fine when the failure being handled is the first failure in a clause, but the process needs to be modified when there is a chain reaction of failures. In Figure 18-18(a), consider what happens if goal 8 fails. Backtrack to goal 6 occurs. However, if 6 also fails, Figure 18-20 goes all the way back to literal 1. This is improper. Goal 8's failure requires a new value for one of the two variables it consumes, namely, either d or e. Goal 6 generates e and consumes b. If we go only by this, we will miss the original cause of the failure, which is a need for a different d than 3, and go back to 1.

To fix this, we need to keep in the choice point a flag which indicates if execution is proceeding normally from left to right, or if it is trying to recover from a prior failure. If the latter, a different table from Figure 18-20 is needed. In this table we need to keep track of all variables consumed by a literal *and all variables consumed by literals which may backtrack into this one*. Figure 18-21 diagrams such a table derivation.

Obviously, modifications to the WAM are needed to support such operations. The set suggested by Chang and Despain (1985) is pictured in Figure 18-22. In use, the new ***i-allocate*** is used in place of an allocate to build on the heap space for the choice point table. Instead of a call **qi** instruction, the compiler would generate:

```
make i
i-call qi
enter i
```

This code would place on the stack a pseudo-choice point that identifies the number in the clause of the literal corresponding to the predicate being called. It then calls the appropriate procedure, saving the current B

Literal	Variables Consumed	Generating Literals	Backtrack Table Entry
1	–	–	*
2	a,b	1	1
3	a,c	1,2	1
4	–	–	*
5	–	–	*
6	b,c,d	1,2,3	3
7	–	–	*
8	–	–	*

Note: "*" implies a normal Prolog backtrack

FIGURE 18-21
Derivation of cascaded Backtrack Table entries.

Instruction	Operation
i-locate n,m,L	Allocate (on heap) space for m entry choice point table. Create environment on stack for m permanent variables, with pointers to new choice point table, and backtrack table at L.
make m	Push a small pseudo—choice point on the stack, containing B, E, and m (m is the literal number of next call).
i-call L	Call instruction which also saves B in current environment.
enter m	Used after i-call to save B in choice point table indicated by environment.

FIGURE 18-22
WAM modifications for semiintelligent backtracking.

register where the backtrack operation can find it. Finally, upon a successful return from the called routine, the enter saves the current value of B in the appropriate choice-point table entry so that a later failure can find it quickly.

Statistics in the referenced paper indicate that this technique varies from having little or no effect (but at little real time cost) to being extraordinarily effective for some problems.

18.6 PROBLEMS

1. Identify the temporary, permanent, void, and unsafe variables in each of the following:
   a. **p**(x,y,z):−**q**(x),**r**(y).
   b. **p**(x,y,z):−**q**(x,y),**r**(y,z),**s**(x).

   c. **p**(x,y,z):−**q**(x,y),**r**(x,z,w),**s**(x,v).
   d. **p**(x,y,z):−**q**(x,y),**r**(y,z,w),**s**(x,z).
2. Convert the set of PROLOG statements for each of the following predicate symbols as found in the text into an optimized WAM program:
   a. **member**(x,y) (use the variable z in place of the anonymous variable)
   b. **reverse**(x,y)
3. Develop a complete optimized WAM program for the PROLOG program of Figure 16-22. Add the query ?−**factorial(s(s(s(0))),**y) to the code.
4. Convert Figure 16-24 into optimized WAM code. Do not generate code for **clause** or **system** (assume that they exist in a library somewhere). Do not do environment trimming, but do use temporary variables, register optimization, and call optimization as possible.
5. How would you modify the full WAM architecture to handle cases where predicate symbols used as clause heads have more arguments than the particular WAM implementation being used has argument registers?
6. Define the register transfers which would go on for read-mode and write-mode versions of unify, get-list, and get-structure instructions.
7. For Figure 18-18(a), show the choice points on the stack as the query **map**(a,b,c,e,f) is executed.
8. Develop the backtrack tables for Figure 18-18(b). Show the choice points on the stack as the query **map(Red,**b**,Green,**e**,Blue)** is executed.
9. Compute the data dependency graph for **map** as defined in Figure 18-18 when the input annotation is (−,−,−,++,−). Then develop the backtrack tables.
10. Write a PROLOG procedure **unify**($x,y,ein,eout$), where x and y are s-expressions and ein is of the form ((⟨var⟩.⟨s-expression⟩)*). This procedure will return successfully if x and y unify successfully, with $eout$ yielding the unifying substitution in the same format as ein. The procedure fails if the unification is not possible.

    Compile this into optimized WAM code, assuming that the built-in **var**(x) succeeds if x is currently an unbound variable, and fails otherwise, and that the built-in **number**(x) does the same if its argument is a number.

CHAPTER 19

PROLOG IMPLEMENTATIONS

The PROLOG inference engine has been implemented in practically every way possible: by interpreters and compilers for conventional machines, by microcoding techniques, and by specialized machines. This chapter reviews briefly some of the more interesting implementations for essentially sequential machines, with emphasis on ones that have used novel computer hardware architecture techniques. These include:

- A brief description of how the WAM can be mapped onto conventional computers
- An overview of the original design proposed by Tick and Warren to support the WAM directly in hardware
- A specialized personal computer with a built-in microcoded PROLOG interpreter (the PSI machine)
- Several different processors that execute variants of the WAM architecture directly (the HPM, PLM, and Xenologic machines)
- A reduced-instruction-set machine matched to the WAM architecture
- Some methods of using associative memory for implementimg PROLOG

While interesting, simply describing these machines does not necessarily give insight into where or how their implementations excel. As with the LISP machines of Chapter 10, benchmarks, performance standards, or yardsticks are needed. The next section briefly introduces such topics.

Finally, the systems described in this chapter were chosen for either historical or pedagogical purposes. There are quite a few other systems to which the interested reader might refer, including:

- A specialized processor just for unification (Woo, 1985)
- The WAM architecture implemented in microcode on a conventional computer, a DEC VAX 8600 (Gee et al., 1987)
- Use of the WAM as an intermediate step to compiling code for a Motorola MC68020 microprocessor (Mulder and Tick, 1987)
- The ***Knowledge Crunching Machine*** (Benker et al., 1989)—a microgrammed CISC-like machine with 64-bit tagged memory and a variety of special addressing modes and cache mechanisms
- The ***Integrated PROLOG Processor*** (Kurosawa et al., 1988), which represents an extension to the WAM that increases its performance on highly pipelined CPUs.

19.1 MEASURING PROLOG PERFORMANCE

A common question that one might ask about a computing system is "How fast is it?." Ultimately such speed can be related only to the time required to solve specific problems. However, there is often a need for a handy "rule-of-thumb" number that one can use for quick comparisons. For conventional computers there are several such measures, ranging from a high-level "average number of key operations per second" to a low-level "average number of instructions executed per second." In the former, an ***operation*** is some relatively high-level function that is closely related to typical applications of interest. A floating-point operation, a compare, and a basic file transaction are all examples. For example, the ***Livermore Loops*** and the ***Linpack Loops*** represent collections of small numeric loops which, when timed on conventional computers, return measures of "megaflops per second," where a ***flop*** is a floating-point numeric operation.

In contrast, an ***instruction measure*** is tied much more closely to the internal architecture and detailed timing of a machine. An operation here consists of basic machine instructions such as load, add, store, compare, branch, etc. A machine's "instructions per second" rate is computed by computing the average time taken by a computer to execute some ***mix*** of instructions (e.g., 20 percent of the instructions encountered during execution of an average program are load, 10 percent are branch taken, etc.). Finding this mix often involves detailed analyses of real programs and how often they are actually run in real applications. An example of a classical mix is the ***Gibson mix***. (See Gibson, 1970; for other mixes also see Shustek, 1978, and Hennessy and Pattersen, 1989, Chap. 4.)

Numerically, conventional microprocessors as found in typical personal computers run on the order of hundreds of kips (1 ***kip***=1000 instructions executed per second), while large commercial computers are rated in the tens of mips (1 ***mip***=1 million instructions per second).

The following subsections attempt to describe similar measures for

PROLOG computers. We will start with a relatively high-level measure of performance, "logical inferences per second," discuss some of the benchmarks in use today for such measures, follow this with some low-level mixes for WAMs, and then discuss ways of factoring out technology variations from different implementations.

The reader may wonder why we did not do this earlier for functional systems. The best answer is that there is no single event or operation within a LISP program, for example, that occurs with enough consistency over enough programs to qualify. In addition, there are several radically different models of execution (compare the SECD Machine to the G-Machine, for example) that have almost nothing in common at the machine level.

19.1.1 Logical Inferences per Second

A flop is a standard high-level measure for conventional computers, but it is clearly inappropriate for logic-based systems. Instead, a more appropriate operation seems to be an ***inference,*** namely, the successful use of a clause for a particular goal. When used as a measure of performance for logic-based languages and machines, one thus counts the number of inference rules attempted (and successfully applied) per second. This is ***logical inferences per second,*** abbreviated ***lips.*** Two variations are possible:

- A count of the number of successful unifications of goals with clause heads
- A count of the number of clauses in which all the right-hand literals succeed

For a variety of reasons we will tend to use the former.

For reference, performance numbers quoted for current logic-based computing systems range from hundreds of lips (for software interpreters running in conventional computers) to perhaps a few hundreds of thousands of lips (for specialized machines).

Such numbers must be taken much more lightly than mips, however. As yet there is no mix that is truly representative of anything larger than toy problems. Further, there are complicating factors that have no analog to conventional measures, including:

- The ratio of inference rules tried that fail to those that succeed, and how those failures are counted
- The complexity of the substitutions derived in the process of trying an inference rule
- The overhead of the decision procedure that selects clauses to try
- The number of inferences that initially succeeded but, because of later failures, contribute nothing to the final solution

- The handling of the typically large amount of storage needed to record the inference sequence as it is developed

As an example of the fourth point, consider a "bad" decision procedure, which spends a great deal of time on "wild goose chases" but makes very simple inferences, versus a "good" decision procedure, which solves the problem with a few relatively complex inferences. The former may yield a very high lip rate and the latter a very low one. To see this, refer back to Figure 18-17(*a*) (repeated as Figure 19-1). A naive implementation could involve 30,000 inferences, each of which executes quite quickly, yielding a very high lip rate. On the other hand, an implementation using really intelligent backtracking might require only four inferences, each of which is longer than above, and thus yielding a deceptively low lip rate. Thus the numbers quoted here, and in the literature, should be taken with a great deal of skepticism.

19.1.2 Benchmarks

Just as with LISP in Chapter 10, the PROLOG community has developed a wide variety of benchmark programs to use as comparisons. Figure 19-2 lists a few of these. The code for many of them can be found elsewhere in this text, or through the cited references. Also, depending on how one counts comments and overall test driver code, the actual number of clauses for each benchmark may vary somewhat around what is listed here.

Unlike the LISP benchmarks, however, there is no consensus as to what feature of PROLOG each program exercises. The only one of any universal reference is the simple two-clause **append** (sometimes called ***concatenate***). This program is used in one of two ways. As a ***determinate append,*** the first two arguments are bound and the third argument is unbound at the beginning. There is exactly one solution. As a ***nondeterminate append,*** the roles are reversed, and the number of solutions equals the length of the list bound to the third argument.

```
p(1).
  ...
p(10000).
r(0,1).
r(1,2).
r(2,3).
?-p(x),r(y,y).
```

Naive Implementation: 30,000 fast inferences

Smart Implementation: 4 slower ones

FIGURE 19-1
What's in an inference.

Program	# Clauses	Source	Purpose
Append	2		Determinate/Nondeterminate
Hanoi	3		Towers of Hanoi puzzle
Tak	3	Touati(1987)	Translated Gabriel Benchmark
Naive Reverse	4		Reverse a list using append
Quicksort	5	Warren(1977)	Sort list of numbers
Serialize	12	Warren(1977)	Structure sort of a list
Diff	14	Warren(1977)	Symbolic Differentiator
Mutest	17	Dobry(1985)	Proof of "muiiu" theorem
Pri2	20	Dobry(1985)	Primes <100 using Sieve
nqueens	21	Despain	Place n queens on chess board
Browse	26	Dobry(1987)	Translated Gabriel Benchmark
Query	52	Warren(1977)	Simple Database query
Ckt2	102	Dobry(1985)	Logic circuit designer
Puzzle	129	Touati(1987)	Translated Gabriel Benchmark
Reducer	348	Van Roy(1984)	Combinator Reducer
CHAT	500	Warren(1981)	Nat.Lang.Database Query
TLI	576	Haridi	Intuitinistic Logic Interpreter
SDDA		Carlton	Data Dependency Analyzer

FIGURE 19-2
Some more complex benchmarks.

19.1.3 WAM Mixes

Tracing a program involves monitoring and recording the activity of the underlying machine while it is executing the program. Given such a trace, one can compute all sorts of performance measures. Perhaps the most common such measure is a histogram of the number of occurrences of each instruction opcode. This is called a ***mix.***

No single mix is a perfect characterization. It clearly depends on the target machine's architecture, but it also depends on the benchmark program being traced, and on the compiler used to generate the code.

With these caveats in mind, Figure 19-3 lists one of the few published mixes for the WAM instruction set (see Dobry et al., 1985). It assumes the WAM variant accepted by the PLM machine described later in Section 19.5.1, and primarily hand-compilation of several standard benchmarks. The individual mixes from each of these programs were averaged together to get the numbers here.

At the time this mix was published in 1985, there was only sketchy data available about details below the opcode level, such as how many get-vars were to permanent or temporary variables, how often trailing occurred, or how many kinds of data references of different types happened. Since then a variety of work, including Ph.D. theses by Dobry (1987a) and Tick (1987) and measurements on the IPP (Morioka et al., 1989), have generated quite a bit more detail. Touati and Despain (1987), in particular, give statistics of some of the missing WAM parameters, and Tick (1988) lists a summary of the memory traffic for a variety of PROLOG programs and how various forms of caches and stack buffers minimize memory traffic. Figure 19-4 gives a brief summary of some of these statistics.

Class	Instruction		Mix%	Class	Instruction		Mix%
Unify		29.2		Procedural		15.0	
	unify-var		8.8		execute		4.0
	unify-cdr		6.9		allocate		3.5
	unify-val		5.0		deallocate		2.9
	unify-nil		4.9		proceed		2.6
	unify-cons		3.3		call		2.0
	other		0.3	Index		11.7	
Get		19.4			swx-on-term		4.9
	get-list		7.3		try-me-else		2.5
	get-str		4.1		trust-me-else		1.3
	get-var		3.4		retry-me-else		0.9
	get-cons		1.8		swx-on-str		0.9
	get-val		1.4		try		0.7
	get-nil		1.3		trust		0.4
Put		17.3			swx-on-cons		0.2
	put-val		10.7		retry		0.1
	put-cons		2.7	Other		7.3	
	put-var		1.8		escape		4.9
	put-unsafe-var		1.2		cut		1.9
	put-list		0.8		fail		0.5
	put-str		0.1				

Note: Adapted from Dobry, Despain, and Patt (1985).

FIGURE 19-3
A WAM mix.

19.1.4 A Common Example

Given the prior discussions, it is obviously difficult, and usually misleading, to try to compare via a simple performance measure the various machines to be discussed in the following sections. However, to permit at least some relative comparison of the value of various techniques used in the machines discussed below, we will try to give relative performance numbers where applicable for some small set of operations, primarily the determinate **append** program discussed earlier. In this case we will very explicitly define a ***logical inference*** as being equivalent to the execution of a call or execute instruction in a WAM program form, with complete disregard for how many rules are tried or whether or not backtracking occurred. ***Lips*** is thus a count of how many of these are executed per second for some specific program.

While informative, such a lips measure does gloss over differences in hardware technology. We would like to be able to distinguish between a machine that is fast because of some architectural feature and one that is fast simply because of raw device speed. Thus, for machines with close

Number of Dereference Loop Iterations	Percent of All Variables at Start of Unification
0	63–74%
1	25–36%
2	0–0.2%

Parameter	Average
Unbound Variables At Unification	19–26%
Choice Point References/All References	52%
—% avoidable via optimization	0–100% (approx 50% ave)
% of Reuseable Choice Points (eg. Tail Recursion)	often small
% Memory Accesses Saved by HB Register (sometimes had a negative effect)	14–35%
% Memory Accesses Saved by Cdr-coding	3%

(*a*) WAM parameters from Touati and Despain (1987).

Area of Memory	% of all References	Reads to Writes
Choice Point	54	1 to 1
Environment	23	1 to 2
Heap	18	2 to 1
Trail	3	1 to 1
PDL	2	1 to 1

(*b*) Data memory referencing statistics from Tick (1988).

FIGURE 19-4
WAM statistics.

instruction sets, a somewhat more reasonable measure for performance comparisons is machine ***cycles per inference,*** or ***cpi.*** This is roughly the reciprocal of the product of the machine's basic cycle time and the lip rate. A lower cpi reflects a machine organization which performs approximately the same amount of work in a fewer number of basic steps. This should correspond to a more efficient architecture.

With these disclaimers in mind, Figure 19-5 lists some lip and cpi rates for several of the machines discussed here. The benchmarks used are ***determinate append*** (appending two known lists), ***quicksort,*** and ***towers of Hanoi.*** For other benchmarks, the literature [cf. Dobry et al. (1985) in particular] seems to indicate that a lip rate of one-half to one-third of the determinate **append** numbers is typical.

19.2 COMPILING TO A CONVENTIONAL COMPUTER

(Newton, 1987; Bowen et al., 1986)

There are several ways of mapping the output of a WAM-based PROLOG compiler to a conventional machine:

Machine	Determinate Append	Hanoi	Quicksort	Source
PSI	30/166		55/90.2	Nakajima (1986)
VAX 8600	108/116	116/108	107/116	Gee (1987)
S/370	870/48			Newton (1987)
HPM	280/35.7			Nakazaki (1985)
IPP	1360/32		573/76	Kurosawa (1988)
PLM	346/28.9	314/31.8	153/65.4	Dobry (1985)
Tick	450/22			Tick and Warren (1984)
PSI-II	400/15		188/36	Nakashima (1987)
KCM	857/15	639/20	465/27	Benker, et al. (1989)
LOW RISC	1200/16			Modified from Mills (1989)

Performance numbers x/y reflect x thousand inferences per second on real or proposed machine, or an equivalent y machine cycles per inference.

FIGURE 19-5
Some relative performance comparisons.

1. Write an interpreter for the WAM variant produced by the compiler, and then execute a WAM program by presenting it as data to the interpreter.
2. Expand each of the WAM instructions generated by the compiler into native instructions for the target machine, and execute the resulting program.
3. If the target machine is microcoded, write a WAM interpreter in the microcode of the machine, and proceed as in the first alternative.

All three approaches have been used. Many variations of the first have been done (often as student projects) by encoding WAM instructions in byte-sized elements and writing a ***byte-code interpreter*** to fetch the program bytes one by one, perform a multiway branch on the selected byte, and execute the subroutine at the branch address. There is one subroutine for each WAM opcode. Although it is obviously slow, such a system has the advantage of being easy to instrument for performance measures. The mix described by Dobry et al. (1985) and discussed earlier was developed with just such a system.

The final approach, developing an interpreter in microcode, is often the fastest of the lot but can be useful only when the microarchitecture of the machine is available to the programmer. This is certainly not true for most of the machines on the market today, such as the vast majority of microprocessors used in personal computers and workstations.

The middle approach, expanding the WAM instructions into the native architecture of the target machine, is the most versatile and most popular. It is the subject of this section.

There are two approaches to expanding the WAM code. First is a simple ***macroexpander,*** which replaces each WAM instruction by a standardized set of assembly code which does the same function. This is often called ***open coding*** the WAM instructions. Very often this can be done by generating WAM code as a text file in an assembly-language format, and letting a macro package written for the machine's native assembler expand these WAM instructions.

While this approach is simple, it does not permit really tight optimization of the final code. Consequently, what is usually done is to have a second pass of the PROLOG compiler take an intermediate WAM program and build specialized code sequences to match. This second pass understands WAM code at a more global level than a macro package can, and it can produce optimized target code for whole strings of WAM instructions.

The following subsections describe the design decisions that framed one such compiler (developed at the California Institute of Technology—Cal Tech), which generates code for the IBM System/370 ISA, and which on an IBM 3090 mainframe attained speeds of over 870 klips for the **append** benchmark (Newton, 1987). We are interested here in those decisions that were made as a result of the target System/370 architecture, and not inner details of the WAM or compiler itself.

The compiler was written totally in PROLOG (approximately 8000 lines of code), produced WAM code in an intermediate pass, and then took these WAM sequences and produced optimized System/370 assembly language that was then assembled by the standard System/370 assembler and linked to a runtime library of about 4000 assembly statements. The compiler was capable of compiling itself.

Another example of such optimizations will be found later in this chapter in discussions on the LOW RISC architecture.

19.2.1 Mapping Memory

Mapping a tagged architecture onto any machine that does not explicitly support tags must be done carefully. In the case of the ISA assumed for the Cal Tech compiler, addresses are 24 bits long and address down to the byte. When stored in a 32-bit word, the address is right-aligned and the upper 8 bits are not used. Further, when loaded into a 32-bit register and used as a base or index, the machine ignores the upper 8 bits, making them an ideal place for tags.

Note that in the more modern IBM XA ISA, addresses are 31 bits long, meaning that there is at most one tag bit available, and any others must be stored elsewhere.

Figure 19-6 diagrams how the tag bits were laid out. The choices were made very deliberately, with consideration of how fast they can be tested by actual code. The leftmost bit corresponds to the sign bit in standard integers, meaning that it can be easily tested by any sequence that tests for positive or negative numbers.

Due to the simplicity of this test, this lead bit is used for the most frequently needed test, namely, between objects which are pointers (references or unbound variables) and all other objects. In the former case, the difference between a reference and an unbound variable can be determined by the address. If it matches the address of its own memory word it is an unbound variable; otherwise it is a reference.

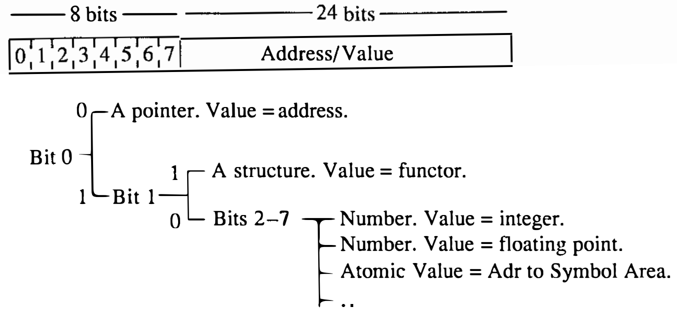

8 bits
24 bits
0 1 2 3 4 5 6 7
Address/Value
Bit 0
0
A pointer. Value = address.
1
Bit 1
1
A structure. Value = functor.
0
Bits 2–7
Number. Value = integer.
Number. Value = floating point.
Atomic Value = Adr to Symbol Area.
..

FIGURE 19-6
Basic cell.

Bit 1 in the tag byte distinguishes between structures and simple atoms. In the former, the value field points to a table in low memory that contains the character string name of the functor. A byte in each table entry indicates the arity of the functor. Words following this word designate the function's arguments.

If this tag bit is 0, the rest of the tag bits distinguish different kinds of basic atoms.

Note that there is no tag for lists. In this implementation, a list is encoded as a structure of arity 2 and functor name "dot."

Again, testing of this second bit can be done easily by a standard BXLE instruction (branch index less than or equal) set up to compare a register to itself.

The actual allocation of memory to WAM data structures is similar to that discussed earlier, but with a few differences:

- Choice points and environments are stored in separate stacks, with separate top-of-stack pointers.
- There is a separate data space for asserts and retracts.
- There is a separate global data area holding standard constants and symbol character string names.
- All stacks grow down rather than up.

This last point is a reflection of the ISA, which permits only positive displacements off a base register. Thus, given a register which points to the top of a stack, we can address into the stack only if the current entries are higher in address than the base.

Finally, stack overflow is handled through the machine's virtual memory system. A special page is placed at the end of each memory area allocated to a stack. This page is marked with a different storage protection key so that any access to it will cause an interrupt. No machine cycles need to be spent in normal processing to test for it.

19.2.2 Register Assignment

The architecture has 16 programmer-addressable fixed-point registers. This is sufficient to hold either a few of the argument registers or the prime WAM registers, but not both. Further, provisions must be made for predicates with an arbitrarily large number of arguments. The approach taken is to store all arguments in the global memory area and use the basic hardware registers to hold all the important WAM registers. Figure 19-7 diagrams these assignments. The registers R1 through R3 are working registers which are loaded with the appropriate argument or temporary registers as needed.

19.2.3 Optimized WAM

When presented with a program as input, the Cal Tech compiler goes through several passes which generate WAM code, register optimized WAM code, condense optimized code, and then assembly output. The first two of these intermediate outputs match fairly closely what we discussed in the prior two chapters. The next section gives an example of the final pass's output.

The second-to-last pass is where all the final optimizations are performed. In this pass, sequences of WAM instructions are analyzed and combined into a form that permits the last pass to produce particularly

Register	Mnemonic	Usage
R0		Unused
R1		Work register—current argument
R2		Work register
R3		Work register
R4	LC	"Low Core" Pointer to special areas
R5	BASE	BASE addressing for S/370 procedures
R6	S	Structure Pointer
R7	H	Heap top pointer
R8	B	Backtrack Choice Point stack top
R9	HB	Heap Backtrack register
R10	TR	Trail stack top
R11	URET	Return address for unify subroutine
R12	CP	Continuation Pointer register
R13	E	Top of Environment Stack
R14	CE	Current Environment
R15		Unused

FIGURE 19-7
Register assignments.

tight code. Figure 19-8 gives a deliberately simplified example for the **append** program when mode information of the form (+,+,?) (determinate mode) is available. There are several classes of commands here. First are more or less conventional WAM instructions, such as get-nil. Next are directives that indicate how certain functions, such as trailing, are to be invoked. In this case trailing is turned off of all variables other than those associated with A2.

Finally there are composite instructions which represent sequences of WAM instructions which should be optimized together. For example, the statement:

```
get-list SP,unifyvar(A4),unifyvar(A1)],L5
```

indicates that the argument to be analyzed for a list has been dereferenced by a previous instruction, and a pointer to the functor has been left in the SP. Further, the two components are to be unified for a variable into A4 and a variable with A1.

The other get-list is similar, except that here any successful unification should be terminated by a branch to a statement labeled **append.** This prevents wasteful double branches.

19.2.4 Final Assembly Code

Figure 19-9 diagrams some assembler code put out by this compiler for the initial switch and the second clause of the **append** benchmark. Figure 19-10 diagrams code for the first clause. Not all of the code is shown; there are actually 113 instructions and quite a few assembler directives and comments in the real code, not counting runtime standard code such as fail and unify. Also, the notation used here is different from that of Newton's report. It was chosen to be a little easier to read, and to agree with prior notation.

Also included is a number in front of each instruction executed during one inference call where A1 and A2 are both bound to lists and A3 is

```
          trail-it (A2)      ;Assembler directive to trail A2
          trailing (off)     ;Assembler directive to turn off trailing.
; Following is code for the second clause.
append:   switch-on-term  L4,RL5 ;*+1 if A1 a var, L4 if str, RL5 atom.
fastL5:   get-list SP,[unifyvar(A4),unifyvar(A1)],L5
          get-list A3,[unifyval(A4),unifyvar(A3),execute(append)]
; Following is code for the first clause.
L4:       get-nil A1
          get-val&proceed A2,A3
          trailing (on)      ;Assembler directive to turn on trailing.
```

FIGURE 19-8
Final optimized WAM code.

```
append:1 BALR  BASE,0       ;Set up R5 for program addressing
; ——— switch-on-term L4,RL5
       2 L     R1,ARGS      ;Get A1 into R1
H1:    3 LTR   S,R1         ;Set CC if R1 a pointer
       4 BNM   H2           ;Branch if R1 a pointer
       5 BXLE  R1,R1,RL5    ;Branch to RL5 if R1 a structure
         B     L4           ;Branch to first clause if R1 atom
;        If here, R1 a pointer—dereference it.
H2:      C     R1,0(R1)     ;See if pointer to self
         BE    H0           ;If so, R1 an unbound var
         L     R1,0(R1)     ;No, load R1 with its contents
;        Following is inline code for 1 level dereference
         LTR   S,R1         ;Set CC if R1 a pointer
         BNM   H2           ;Branch if R1 a pointer
         BXLE  R1,R1,RL5    ;Branch to RL5 if R1 a structure
         B     L4           ;Branch to first clause if R1 atom
         ....
; ——— get-list SPh,[unifyvar(A4),unifyvar(A1)],L5 in read mode
RL5:   6 CLC   0(4,S),CDOTPTR   ;See if function=dot
       7 BNE   FAIL             ;Fail if not
       8 SL    S,*cons1*        ;Subtracting *cons1* adjusts S as base
       9 MVC   ARGS+12,8(S)     ;Put cdr (at 8(S)) into A4 (at ARGS+3×4)
      10 MVC   ARGS,4(S)        ;Put cdr (at 4(S)) into A1 (at ARGS+0×4)
; ——— get-list A3,[unifyval(A4),unifyvar(A3),execute(L1)]
      11 ICM   S,X'F',ARGS+8    ;Load S with A3 and test if pointer
H12:  12 BM    H9               ;Branch if S not a pointer
      13 C     S,0(S)           ;It is—see if unbound
      14 BE    H13              ;Branch if so
         ... code dereferences S if not unbound var ..loop to H12
;        At this point, S (A3) known to be var—write mode
H13:  15 O     H,STTTAG         ;Set tag of H to structure
      16 ST    H,0(S)           ;Bind structure pointer to var at S
      17 MVC   0(4,H),CDOTPTR   ;Set functor cell to "dot"
      18 SL    H,STTTAG4        ;Bump H and clear its tag
      19 CR    S,HB             ;See if var at S need be trailed
      20 BL    H15              ;Branch if no trail
         ST    S,0(TR)          ;Push S to trail
         S     TR,4             ;Adjust trail top
; ——— unifyval(A4) .. Now handle car—write mode
      21 ICM   R1,X'F',ARGS+8   ;Load R1 with A4 and test if pointer
H17:  22 BM    H18              ;Branch if A4 not a pointer
         C     R1,0(R1)         ;It is—see if unbound
         BE    H18              ;Branch if variable
         ... As after H2 .. go indirect and loop to H17
H18:  23 ST    R1,0(H)          ;Push R1 as car to heap
      24 S     H,FOUR           ;And adjust H by subtracting 4
; ——— unifyvar(A3), execute ...—again write mode
      25 ST    H,ARGS+12        ;Save pointer to heap in A3
      26 ST    H,0(H)           ;Push unbound var to heap
      27 S     H,FOUR           ;Adjust H
      28 BR    BASE             ;execute append
;        .. code for read mode on argument
```

FIGURE 19-9
Partial **append** code output.

```
; ——— trust-me-else fail
L6:  LM    B,HB,B$B(B)     ;Delete topmost choice point
     ST    B,CUT           ;Save in cut register
;    Now start code for first clause
L4:  BALR  BASE,0          ;Establish addressability
     ICM   R1,X'F',ARGS    ;Get A1 into R1
H21: BM    H19             ;Branch if not a pointer
     C     R1,0(R1)        ;A pointer—see if unbound var
     BE    H22             ;Branch if unbound
     ICM   R1,X'F',0(R1)   ;No—must load word pointed to
     BM    H19             ;Again branch if not a pointer
     C     R1,0(R1)        ;A pointer—see if unbound var
     BE    H22             ;Branch if unbound
     B     H21             ;Loop backwards if >1 level reference
;    At following, A1 an unbound variable. Bind it to nil
H22: MVC   0(4,R1),NILPTR  ;Load var with Nil constant
     CR    R1,HB           ;See if trailing necessary
     BL    H20             ;Branch if not
H23: ST    R1,0(TR)        ;Trail it
     S     TR,FOUR
     B     H20
;    At following we know A1 =R1 an atom. see if nil
H19: C     R1,NILPTR       ;Compare to nil
     BNE   FAIL            ;Branch if not
; ——— getval-proceed(A2,A3)
H20: LM    R1,R2,ARGS+4    ;Load R1 and R2 with A1 and A2
     LR    URET,CP         ;Save CP for unify
     B     UNIFY           ;If successful, go to CP
```

FIGURE 19-10
Code for the first clause.

bound to a variable in a part of the heap where it does not need be trailed. As can be seen, there are 28 such instructions executed.

The first instruction merely loads BASE (R5) with the address of the start of the **append** code. This will be used throughout as a base register for program branching and at the end to complete the tail-recursive loop. It is standard for a System/370 program to start like this.

The next four instructions load a working register R1 with argument A1, and perform the dereferencing and tag checking required of the switch-on-term. In this case A1 is a list, so the code branches to RL5. Notice that at RL5, the code can assume that both R1 and S (R6) contain dereferenced copies of A1. Note that the code following the BXLE labeled 5 handles the case where R1 is an atom (and clause 1 at L4 can be tried), or where R1 is a pointer that is either an unbound variable (branch to H0 to try-me-else) or a reference. In the latter case one iteration of dereferencing code is included for those many cases that require just one loop. This ***loop unrolling*** saves a performance-destroying branch.

Instructions 6 and 7 test if the four bytes at S are a structure with a functor of name "dot" and arity 2. If not, unification fails, as signaled by the branch to FAIL. The next three instructions then correspond to the two unify-vars by copying the car and cdr into A4 and A1, respectively.

Instructions 11 through 14 perform the tag processing of the get-list A3. In this case A3 is assumed to be an unbound variable, so the code that is not shown beyond 14 is not executed.

At H13 we start the code for the write-mode form of the get-list. A structure header consisting of a functor with name "dot" and arity 2 is pushed to the heap, and the variable at S (A3) is bound to a pointer to this new heap entry. The heap is below the HB register, so no trailing is necessary.

The handling of the unify-val A4 when we know that we are in write mode is quite simple. We simply push a copy of A4 onto the heap just below the functor header. This is done in instructions 21 through 24.

The code for the unify-var A3 in this case is equally optimized for write mode. An unbound variable is pushed as the cdr, and a reference to it is placed in A3.

Finally, the execute really becomes a simple branch back to the start, just as discussed in the last chapter.

Figure 19-10 diagrams the code for the first clause. All code is shown here. Much is the same as before in structure. The primary difference is in the call to the **unify** subroutine, with a return address equaling the CP at entry. A successful return from **unify** will be a shortcircuit back to the **append**'s caller.

19.3 THE ORIGINAL TICK AND WARREN MACHINE

(Tick, 1983; Tick and Warren, 1984)

The first attempt to lay out a hardware architecture of a processor that would execute the WAM architecture directly ended up with a surprisingly conventional feel to it, but with suggested performance of perhaps an order of magnitude better than what was possible at the time with compilers and conventional computers. Although the machine itself was never built, its influence can be seen in practically every PROLOG machine since then.

The key characteristics of the Tick and Warren design included:

- A tagged memory and tag-checking hardware in the CPU
- Hardware registers corresponding to the main WAM registers
- Specialized cachelike support for the top entries in the stack, trail, and PDL
- A pipelined dataflow optimized to the WAM operations

It is interesting to note that the same features optimized here, namely, tagged memory, tag compare logic in the CPU, and direct support and caching for stacks, are those that were used in the specialized LISP machines of Chapter 10. In fact, a machine optimized for one of the two languages tends to be a decent target for the other.

System issues such as I/O, memory management, multiprogramming, etc., were not considered in this design. The emphasis was on what performance limits might be possible, not on a fully developed and general-purpose machine.

Figure 19-11 outlines the machine design. The ***stack buffer*** and ***trail buffer*** are small memories that have the top few entries of the stack and trail in them for immediate access to the dataflow. The ***PDL*** is a memory that supports internal unification stack needed by the UNIFY-xxx instructions when nested terms are encountered. The ***register file*** is also a small memory containing the current argument registers and some temporary variables. Again, any one of these entries is available to the main dataflow in one machine cycle. The other registers in the WAM model are mapped directly into separate registers also close to the dataflow.

The dataflow is itself ***pipelined*** (see Kogge, 1981), with ***staging latches*** (T, T1, and R) surrounding the main arithmetic/logic unit. This permits three simultaneous activities to be going on in each machine cycle:

- Copying values from the registers and buffers into T and T1
- Performing an operation on the contents of T and T1, with the result destined for R
- Copying the contents of R back into the registers and buffers

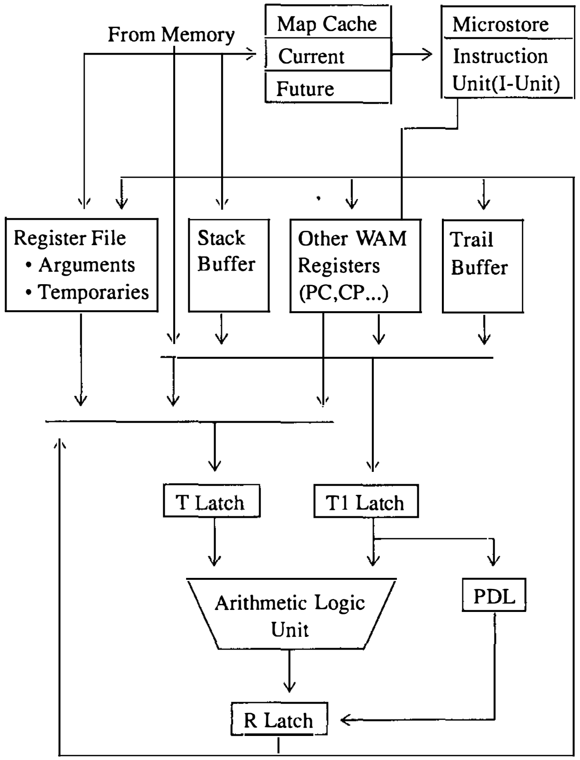


FIGURE 19-11
The Tick-Warren Machine design.

The pipelined design means that these changes to the latches occur only at the end of one machine cycle, and their values are stable in between. Also, the register file permits a simultaneous read of one register into T or T1 and a write to another from R in the same cycle.

Instructions are fetched and decoded sequentially by the ***I-unit,*** which also has several specialized memories surrounding it. First are two instruction buffers, where instructions are prefetched from memory before they are needed. One of the buffers is full of instructions which follow sequentially the one currently in execution (the ***current instruction buffer***). The load of the other (the ***future instruction buffer***) is triggered by a ***prefetch*** instruction, which the compiler inserts into the code when it knows that there is an execute or call instruction coming up. This permits an execute, in particular, to be replaced by a simple instruction which indicates that the future buffer is now the current one.

If both buffers can be kept full, then the I-unit never needs to wait for a memory read when it finishes one instruction and is ready for the next, even after a call or execute which changes the PC (and switches the relative roles of the two buffers).

An additional source of prefetched instructions is the ***map cache,*** which contains previously seen switch-on instructions. Thus when a call or execute instruction changes the PC, if the new value is in the map cache, then a fetch of the switch instruction itself is not needed. Further, the current tag for A1 is tied directly into accessing of the cache, permitting the appropriate subchain to be signaled directly, without actually executing the switch.

Once the I-unit gets an instruction, it uses the opcode to start the execution of a microprogram which then · performs the appropriate dataflow and other control operations. Each microinstruction has fields controlling each of the three activities described above, plus control for the next microinstruction to be executed. Some of the microbranch modes use tag bits from various argument and other registers to perform multiway branches. This permits rapid unification execution by direct branching to the microcode which handles the particular combination of objects being processed.

As with many high-performance machines (again, see Kogge, 1981), all these instruction prefetches, decodes, and executions (in some cases) occurs while the dataflow is busy with the previous instruction. Ideally, the net result is that performance is limited solely by the rate at which instructions which require dataflow operations can be processed.

19.4 FIFTH-GENERATION PROLOG MACHINES

The Japanese ***Fifth-Generation Computer Project*** has emphasized the development of high-performance machines that execute logic programs directly. Two of these described below are the PSI Machine and the HPM Machine. The former is an example of a machine designed around a PROLOG interpreter; the latter is one of the first WAM-based machines actually built. Both are essentially sequential machines; other ICOT (Institute for New Generation Computer Technology) machines built for parallel-logic languages are discussed later. Also not discussed here are several variants of these two machines, such as the ***PSI-II,*** a WAM-oriented design based on special Large-Scale Integration (LSI) chips (Nakashima and Nakajima, 1987).

The actual language implemented is a variant of PROLOG called ***KL0*** (for kernel language 0), which is very close to a subset of DEC-10 PROLOG as implemented on a DEC 2060 computer. This language was designed as a base language capable of supporting operating systems, interpreters, and compilers for more powerful logic languages called KL1 and KL2.

19.4.1 PSI—The Personal Sequential Inference Machine

(Nakajima et al., 1985; Taki et al., 1987)

One of the first PROLOG machines to be manufactured in significant numbers was the ***Personal Sequential Inference Machine*** (or ***PSI***) designed and built by ICOT. This machine was designed as a personal workstation which could run independently or be attached to a larger host. Its main characteristic is that it supports an interpreter for KL0 totally in microcode. Further, this interpreter supports the entire operating system, which includes bit-mapped graphics and a window-based program development environment. The overall performance of the interpreter is roughly on a par with that of the DEC 2060 running compiled PROLOG code.

Each clause in a KL0 program is translated directly into a sequential set of ***tokens,*** with one token for each predicate and argument symbol. These tokens are then packed into memory words, with one word capable of holding up to four tokens. For predicates the encoded information includes a pointer to the next clause with the same predicate symbol. For arguments the entry format is similar to that described for the WAM architecture, a small tag and a value field, with the tag indicating what kind of object (variable, constant, structure, etc.) the argument is.

The interpreter sequences its way through these token strings much as our earlier abstract interpreters did, checking tags and saving information as needed on four stacks:

- A ***control stack*** for call/return and backtracking information
- A ***local stack*** for the arguments to goals
- A ***global stack*** similar to the WAM heap for semipermanent objects
- A ***trail stack*** for undoing unified variables

These stacks are built from a memory in which each cell consists of an 8-bit tag and a 32-bit data field. The equivalent of choice points and environments are built from 10-word memory frames on the control stack.

A file of 1024 registers called the ***work file*** holds copies of the arguments for the current goals. Argument handling on a predicate call consists of ***argument copying*** of goal arguments found in one set of registers to a second set of buffer registers before unification tests. Further, this copy is in the order needed by the first literal in the rule's body, making for an efficient call to it. Unification is done on this copied set, so that if backtracking is necessary, the original arguments are still in the original register set and need not be reloaded from anywhere.

Other hardware characteristics of the machine include:

- A large microprogram store of 16K 64-bit words. Of this, 2.5K words support the basic KL0 interpreter, 1.5K are for garbage collection, 6K are for built-in predicates (160 of them), and 2K words are for other support functions.
- A 200-ns machine cycle.
- Hardware support for several stacks.
- Tags on memory objects, with good tag comparison hardware.
- Hardware support for "argument copying" and "structure sharing."
- An 8K-word two-way set associative memory cache.
- A logical addressing mechanism which permits several sets of stacks to be allocated at the same time.

As with the LISP-oriented machine ***FACOM ALPHA,*** this last feature permits the PSI to support several concurrent processes, all representing separate KL0 programs, at the same time. Each process has its own set of control stacks in its own address space, with only a common heap for communication between them. This in turn permits implementation, in KL0, of a complete operating system.

In terms of performance, the PSI, even with its interpreter base, executes at somewhere between 0.7 and 1.6 times the performance level of the PROLOG compiler for the DEC 10. Its average of about 30 klips translates into about 166 machine cycles per inference. Programs which require fairly simple list and other operations are on the low side of this ratio, mainly because compilers can optimize out many of the control operations. Programs involving complex unifications and/or heavy backtracking favor the PSI because of its heavy built-in support for them. Figure 19-12 breaks down the average time spent by the PSI per inference into several categories. Not surprisingly, more than 50 percent of the time is spent just in control and unification.

19.4.2 HPM—The High-Speed PROLOG Machine

(Nakazaki et al., 1985)

As a companion to the PSI Machine, ICOT has also designed the ***High-Speed PROLOG Machine (HPM)***, sometimes called the ***Cooperative High-Speed Inference Machine (CHI)***. This machine connects as an independent coprocessor to the PSI, and can execute KL0 programs roughly an order of magnitude faster than the PSI. Its major characteristics include:

Interpreter	Percent of Time		Average Machine
Function	Range (%)	Average (%)	Cycles Per Inference
Control	22–31	27	44
Unification	11–46	29	49
Trailing	2–8	5	8
Argument Handling	5–23	12	20
Cut	0–10	5	8
Builtins	11–31	22	36

Notes: Percentages abstracted from Taki et al. (1987).
Data measured over 4 benchmarks.
30K lips at 200ns cycle = 5,000,000/30,000 = 166 cpi

FIGURE 19-12
Where the PSI spends its time.

- Use of the PSI as a front-end interface to the user
- An instruction set based on the WAM model
- A matching optimizing compiler
- A large memory space for program execution, separate from the host's memory
- Support for logic language extensions that provide a multiprocess programming environment

The entire programming environment and operating system for the HPM is written in an extended PROLOG that provides built-in operations to help in three areas:

- "Side-effect" operations for fast system state changes
- Nonlocal exit mechanisms for exception processing
- Support for a multiprocess environment

The side-effect primitives permit conventional read-write access to memory areas called ***vectors,*** which hold such things as process or file control blocks, interrupt handler tables, current I/O assignments, etc.

The nonlocal exit primitives provide alternatives to PROLOG's standard cut and fail primitives, and are based on the ***catch*** and ***throw*** mechanisms introduced into many LISP systems (see Chapter 10). Here a literal of the form:

catch(⟨*label*⟩, ⟨*goal*⟩)

builds a special kind of choice point which contains the ⟨***label***⟩ field as a name. When a literal of the form

throw(⟨*label*⟩)

is encountered as a goal, execution backtracks as for a cut, except that the choice point retreated to is the last one established by a **throw** with the same ⟨***label***⟩ designator. The goal restarted after the catch is the one packaged as an argument to it.

The multiprocess environment support consists of predicates to do process exchange operations and interrupt and trap management.

Figure 19-13 diagrams the basic architecture of the HPM. As with the PSI, it is a highly microcoded machine, but instead of an interpreter, this microcode supports a variant of the WAM architecture directly. There are hardware registers corresponding to all the WAM registers, plus several controlling the currently executing process. This includes 32 argument registers for direct support of up to 32 arguments or temporary variables. These are stored in a register array which permits two simultaneous reads. A tag comparator on the output of the array permits very fast unification comparisons between any two objects.

Memory is divided into six areas, the code, heap, local stack, and trail from the Warren model, plus an area for storage of objects that create side effects when updated, and a system area for process and memory management information. A cache buffers the slower memory from the rest of the processor.

Execution of an instruction follows a three-stage pipelined flow similar to the Tick and Warren approach. In addition, the design permits dataflow operations to go on in parallel with stack address manipulation. Thus a simple get-list can take as little as 2 cycles if the argument register points directly to a list, or 8 cycles if the argument register contains a reference to a variable in memory that must be trailed.

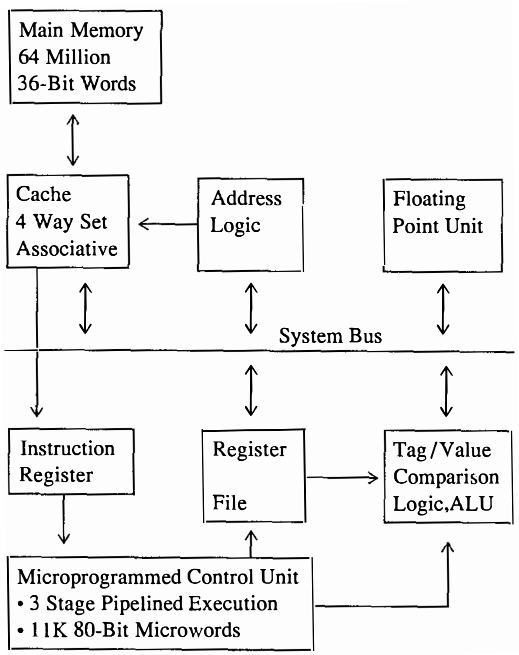


FIGURE 19-13
The HPM Machine.

The completed processor is built out of relatively low-density CML logic, runs at a machine cycle time of 100 ns, and takes up about 75 boards' worth of logic parts. When executing deterministic concatenation, the nine Warren instructions executed between call instructions (see the optimized **append** program of Figure 18-2) take about 35 machine cycles, for a performance of about 280 klips.

19.5 THE PLM AND ITS DERIVATIVES

(Dobry et al., 1985; Borriello et al., 1987; Dobry, 1987a)

Another direct implementation of the WAM model is the ***Programmed Logic Machine (PLM)***, designed and built at the University of California at Berkeley as an experimental part of the Aquarius Project. The machine consists of about 600 non-VLSI parts, and as with the HPM machines, serves as a coprocessor to a more conventional processing system. It is heavily instrumented, and has successfully executed several small benchmarks.

Although memory interface problems prevented execution of larger programs, the PLM did provide sufficient data to permit optimization of its design and conversion into a commercial product, the X-1. Both are described below.

19.5.1 The PLM

As with the HPM machine, the PLM design, Figure 19-14, is a modified version of Tick's original design, with differences from the HPM in several areas:

- A memory bus and memory that is shared with the host computer
- No general memory cache
- An optimized cache containing the current choice point
- A write buffer between the main dataflow and memory to let the processor go on while a store is in progress
- A deeper prefetch buffer

Memory is 32 bits wide; given that it is shared with a conventional processor, there are no extra bits, or other support for, tags. Instead, addresses are limited to 28 bits, and the top two bits are used for an initial tag of either reference, constant, structure, or list. The next two bits are for cdr coding and garbage collection. Constants use an extra two bits of tag to distinguish between tag types. Constant types that do not fit in 26

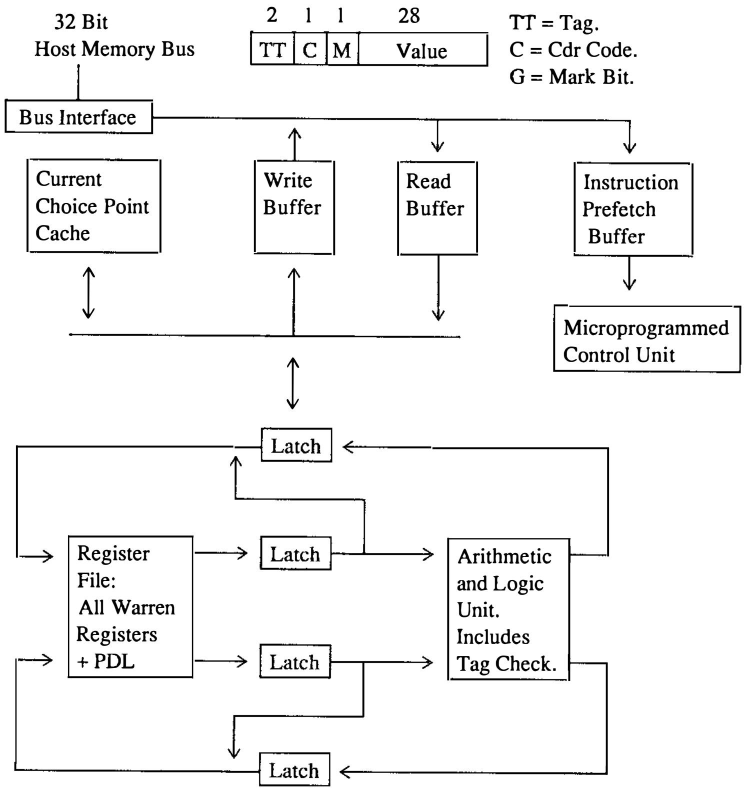

32 Bit
Host Memory Bus
2 1 1 28
TT C M Value
TT = Tag.
C = Cdr Code.
G = Mark Bit.
Bus Interface
Current Choice Point Cache
Write Buffer
Read Buffer
Instruction Prefetch Buffer
Microprogrammed Control Unit
Latch
Register File: All Warren Registers + PDL
Latch
Latch
Arithmetic and Logic Unit. Includes Tag Check.
Latch

FIGURE 19-14
The PLM Machine.

bits are referenced using the 26 bits as a pointer. Although a bit was allocated to it, garbage collection was not implemented on the PLM.

The *cdr-code* bit is used a little differently than discussed earlier. There is only one bit that indicates whether or not the cell it is in is a car (0) or cdr (1). If it is a car, then the next sequential word in memory is part of the list. If it is a cdr, the rest of the list, if any, is indicated by the type of the cell. This approach permits the prefetcher in the PLM a little advance notice that the next word must be fetched.

This form of cdr coding also permits some optimization of the unification of lists. As shown in Figure 19-15(*a*), conventional WAMs unify long lists with a series of unify-vars and get-lists. The PLM augments this by splitting the unify instructions in half. One set unifies the next element in line as long as it is marked as a car element. If a word marked as a cdr is encountered, it is decdred to find the next car.

The other class of unifys work similarly for cdr elements. Figure 19-15(*b*) diagrams the resulting code.

The PLM implements ***environment trimming,*** with an N register loaded by the call to indicate the size of the current environment to keep.

```
get-list A1
unify-cons 1
unify-var A4
get-list A4
unify-cons 2            get-list A1
unify-var A4            unify-cons 1     ;expects car
get-l ist A4            unify-cons 2     ;expects car
unify-cons 3            unify-cons 3     ;expects car
unify-cons nil          unify-nil        ;expects cdr
(a) Conventional WAM.         (b) PLM.
```

FIGURE 19-15
PLM unification against the list (1 2 3).

Another modification is in the switch-on-term, which includes 8 bit displacements if the register's tag dereferences to constant, list, or structure respectively. The case of an unbound variable tag is handled by simply dropping through to the next instruction.

Built-in predicates are handled in three ways. First, some predicates, such as **repeat** or **var,** are implemented with standard PLM WAM instructions. Other predicates use a combination of these and somewhat standard arithmetic, logical, and comparative operations implemented directly as PLM instructions. Finally, since the PLM is a coprocessor to a more conventional computer, there is a need to invoke programs in that computer. The approach taken in the PLM is to use an ***escape*** instruction, which copies the major PLM registers to a common area in memory and interrupts the host computer. At completion of the desired feature in the host, the PLM is restarted, and appropriate registers are reloaded.

Of particular interest in Dobry et al. (1985) and Dobry (1987a) are detailed statistics of the machine cycles required to execute individual instructions in different modes, plus a count of how many times each instruction type is executed during a set of benchmark program executions. The set includes 15 separate benchmark programs of an average of 191 source lines of PROLOG code each.

Figure 19-16 summarizes this data for the major classes of Warren instructions. Given that the timing equations for each instruction are often complex and a function of parameters that are not measured directly by either simulators or in the hardware, the columns labeled "Percent of total cycles" and "Average cycles per instruction" incorporate some estimates of what the parameters might be.

The results indicate that well over 50 percent of the machine's execution time is spent in the initial unification tests, even after the various indexing schemes have weeded out clauses with no hope of applicability. The time spent setting up for new goals (puts), and calling them (***procedural***), is less than 20 percent of the total.

Also from the original statistics, one finds that 10.91 percent of all instructions executed are call, execute, or escape. As was suggested ear-

Warren Instruction	Average Cycles per Instruction	Percentage of Executed Instructions	Percentage of Total Cycles
Unify	6.7	29.17	30.0
Get	9.6	19.38	28.6
Index	9.9	11.71	17.8
Procedural	4.25	15.0	9.8
Put	3.3	17.26	8.8
Other	4.45	7.31	5.0

Note: The average instruction requires 6.5 machine cycles.

FIGURE 19-16

Dobry's statistics on instruction times.

lier, if we count each of these as an inference, then each inference consumes on the average 6.5/0.1091=59.7 machine cycles, which at 100 ns per cycle equates to 167 klips for the benchmark set.

In contrast, for the standard determinate **append** and using a variety of compiler optimizations, performance can go as high as 526 klips.

Some final data in Dobry's paper permits some analysis of the efficiency of some of the PLM's design features. The initial simulation data assumed a PLM design where:

- All memory accesses take one cycle.
- An instruction is always available from the prefetch buffer when needed.
- There is no write buffer or choice point cache.

Under these conditions the average performance was 272 klips. When a realistic three-cycle memory and a finite prefetch buffer were factored in, this performance dropped 41 percent. Adding in the write buffer (to hide the three-cycle store time) and the choice point cache added back in 27.9 percent to give 206 klips, or 48.5 cycles per inference. This indicates that the original estimates were somewhat pessimistic. Tick (1988) goes into much more detail.

19.5.2 The X-1

(Dobry, 1987b; Ribler, 1987)

The X-1 is a commercial upgrade to the PLM that plugs into conventional workstations and interfaces with conventional operating systems and software running on that host workstation. Although it is very close to the PLM, there are some interesting variations:

- The interface to the host operating system, particularly for exceptions, memory management, and foreign calls (calls to the host via an escape), is more robust.

- Several different kinds of choice points are possible, which are selected by a modified try instruction.
- Numeric constants are either short 26-bit integers, or 32-bit integers and 64-bit IEEE-format floating-point numbers compatible with the host.
- The cdr coding used is the more conventional one of signaling if the next cell is the cdr of this one or the car of the next dotted pair.
- Garbage collection is implemented by a program running in the host and scanning the shared memory.
- Extra built-ins have been added to permit the X-1 to support not only PROLOG but also LISP.

The unique choice point introduced in Dobry's (1987a) thesis and implemented in the X-1 is a smaller version of the standard one discussed earlier, in which the argument registers are not saved when the choice point is built and thus are not reloaded when backtracking occurs. This permits implementation in many circumstances of a quick form of backtracking called ***sidetracking***. Such choice points are implemented when it is known that if the head/goal unification code fails, none of the appropriate argument registers will have been modified and thus need not be reloaded.

Support for LISP is another unique feature of the X-1. A compiler running on the host takes LISP code and converts it into an X-1 WAM program, from which point it is assembled and link-edited into an executable file. While the code is perhaps not as optimal a match as in some of the specialized LISP machines discussed earlier, it is relatively good and certainly understandable. As an example, Figure 19-17 diagrams (in our notation) the code for both the PROLOG and LISP forms of **append.** As before, the actual X-1 notation is a little different.

19.6 THE LOW RISC APPROACH

(Mills, 1986, 1989)

All the machines presented so far are heavily microcoded. This reflects the relatively complex nature of the WAM model, in which each instruction can require several, often a variable, number of operations to perform.

One of the strongest trends in computer architecture today, in contrast, is a move toward instruction sets that are extremely simple, and thus are easy to design with short machine cycles and have hopefully very high performance. Such machines are called ***RISCs*** (for ***Reduced Instruction Set Computers***), in contrast to architectures like the WAM model, which are categorized as ***CISCs*** (for ***Complex Instruction Set Computers***). Chapter 10 outlined the general design characteristics of RISCs and gave one example of a RISC for LISP, the ***SPUR***.

In a RISC architecture, complex CISC operations must be built out of strings of these simple instructions, meaning that a higher mip rate

```
append(nil,x,x).
append((h.l1),l2,(h.h3)):−append(l1,l2,l3).

append:  switch-on-term   C1,C2,fail   ;Fall thru if A1 unbound
         try-me-else      C2a
C1:      get-nil          A1
         get-value        A1,A2
         proceed
C2a:     trust-me-else    fail
C2:      get-list         A1
         unify-var        A4
         unify-cdr        A1
         get-list         A3
         unify-val        A4
         unify-cdr        A3
         execute          append
```

(*a*) A Prolog form.

```
Lisp:
(defun append (x y)
  if (equal x nil)
  y
  (cons (car x) (append (cdr x) y))))

append:  try-me-else      C2          ;build a sidetrack choice point
         get-nil          A1          ;if A1 not nil, go to C2 without
                                      ; register reload.
         trust-me-else    fail        ;don't need the choice point now
         put-val          A2,A1       ;Here x is nil, return y
         put-nil          A2          ;keep a nil in A2 for recursion
         proceed                      ;A simple return thru CP
C2:      trust-me-else    fail        ;eliminate choice point
         allocate                     ;save space for the (car x)
         get-list         A1          ;set up SP to point to list x
         unify-var        1           ;put car(x) into local var 1
         unify-cdr        A1          ;get cdr(x) into A1 for next argument
         call             append,1    ;recursive call, A2 same. Keep 1 var
         get-var          A8,A1       ;save the result
         put-list         A1          ;start doing the cons
         unify-val        1           ;car = car(x)
         unify-cdr        A2          ;leave a hole for cdr result
         get-val          A8,A2       ;put result in cdr
         deallocate                   ;release storage
         proceed                      ;return through CP
```

(*b*) A Lisp form.

Note: Above code adapted from Dobry(1987).

FIGURE 19-17
PROLOG and LISP code on the X-1.

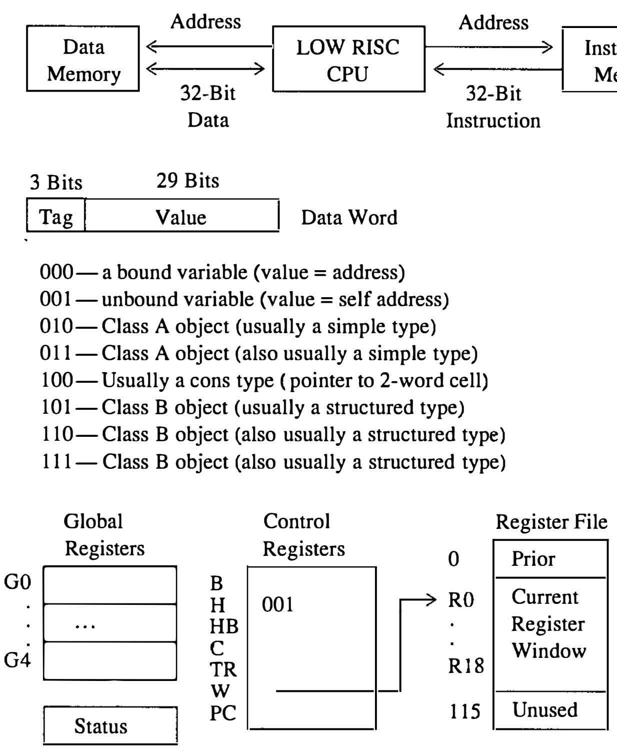
(more instructions per second) is needed to do what was done before in a CISC in the same time. However, if the basic operations needed are close enough to those supported by the machine, and the machine cycle time is short enough, the net time to solve the problem is less. Also, given a sufficiently talented compiler, this expansion of instructions (called ***open coding***) need not be the same for each occurrence of a CISC operation, but can be tailored to exactly the situation called for, thus removing extraneous instructions (and cycles) entirely. Borriello et al. (1987) studied just such an approach for the SPUR RISC; their conclusion was that the SPUR, originally optimized for LISP, could without modification reach about 29 percent of the performance of the PLM CISC, and with small modification about 50 percent, with a lot fewer logic gates.

This section describes the architecture of an even simpler RISC that is specifically oriented toward PROLOG, the ***LOW RISC*** (LOgic programming Windowed RISC) Machine. As before in this chapter, PROLOG on the LOW RISC uses the WAM, but only as an intermediate language which the compiler then expands into sequences of basic LOW RISC instructions.

19.6.1 Basic Architecture

Figure 19-18 overviews the architecture of the LOW RISC design. There are two memory busses, one for instructions and one for data. Data memory is tagged, with a 3-bit tag field and a 29-bit value field. Of the eight tag combinations, only "000" is meaningful to the hardware as a specific type, namely, a bound variable where the cell's field points to some other word. This is a ***forwarding reference*** in Chapter 9 terminology. Although the other seven tags have no hardware-imposed meaning, there is hardware support for rapid testing of "classes" of them, namely, the tag "010" or "011" are in class A, and "101," "110," and "111" are in Class B. This is useful for dealing with similar types such as cons, arrays, and strings, all of which utilize more than one cell for their representation.

There are 32 registers visible at any one time to the compiler or assembly-level programmer: 7 control registers, 5 global registers, 1 status register, and 19 general-purpose registers. These latter 19 registers hold arguments and variables (both temporary and permanent) for the current goal, and actually represent a ***window*** into a larger file of 115 registers that represents the top of a stack that extends into memory. The W register (***window pointer***) in the control set points into this file to select the starting point for the 19 registers. A change to W will change the set of 19 registers actually seen. If W is changed to something outside the current range, the appropriate parts of the register file are swapped into and out of memory. Note that this is different from SPUR, where each call shifts the register window by some fixed amount. Katevenis (1985) describes how such windowing works in more detail.



000—a bound variable (value = address)
001—unbound variable (value = self address)
010—Class A object (usually a simple type)
011—Class A object (also usually a simple type)
100—Usually a cons type (pointer to 2-word cell)
101—Class B object (usually a structured type)
110—Class B object (also usually a structured type)
111—Class B object (also usually a structured type)

FIGURE 19-18
Memory and registers in the LOW RISC.

With a few exceptions, the seven control registers correspond directly to WAM registers. The C register is an addition to handle cuts more easily (it will contain the address of the choice point calling the head of the current clause). The W (or window) register is used in place of the WAM E register to select the appropriate register set. The SP register is supported as needed by a register in the general set. There is also no single CP register, with R0 and R1 in the current window containing by convention the continuation pointer and the continuation window pointer.

The status register represents an extended condition code that is set by most instructions to reflect the normal arithmetic results (=0, >0,...), the tag of the last object generated or stored, and an indication of how the two tags for the inputs to the last arithmetic operation compared (=, >,...).

There is only one set of global registers, and they are used as needed, usually to define the boundary points of the various data areas.

All these registers have the same format as a memory cell, namely, a tag and a data field. In all but the control registers, when these registers are loaded, the tag field is changed by the instruction. In the control registers the tag field is permanently hard-wired to the tags 000 through 110. Thus, for example, when H is stored into the location specified by H, that location is automatically initialized to an unbound variable with tag 001 (the tag value wired to H).

19.6.2 Instruction Set

Figure 19-19 gives a somewhat simplified description of the instruction set. There are exactly seven instructions. The LOAD and STORE instructions permit transfers between any register and memory. The ADD and

3	5	1	5	1	1	1	15
xxx	R1	Z	R3	T	S	I	R2/Imm

LOAD —Memory[Source1 + Source2] → Register R3. Set just tag status.
STORE—Register R3 → Memory[Source1 + Source2]. Set just tag status.
ADD —Source1 + Source2 → Register R3. Set entire status
SUB —Source1 − Source2 → Register R3. Set entire status

Source1 = if Z=0 then register R1 else all 0's
Source2 = if I=0 then register R2 else Immediate
If T is set, use tag from source, else 3 bits from Imm.
If S is set, change status register to reflect result.

3	5	24
IF	cond	branch offset

Cond specifies one of 31 tests on status register.
If test is true, add offset to PC.

3	5	4	6	6	8
SWX	R1	Off1	Off2	Off3	Off4

Branch on tag of R1 as follows:
000 —Add 1 to PC
001 —Add Off1 to PC
01x —Add Off3 to PC (Class A tags)
100 —Add Off2 to PC
101,11x—Add Off4 to PC (Class B tags)

3	14	15
HOOK	address	parameter

Write parameter to designated address
(usually coprocessor control register)

FIGURE 19-19
A simplified LOW RISC ISA.

SUBTRACT instructions do arithmetic between registers. The program counter is a valid target for these instructions, meaning that branches of various sorts can be constructed from them.

The IF instruction tests some combination of bits in the status register and branches accordingly. The SWITCH instruction tests the tag of some designated register and does a five-way branch on the result. Finally, the HOOK instruction writes a parameter to memory. This would be used to communicate a command to a memory-mapped coprocessor to do something beyond the capabilities of LOW RISC, such as I/O or floating-point arithmetic.

These instructions make explicit assumptions about timing similar to that in the Stanford ***MIPS Machine***. The effects of both loads and branches taken (IF and SWITCH) are ***delayed***. Thus a load into a register (say, R7) is guaranteed not to happen until after the next instruction completes, meaning that this next instruction can reference R7 and still get the value before the LOAD. Likewise, on a branch the next sequential instruction is executed regardless of whether or not the branch is taken. In either case the reason is to at least partially hide memory latency and keep the internal pipeline running at full tilt.

19.6.3 Append Benchmark

As an example of how this architecture might be used to its fullest, Figure 19-20 gives a partial coding for the **append** program. This version is slightly faster than that given in Mills (1989). The code assumes many of the optimization techniques discussed in the last section, plus a smart compiler that can cut and paste the instruction sequences together as needed. Extensive use is made of the delayed branch, and care is taken that delayed loads are handled properly. In addition, several optimizations have been built in when it is known that, for example, an argument is something that will cause either read- or write-mode execution only.

The format for each instruction resembles normal assembly language, with the destination register placed first. Also, a NOP corresponds to any instruction that has no effect on anything, such as adding 0 to a register. Finally, because of the use of R0 and R1 as continuation values, the register usage shown here assumes that A1 is in R2, A2 is in R3, and A3 is in R4. As before, this format is somewhat different from that of Mills, and was adapted for its readability.

The reader should carefully check through this code, and relate it back to the original definitions of the WAM instructions. In particular, the reader should follow the numbered execution path that takes one recursion through a determinate execution where the first two arguments are lists and the last is an unbound variable. This path corresponds to one inference in our relative performance measure of Figure 19-5, and assumes that memory latency is quick enough to respond to read requests without stretching the delays any further. It includes:

```
;Following code is positioned BEFORE append start to save a branch.
;At A3var we know A3 is a variable—thus get-list A3 in write mode
A3var:     6 SUB R4,HB,R7,setstatus ;Compare A3 to HB to see if trailing
           7 IF ≥,next3
;Whether or not trailing needed, do next to write list pointer to A3
           8 STORE list#H,0(R4)     ;set A3 to list#H as we go
;Fall through here only if trailing needed
             STORE R4,0(TR)         ;mem[TR]←A3
             ADD TR,1,TR            ;update trail pointer
;Now continue with unifyTval A4—in write mode
next3:     9 STORE R5,0(H)          ;mem[H]←A4=R5
          10 ADD H,1,H              ;update heap pointer
;Now do the unifyTvar A3—again in write mode.
;Then drop directly into append for recursive call.
          11 STORE H,0(H)           ;cdr = unbound var = adr of itself
          12 ADD H,1,H              ;increment heap-always executed
; ******** Enter here for start of append *************
append:    1 SWITCH R2,A1var,A1const,A1list,fail ;test for tag of A1
           2 LOAD R5,0(R2)          ;Read A1 for dereference or list
;Following code executed only if A1 needs dereferencing. Tag=000.
             IF always,append       ;Loop back AFTER next instruction.
             ADD R2,0,R5            ;Copy dereferenced value back to R2

A1var: ... code not shown—handles indeterminate append

;Come here if R2=A1 a constant
A1const:     SUB A2,nil             ;Encode a nil constant in immediate
             IF ≠,fail              ;test for equality
             NOP                    ;wait for IF to finish
;Following emulates the proceed instruction
             ADD PC,0,R0            ;R0 is CP—load into PC
             NOP                    ;must wait for branch to work

;Come here if A1 a list. R2 points to car (use as SP). R5 is car.
;Thus we need not do a unifyTvar A4—its already done!
A1list:    3 LOAD R2,1(R2)          ;unifyTvar A1. A1 receives cdr
;Now do get-list A3=R4—dereference loop is in line as before
get2:      4 SWITCH R4,A3var,fail,A3list,fail
           5 LOAD R6,0(R4)
             IF always,get2
             ADD R4,0,R6            ;Again reset R4 to dereferenced value
A3list: ... code here corresponds to A3 a list—not shown here.
```

FIGURE 19-20
A partial LOW RISC **append** program.

- Two cycles at append: the SWITCH and the LOAD that follows it but is executed before the branch takes place
- Three cycles at list, where the SWITCH to var3 is taken, but again the instruction following it is executed anyway
- Three cycles at var3 (no trailing assumed)
- Four cycles at next3

Assuming a 50-ns machine cycle and a memory fast enough to respond to each LOAD without further latency delays, these 12 cycles correspond to a very substantial 1.6 Mlips. Slower memories would impact this, with a cache recovering a good part back again. Mills (1989) estimates that with a memory of about 200 ns access (a slow memory by today's standards) and a cache with an 80 percent hit rate, the performance degradation would be about 25 percent.

This speed is not without its costs, however. Mills also estimated in the same paper that a fully optimized LOW RISC PROLOG program would consume approximately seven times the storage required for the PLM. Intermediate levels of optimization, such as generating a WAM program in ***threaded code*** (see Kogge, 1982) and then interpreting it with an emulator written in LOW RISC, would substantially reduce code size, but at the equivalent cost of lost performance.

19.7 USING ASSOCIATIVE MEMORY

(Stormon et al., 1988b; Kogge et al., 1989)

As described in Chapter 8, an ***associative memory*** (or ***content addressible memory—CAM***) is one in which, instead of reading and writing to specific memory cells, one can either identify in one access all cells where certain bit fields ***match*** certain patterns, or write (***broadcast***) a value to a set of cells. These features, especially the match, provide some unique capabilities for supporting logic languages like PROLOG. This includes:

- Managing the bindings given variables
- Performing the unifications
- Identifying which clause(s) might match a particular goal

This section describes some of these techniques when an inference engine like PROLOG's manages exactly one solution at a time. Several of these techniques are usable in a reverse fashion when more than one solution is desired (as described in the next two chapters).

19.7.1 Variable Bindings

Perhaps the first place where associative memory is useful is in maintaining the values bound to variables during inferencing. In this case variable storage is not allocated when a clause is tried. Instead, each time a variable is bound during the unification process, a name representing the variable is paired with the binding value and placed in the CAM. Then, whenever the variable needs to be ***dereferenced*** (find its binding, if any), its name is compared against all the entries in the CAM. If there is a match, then that entry has the matching binding. If the bound value is itself the name of a variable, the process must be repeated. If no matching entry is found, the variable is unbound. Note that no initialization of a cell is needed for an unbound variable.

In addition, when a binding is made to a variable, it is also possible to use the CAM to search through all current entries for any value that equals the variable's name and replace that value by the new value with a single broadcast write. This covers the case where one variable without a current value (say, *x*) has been bound as a value to some other variables (say, *y* and *z*), and then later receives a value itself (say, 3). Not only do we record the binding of 3 to *x*, but also in one cycle we also update the bindings of *y* and *z* from *x* to 3. This saves future dereferencing cycles for *y* and *z* and is called ***path compression.***

The key trick needed to make either of these work is to pick a naming convention that allows unambiguous identification of a variable, particularly when the clause in which it is used is repeated many times in a recursive loop. Typically such names are formed by concatenating the binary representation of three integers [see Figure 19-21(*a*)]:

- A unique number given each variable within a clause
- A unique number given each clause within a program
- A unique number for each instantiation of a clause during execution of a program

The first of these corresponds to the index defined for the WAM. The last usually becomes something like the depth of the choice point stack (or equivalent) when the clause holding that variable was instantiated.

If no **assert**s or **retract**s are done during program execution, it is possible to collapse the first two of these numbers into a single number which simply gives each distinct clause variable in the program listing a unique number. This can simplify the process of deciding how many bits to allocate, and reduce the total number of bits that must be allocated to each of these name subfields in an entry of CAM.

Figure 19-21(*b*) gives an example of a set of bindings. We assume that these names and binding values are stored together in an CAM (called the ***binding array***) so that at least the name part can be searched associatively. As an example, the variable *u* in the original query [name=(0 0 2)] matches none of the names in the binding array and is thus unbound at the time the binding array represents.

The column labeled ***Binding Depth*** is the depth in terms of choice points that exists when the binding is made. Depending on other factors, it may or may not be stored in the binding array. Note that it is usually different from the depth at the time a variable comes into existence (the one used in its name). Thus the three bindings with a binding depth of 2 correspond to the three bindings made when a goal **q**(*w*, 13, *z*) is unified with the head of clause 20.

Having these names in an CAM permits a wide variety of interesting actions besides simple dereferencing. For example, a search on just part of a name, such as the definition depth, is possible. This might be useful to quickly identify when some particular variables, such as those

Depth	Clause	Var	Binding Tag#Value

Depth = Number of choice points active when clause is first tried.
Clause = Unique number for clause defining the variable.
Var = Unique number given each variable within a clause.

(*a*) Typical binding entry.

```
0: ?-p(w, 13, u). ;x labelled (0 0 1), u (0 0 2)
  ...
10: p(x, y, u):-q(x, y, z),r(z,u).
;   When tried against clause 0, x labelled (1 10 1), y (1 10 2),
;   u (1 10 3). z (1 10 4)
  ...
20: q(6, v, v).
;   When tried against q of clause 10, v labelled (2 20 1)
  ...
30: r(44, x).
;   When tried against r of clause 10, x labelled (3 30 1)
31: r(x, x).
;   When tried against r of clause 10, x labelled (3 31 1)
```

	Variable Name				
Binding Depth	Depth	Clause#	Var#	Binding Value	
1	1	10	1	var#(0 0 1)	w bound to x
1	1	10	2	int#13	13 bound to y
1	1	10	3	var#(0 0 2)	u bound to u
2	0	0	1	int#6	6 bound to x,w
2	2	20	1	int#13	13 bound to v
2	1	10	4	int#13	13 bound to z

(*b*) A sample binding when clause 30 is being tried.

FIGURE 19-21
Variable bindings in an associative memory.

in the original query, have been given values, without computing stack addresses or following dereference chains.

19.7.2 Simplifying Backtracking

One of the more complex operations in implementing PROLOG on a conventional machine is proper control of bindings after a backtrack. The WAM uses a ***trail stack*** to remember when a variable was bound, and an ***unwinding*** operation to reset those bindings. The total time spent is at least proportional to the number of separate bindings made and unmade. Note that some time is spent even when no bindings are ever reset; tests are often made of each variable's address at the time of binding to see if trailing is necessary to begin with.

Having an CAM contain bindings can greatly simplify this, and make the backtrack time independent of the total number of variables either bound or unbound. For many programs this can be significant.

At least two approaches to performing this have been proposed. First (see Naganuma et al., 1988), the CAM containing the bindings can be assumed to be an unorganized pool of cells, each of which can be allocated to any binding. A status bit associated with each memory cell can be used as the equivalent of a ***free bit*** to indicate whether or not that cell contains a useful binding. When a new binding is made, the memory is searched for a cell with a free cell, and the status bit is flipped. When a search for a binding is necessary, only those cells whose status bits are set to "in use" are searched.

Then, each variable name can be augmented by a number reflecting the depth of the choice point stack at the time the binding was made [the leftmost column in Figure 19-21(*b*)]. Now during forward execution no trailing operations of any kind are needed. When a backtrack is necessary, however, we do a search though the CAM for all locations that are in use and whose binding depth equals the depth of the choice point being revoked. In one machine cycle, all bindings made at that time can be identified in parallel, and their status bits can be switched from in use to free.

In fact, with a small amount of logic at each cell to augment this status bit, all cells with binding depths greater than some number can be deleted. This opens up the possibility of using some of the intelligent backtracking schemes discussed earlier to identify which choice point to return to and then reversing all associated bindings in one cycle.

Figure 19-22 gives an example of this approach based on the earlier example of Figure 19-21.

A second approach to unbinding at a backtrack (see Stormon et al., 1988b) requires a somewhat more complex set of logic attached with each

Binding Depth		1/0 = In use/free
1 1 10 1	var#(0 0 1)	1
...		0
1 1 10 2	int#13	1
2 0 0 1	int#6	1
...		0
2 2 20 1	int#13	1
1 1 10 3	var#(0 0 2)	1
2 1 10 4	int#13	1

(*a*) Before backtrack.

1 1 10 1	var#(0 0 1)	1	
...		0	
1 1 10 2	int#13	1	
		0	*
...		0	
		0	*
1 1 10 3	var#(0 0 2)	1	
		0	*

* = cells freed by backtrack
(*b*) After release depth 2 bindings.

FIGURE 19-22
Unbinding via binding depth search.

CAM cell but does not require the binding depth appended to each name. Instead, the cells are allocated sequentially from the memory, with each choice point generating its own set of bindings on the top of the current stack. Also, the status bit now indicates the top of a set of cells associated with one choice point. Thus the "topmost" cell whose status is "1" (and above which all cells are "0") is the cell with the last set of bindings.

With this usage, on a backtrack we identify all cells between the topmost "1" in its status bit and the next topmost "1." All the cells in between represent bindings that are to be undone, and setting their status bits to "0" (a one-cycle operation in a properly designed chip) essentially frees them. Stormon (1988a) has a good description of such a chip; Figure 19-23 gives the same example as before.

19.7.3 Head-Goal Unification

One way of enhancing performance when bindings are held in an associative memory is to combine the dereferencing step with at least a partial unification. The dereferencing discussed above uses only the name part of the binding in a match, but there is nothing (except perhaps the width of the memory) to prevent matching on the value. This is of particular power when the formal argument in the clause head being tested is a constant whose complete value fits in the value field. Now, a match back from the memory indicates that not only does the variable argument have a binding, but its value is exactly what is desired. Unification is totally complete.

If a no-match occurs, it could be because the binding does not exist, because it exists but needs to be dereferenced, or because it exists but is the wrong value. Exactly which occurred must be determined by other steps.

....	0
....	0
1 10 4 int#13	1*
2 20 1 int#13	0
0 0 1 int#6	0
1 10 3 var#(0 0 2)	1
1 10 2 int#13	0
1 10 1 var#(0 0 1)	0

* = topmost binding

(*a*) Before backtrack.

....	0
....	0
	0
	0
	0
1 10 3 var#(0 0 2)	1*
1 10 2 int#13	0
1 10 1 var#(0 0 1)	0

(*b*) After backtrack.

FIGURE 19-23
Unbinding via an explicit stack.

As an example, consider Figure 19-21. When attempting to unify **r**(z,u) with **r**(44,x), we could construct a compare value with a name (1 10 4) (variable z in clause 10) with 44 (the desired value). No match is found, so we search on just the name z (1 10 4), and find a match whose value is a constant 13. No match is possible, so the unification fails.

To be precise, the steps in building a combined dereference and unification between two arguments, one of which is a formal variable and the other of which is a simple constant, goes something like this:

1. Construct the name of the formal variable and pair with the desired value.
2. Compare with the binding array.
3. If a match is found, continue—the formal variable is in fact bound to the desired value.
4. If no match is found, try a second match with just the formal name.
5. If again no match is found, then the formal variable has no binding—add the match entry to the binding array as a new binding.
6. If a match is found, look at the tag of the bound value.
7. If a variable tag is found, pull out the new variable name, pair with the desired value, and repeat the entire process.
8. If anything else is found, a mismatch has occurred—declare a unification failure.

19.7.4 Unifying Complex Structures
(Sohi et al., 1985)

Even more interesting than unification of simple constants is unification of complex objects such as lists. In the WAM this requires a series of relatively complex instructions to cycle through the pointers of the underlying structures. With CAM, however, we can again explicitly name things (leaves in this case), and then do searches on those names, with expected values attached.

The trick to the naming convention here is to use a labeling scheme which assigns a number to every node of a list. The root cell is "1." Then the car of every cons cell receives a label of two times that assigned to the cons, and the cdr receives 2 times the parent plus 1. A simple analysis reveals that this guarantees a unique number to each node, even though for nonbushy lists the numbers are relatively sparse.

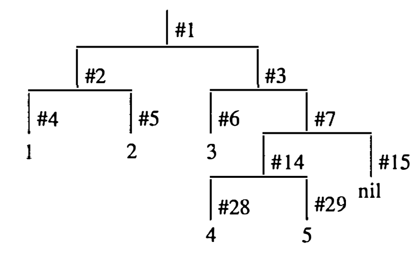
What goes into the binding array is the labels and values of only the leaf nodes, the nodes which are not themselves cons cells. Further, we will express each label as a binary number, left-justify so that the leftmost bit is always "1," and fill the rightmost bit positions with a special code of ***don't care***. [This latter code requires that each bit of the CAM actually be able to take on three code values—"0," "1," and "don't care." This usually requires two physical bits of memory (often called a *trit*) for each logical bit. See Stormon (1988a) for a more complete description.] Figure 19-24(*a*) gives an example.

Now, assuming that this pairing of labels and values is in associative memory, consider how to unify this with a formal list expression like ((1.*x*) *x*.*z*) [see Figure 19-24(*b*)], where none of the variables have bindings yet. Instead of tracing through the list itself via car and cdr, we will take the leaves of the formal arguments one at a time, pair the value (if any) with the leaf label, and attempt to find a matching leaf in the CAM. The first such leaf (110xx 1) has an exact match in the CAM, so unification continues. The second leaf (with label 101xx) is a variable *x* with no binding as yet. Consequently, we search the CAM for any entry whose label equals 101xx. There is only one such, and it has a matching value of 2, so that value is paired with whatever is the labeling for *x* and is added to the binding array in CAM.

The third formal argument leaf (at 110xx) is the variable *x*, which already has a value 2, so the pattern compared to the CAM containing the list leaves is (110xx 2). Again there is an exact match.

The final formal argument *z* (at 111xx) is also a variable without a current binding. When matching just its label against the CAM, not one but three cells respond (those with labels 1111x, 11100, and 11101). This should indicate to the unification algorithm that what should be bound to *z* is a list. The values that match should be read out one by one, and copied for a binding for *z*.

Assume list: ((1.2) 2 (4.5))



Node	Binary Label	Value
4	100xx	1
5	101xx	2
6	110xx	2
15	1111x	nil
28	11100	4
29	11101	5

#n implies label for node is n
x is a don't care value

(*a*) Leaf node labelling.

Pattern: ((1.x) x.z)

Pattern		
Label	Value	Number of Matches
100xx	1	1 — exact
101xx	x	1 — make binding
110xx	x	1 — exact
111xx	z	3 — bind sublist

(*b*) A sample pattern match.

FIGURE 19-24
Encoding lists for associative access.

One problem with this copying is the labels of the sublists leaves. First, they have to be readjusted to reflect their new position as a complete list by themselves. This requires stripping off the leading "11" from each label. Second, if we store this binding in the same CAM as the other actual list, then something has to be added to prevent confusion when other list searches are made. This is most easily done by appending some unique identifier to the front of the label for each node in a common list. This common identifier would also be stored along with the "list" tag as the binding value given a variable to receive the list, and would be used to construct list labels for matches later on. It is left as an exercise to the reader to construct a mechanism for generating such labels.

Finally, the observant reader may also have noticed that all the labels of Figure 19-24 have a leading "1." This is in fact always the case with the labeling scheme described here. To save bits, most real labeling schemes simply normalize this leading "1" out of all labels.

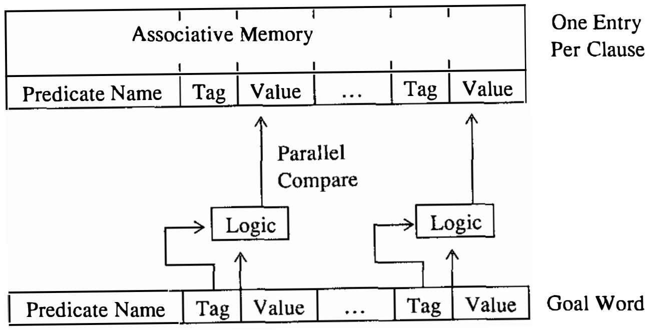
19.7.5 Clause Filtering

The main purpose of ***indexing*** and the switch-on-term instructions was to rapidly eliminate from consideration large subsets of clauses that clearly could not unify with the current goal. While these techniques are normally effective performance enhancers, they break down when either there are many ***assert***s or ***retract***s that dynamically change the clause set, or the choice of clauses cannot be determined by testing a single argument alone.

Associative techniques can help in both cases by performing at least partial unification matches across more than one argument position. Information on each clause head is stored in a CAM along with pointers of some sort to the right-hand side of the clause. Now, when a new goal is to be handled, the arguments for that goal are compared in various fashions against the stored values. All clauses that survive the comparison process are then candidates for the goal. For obvious reasons this is called ***clause filtering,*** and as before there are several techniques.

The most direct approach is simply to try unification between all the actual and formal arguments without regard for variable bindings. We assume for the purposes of an initial trial that a variable matches anything in its matching argument position, whether or not it is used in multiple places and picks up a bindings along the way. This must be checked for after the fact.

To demonstrate, envision an associative memory (Figure 19-25) as wide as the maximum number of arguments plus a field for an encoded



Logic Replaces Variables by xxx#xxxxx
Lists by List#xxxxx
Structures by str#xxxx

Where x = don't care (match anything)

FIGURE 19-25
Associative clause filtering.

predicate name, and as deep as the number of clauses in the program. Each argument field is as wide as a normal cell (tag+value), with a variable in either the actual or formal position represented by all "don't cares." When either argument position is not used (such as the third argument of an **eq** predicate), its tag and value take on a distinct value, say, all "1"s or, again, "don't cares."

In addition, any argument (goal or actual) which is some kind of a reference to a data structure elsewhere (such as a list or a structure) should keep its tag intact but change its value (a pointer) to "don't care." This reflects the fact that two lists can have identical entry values but be in different locations.

With this in mind, an actual goal is constructed as a long data word that contains the name of the predicate and tag+values for each actual arguments. This word could be constructed from the argument registers in the WAM, making integration with the WAM relatively easy. The accuracy of the unification is improved if all variables in this goal are dereferenced as far as possible before filtering. Any unbound variables that remain are replaced by "don't cares."

Now in one CAM cycle the actual goal can be compared against all program clauses, and only those entries which have some chance of succeeding signal a match. All others do not.

If there are no matches, we have an immediate backtrack, without any choice point builds, try-xxxs, or any kind of clause chaining. If there is exactly one match, that is the only clause we need try, and again no choice point need be built. Only in the multiple-response case is a choice point necessary. In this case we could recover a precompiled pointer to a try chain, or perhaps a bit vector of the responses.

Having identified a potential clause, a more formal check must be made by actually establishing any necessary bindings. This can be by a standard set of WAM instructions or by replacing appropriate entries in the goal registers by the bindings and trying the unification again.

An acceleration on this idea would encode in the references to a structure or list something about the first entry of the structure, such as its tag and perhaps a hash of its value. This would permit a simple test on those values without again having to chase down pointers.

In a more complex arrangement, one could envision a small binding array with each clause entry operating in parallel to the associative match to capture any bindings and spread them out as appropriate. Shankar (1988) describes just such a system, in which, in fact, the associative clause head array is actually a cache for frequently used clauses found more completely in a backing conventional memory. Shankar's simulations indicate that performances as high as 1.8 Mlips (14 cycles per inference) on **append** may be possible.

Yet a simpler approach uses the concept of ***superimposed code words*** to construct encoded forms of each argument and then logically **or**ing the encodings together (Colomb, 1985; Colomb and Jayasouriahn, 1986). This creates a smaller word but increases the "false alarm" rate. A typical encoding might be a 2 out of n code where a hashed form of the tag and value specify the location of two "1"s within the field. Very often some sort of cyclic shift based on the argument position is employed so that the same value in different argument positions would set different bits. As long as the same encoding scheme is used for both goal and actual arguments, this can be quite effective. Using a variant of this, Stormon et al. (1988b) have measured success rates of up to 89 percent in unambiguously identifying the one and only clause that might work for some goal over relatively large programs.

19.8 PROBLEMS

1. What does the mix of Figure 19-3 tell you about the average number of literals on the right-hand side of clauses executed in the programs that were measured? (*Hint:* Look at the calls, executes, and proceeds.)
2. Figures 19-3, 19-12, and 19-16 all give at least partial insight into how various machines spend their time. Compare and comment.
3. Pick the six highest-percentage instructions from Figure 19-3 and open-code then in an instruction set of your choosing. Estimate the number of instructions executed and references made to memory for data. What seems to be inefficient? (Describe the approach you took for tagging.)
4. Define how a compiler for a simple s-expression language might generate code for a WAM. What kinds of architectural features are missing from the original WAM and where are there unused features?
5. Use the statistics of Figure 19-4 to get some handle on how useful path compression like that described in Section 19.7.1 would be.
6. Compare the number of data references required by the various implementa-

tions described in this chapter for one inference of the deterministic **append** program.

7. Implement the LISP **rev** function in both PLM and LOW RISC code. Comment on what you think are the strong points and weaknesses of each relative to the specialized LISP machines of Chapter 10.
8. Finish up the missing code in Figure 19-20.
9. Write a general unification routine in LOW RISC code. Assume the arguments to be unified are in R1 and R2, respectively.
10. Architect a new PROLOG-oriented RISC that has tag support like either SPUR or LOW RISC, but that:
    - Does not employ a sliding register window
    - Keeps all arguments in a memory stack, not registers
    - Has perhaps two to three machine registers for working values

    Show how a sampling of WAM instructions such as SWITCH, getTvar, getlist, etc., are open-coded, and describe which optimization techniques might be applicable.
11. Draw the bindings after clause 31 of Figure 19-21(*b*) completes.
12. Rewrite the variable/constant unification algorithm described for associative memory assuming that you can also selectively search on the tag field of the value.
13. Describe an algorithm for constructing a label that would return the result of applying an arbitrary string of **car**s and **cdr**s to a list stored in a CAM as pictured in Figure 19-24.
14. Develop a complete algorithm for unification assuming an associative memory to hold the binding array.
15. Describe how you might define an abstract machine for PROLOG in which all of the memory is associative.

CHAPTER 20

ALL-SOLUTIONS INFERENCE ENGINES

By design, the PROLOG inference engine goes after exactly one solution (binding of values to the query variables) at a time. In many cases this is what is desired, but other applications lead to interest in the set of all possible solutions. This can be found using PROLOG, but usually only by encapsulating the original query in a new one that involves a **set-of, bag-of,** or the equivalent, which causes the inference engine to backtrack repeatedly through all possible inferences, binding and unbinding all possible assignments, and catching the ones that work into some sort of list of responses.

There are inference engines that are designed from the beginning to look for all possible solutions. They have usually been designed for languages that, although they are still based on predicate logic, are considerably different from PROLOG. This chapter discusses two of these: the ***relational model*** and ***production-rule systems***. The former, which is closely related to the *database* field, goes back to the basics of Chapter 2—the processing of tuples of objects that are organized into sets called ***relations***. A query in such a program looks to compute an entire relation whose tuples are constructed from pieces of other tuples that satisfy some relationship. Individual program statements specify how to build such new relations from old ones. Unlike PROLOG, tuple components usually are limited to simple objects, greatly simplifying the equivalent of unification and introducing opportunities for parallelism.

The second type of language addressed here, production rules, look somewhat like PROLOG statements "turned around," such that all the antecedent sides of the statements are executed as fully as possible to locate all possible tuples that satisfy them. Then one of these enabling tuple sets is chosen (the clause "fires"), and the consequent is activated. This corresponds to the definition of a ***forward-chained*** inference engine as defined earlier. Structures and complex objects are allowed in tuples, but they usually must be fully grounded in the actual sets of facts that are beat against the rule antecedents. Thus the unification is something between that of the relational model and full PROLOG.

A production-rule system shares more than just the concept of computing all tuples with relational systems; the basic relational operation called a ***join*** turns out to be the key computation carried out internally to an optimized production-rule inference engine. In addition, however, perhaps the strongest difference between either of these and any of the more classical logic-based systems is that rather than simply recording that certain tuples belong to some relation, the tuples "produced" by either model are passed to pieces of code which often dynamically modify the relations. The same tuple may be in some relation at one time and out of it at some other time. While this violates our formal definition of a relation as a "timeless" object, the process does greatly resemble many real-life programming metaphors, and it has found frequent use in such diverse applications as commercial databases, expert systems, and other artificial intelligence applications. In a sense it represents the beginnings of inference engines based on ***nonmonotonic logic,*** whereby the inference engine can both expand *and contract* the set of tuples belonging to a relation as a function of time. This is in contrast to conventional ***monotonic logic,*** where the truth of a tuple relative to a relation is fixed in time, and running an inference engine longer merely monotonically "increases" our knowledge of which tuples really are or are not in the relations.

Because of this emphasis on computing all solutions, both kinds of systems open up opportunities for parallel execution, which will be introduced briefly here. A more general form of such parallelism, called ***OR parallelism,*** will be discussed in Chapter 21.

In terms of what is of most importance to gain from this chapter, a thorough understanding of the **join** operator and some of the ways that it might be implemented is first. This is followed by the mathematical rigor on which the relational model is built, and an understanding of how production systems can be compiled into yet another abstract machine formulation.

20.1 DATABASE OVERVIEW
(Ullman, 1982; Date, 1989)

A ***database*** is a (usually large) collection of data that is stored in a computer, is of value by itself, and persists in approximately the same form for relatively long periods of time. Examples include school class records, airline flight schedules and seating assignments, and employee salary and assignment data. The major uses for a database are to ***query, insert, delete,*** or ***update*** some information stored in it.

The major characteristic of a database is that it consists of many smaller "pieces," often in several distinguishable ***files,*** where each file holds multiple ***records*** or ***tuples,*** and where each of these have multiple ***fields, attributes,*** or ***components.*** There is an internal structure to this data that permits "navigation" through it to find some piece requested by a user.

The key factors distinguishing the various models for computing with such databases include the structure of the data as seen by the user, the operations the user can perform on it, the efficiency of these operations (often tied directly to how the data is stored and indexed), the ease of modifying the structure of the data, and the ability to utilize several separate databases in complex queries.

20.1.1 Components of a Database System

Figure 20-1 diagrams the components of a typical database system. The amount of data in most real databases is usually too large to fit in a computer's main memory, and must therefore be stored on one or more rotating magnetic media such as disks or tapes. This means that it must be accessed in pieces as needed. For simplicity this access is usually staged, with the highest level requesting the particular structured data, the next lower level representing the files and logical records where that data is found, and the lowest level specifying the specific tracks and sectors on the real devices that hold the binary representation of the data.

This staging of data is reflected in the software that handles it. At the highest level is the ***Query Processor,*** which takes ***queries*** somewhat similar in nature to PROLOG from the user (often multiple simultaneous users) and translates them into commands relevant to the particular databases at a logical level. Applications programs which consist of preprogrammed queries can also be translated into such commands by the equivalent of compilers (called an ***Application Language Processor*** in Figure 20-1). This is the level we will be emphasizing in the rest of this chapter.

At the next level down is the ***Database Manager.*** This program takes requests for structured data from the higher levels and translates them into requests for particular portions of logical files. To do this requires a description (***Data Description Schema***) of how these translations are to be made. Usually the designer of the databases has provided these earlier when the databases were created. Again a compilerlike program, called a ***Data Description Processor,*** takes the formal descriptions and translates them into the format needed by the database manager.

The lowest level of software is usually the ***File Manager.*** This pro-

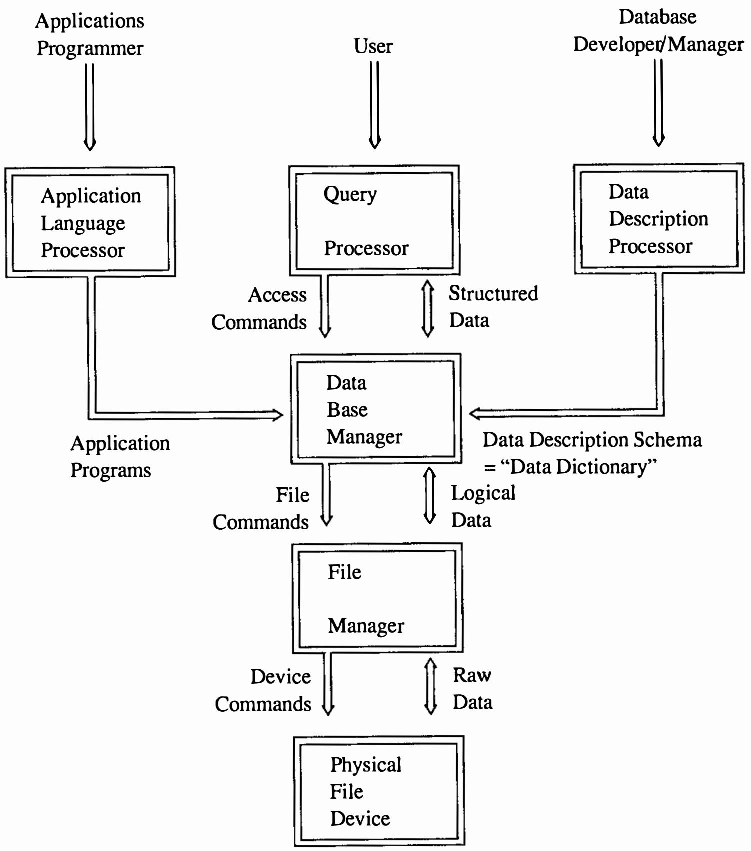


FIGURE 20-1
A database system.

gram takes requests for parts of logical files and maps them into the low-level commands for particular pieces of physical files on real devices.

20.1.2 The Entity-Relationship Model

The ***Entity-Relationship Model*** is an informal and high-level model of database structures that tends to model "real-world" applications relatively well. It is based on the definition of an ***entity*** as something that is distinguishable in some quantifiable way from all other entities (almost exactly our definition of ***object*** from Chapter 1). An ***entity set*** is then a set of similar entities, such as the set of all humans.

A database will contain records for each entity associated with it, where each record is augmented by information that permits identification of how it relates to other entities. In most cases a single database will contain many different types of entities, such as students, classes, classrooms, teachers, grades, etc., in a university database.

An ***attribute*** is a property of an entity that helps distinguish it from others of the same type. In the correspondence between an entity and a record, an attribute thus corresponds closely to a ***field*** of a record. Treating an entity as a tuple makes an attribute correspond to a particular component position of the tuple. The set of possible values for an attribute is called its ***attribute domain*** (or ***domain*** if there is no possibility of confusion). Individual entities usually have several attributes. For a **student** entity the attributes might include his or her name, address, student number, major, matriculation date, etc.

A ***key*** is an attribute (or set of attributes) whose values can uniquely distinguish entities. It is like a "fingerprint" or "name" that permits unique identification of an entity. There may be multiple keys (e.g., a file of students could have separate keys on names, Social Security numbers, student IDs, etc.), and there may be attributes that are in no way associated with keys (such as the color of a student's hair).

Keys are used for efficiency. If we have an attribute that is a key, then a single table indexed by attribute values, with entries consisting of pointers to individual entities, provides a very fast access mechanism for the database.

A ***relationship*** is a statement that various entities are related in some way. When this can be formalized to the point of grouping entities into tuples which accurately represent the relationship, we get something very close to a ***relation*** as defined in Chapter 2. The same database may have many such relationships defined among the descriptions of its entities.

Further, if, as was done for functions, we assume that all relationships are sets of two entity tuples (***pairs*** consisting of the cross-product of the ***domain set*** and the ***range set***), then there are several kinds of standard relationships (see Figure 20-2 for examples):

- ***One-to-one,*** where in each position of the pair no entity appears more than once
- ***Many-to-one,*** where an entity can appear more than once in the domain but at most once in the range
- ***Many-to-many,*** where the same entity can appear many times in either position

As before, there is nothing to prevent either or both elements of such pairs from themselves being tuples of arbitrary size.

The reason for the interest in this classification is efficiency of implementation. For example, assume that a standard query to be handled is a search of a database in which entities are linked in several relationships and the one of interest is many-to-one in nature. An example might be students in a class, or suppliers for some part. Assume also that this typical query is structured to provide one component of the pair (such as the class) and to retrieve the other (which students are in the class).

An efficient implementation would then make sure that there are

Entities:	Attributes:
Students	Name, Id#, Address, Major,...
Classes	Title, Dept, Number, Section, Time, Room...
Faculty	Name, Address, Position, Dept., ...
Departments	Dept., Office

Sample Relations:

1 to 1:
Chairman-of: Faculty × Departments
Current-Instructor-for: Faculty × Class-Section

Many to 1
Majoring-in: Students × Departments
In-School-of: Departments × School

Many-to-Many
Enrolled-in: Students × Classes
Has-taught: Faculty × Classes

FIGURE 20-2
Entity relationship model for sample problem.

links in the database between entities that represent the known components and those that are desired, and that there is some fast way (perhaps by using the known entity's key field) to locate those entities in the known component's position that match some particular query value.

Important characteristics of such an implementation include:

- How fast one can perform queries for which the system was designed
- How fast (if at all) one can perform different queries, particularly ones that are "backward" from the ones for which the database was designed
- How difficult it is to insert, delete, or modify entity descriptions
- How feasible it is to modify the description of a class of entities, such as adding another attribute to each of them

The latter two characteristics in particular become quite sticky when the information involved in the modifications is part of that which forms a relationship modeled directly in the database.

20.1.3 Nonrelational Models
(Ullman, 1982)

Several more formal models of database systems were developed before the relational model. For the most part they were based on implementation concerns, not ease of use, and they have proved notoriously difficult to adapt to new types of queries. We describe two of them briefly here so that the advantages of the relational model will become apparent.

The first of these models is called the ***Hierarchical Model,*** and it was designed to handle directly only the many-to-one class of relationships. The basic implementation technique groups all entities of the same kind in separate files and interconnects them in a treelike fashion. For a particular relationship, any entity that is in the range position will be the root of a small tree in the database, with links to and/or from all the entities that are in the domain positions of tuples that map to it in the relationship. An individual entity can be a leaf of only one tree. For example, in the relationship **residents-of** over the tuples of **humans** and **cities,** each entity representing a city is a root of a tree where each entity representing a person has a pointer (or key) to exactly one other person's entry. The city entry has a pointer (or key) to the first person so linked.

Note that the database consists of many separate descriptions of different types of entities, all joined in a set of small trees. If there are multiple relationships represented in the database, it is entirely possible for the same entity to be in many different trees, either as roots or leaves.

Usually, fast access via keyed index tables or the like is available to the entities that represent the roots of these trees. This makes queries where something is known about the desired range elements of the relationship fairly efficient. Fast access to the relevant root entities can be followed by a highly constrained search of the children on that root. Thus, if the relationship maintained by a database is **resident-of,** for example, a query looking for certain kinds of individuals who live in a specific city would involve a fast lookup to identify the record associated by the city, followed by looking (via the hierarchical links) at all residents for ones that match the desired profile.

Queries going the other way may be quite expensive in terms of time. For example, to see what city an individual lives in may require a tree-by-tree search, one per city. For a database covering the United States, for example, that might be hideously expensive in computer time.

Inserts, deletions, and modifications are also nontrivial. For example, it may be impossible to add a new student until you first also specify which classes he or she is enrolled in (even if that turns out to be a "fictitious" "not-enrolled-yet" class entity). As another example, modifying an entity used as a leaf of some tree might require another search of the whole database several times to make sure that all relationship links based on that entity remain consistent.

The second of the nonrelational models, the ***Network Model,*** was also built on the Entity-Relationship model, restricted to binary many-to-one relationships. The reason for this is that we can implement each relation by including with the storage for each entity a single link (for each relationship) to the single entity on the other side of the relation pair. This pointer can be implemented in several ways:

- As a separate pointer or index field in the storage associated with each entity in the domain.

- As a separate file the same length as the domain set of the relation pair, where each element is a simple pointer to the entity from the range that matches.
- By constructing for each entity in the range of the relation a set of links to the first-element entities that map into it. Each such set is called the ***set occurrence*** of that entity, with the entity called its ***owner.***

The two operations supported by such pointers consist of ***selection*** of entities based on the attribute values given for the domain of the pairs, and ***navigation*** through the file by following pointers, particularly when the domain and range are the same set.

20.2 THE RELATIONAL MODEL

(Codd, 1970; Ullman, 1982, chap. 5; Maier, 1983; Date, 1983; Date and White, 1989)

The ***relational model*** of database design is a much more formal approach than the two models discussed briefly above. It is based firmly on the definition of a ***relation*** from Chapter 2. There are no explicit links, no restriction on the kinds of relations that can be modeled, and no need to mix many different entity types and links between them in a single file. As with Backus and FP, its developer, E. F. Codd, was honored with the ACM Turing Award for this work.

Intuitively, a relational database consists of one of more relations, each expressed as a set of tuples and often represented as a ***table.*** Each row in such a table represents one of the tuples currently in the relation, and each column represents one of the component positions of all tuples. The table's columns have names (***attributes***), with matching domains of acceptable values. There is absolutely no intrinsic order to the rows within the table. There is also absolutely no required order for the display of the columns.

Figure 20-3 gives an example of a relation table in which the tuples give information on different classes. Other tables might list instructors and classes, students and classes, rooms and their capacities, instructors and their office hours, etc. All such tables would together comprise a relational database.

Columns:	Dept.	Number	Time	Room	Instructor	Dept.	Number
	Math	200	10AM	308	Ghose, K.	CS	515
	CS	390R	6PM	308	Stone, H.S.	EE	401
	CS	214	2PM	106	Smith, A.J.	EE	401
Tuples:	EE	251	10AM	212	Stone, H.S.	CS	390R
	ME	147	10AM	102	Intrierie, M.	CS	214
	CS	212	2PM	102	Iwobi, M.	CS	212
	EE	401	4PM	202	...		

(*a*) 1989-course-location table. (*b*) Instructor-course table.

FIGURE 20-3
Sample relations.

Codd's primary contribution was the formalization of a ***relational algebra*** that combines a basic set of functions and predicates with a sound mathematical description of their meaning. With this basic set, wide-ranging expressions which compute new relations from existing ones can be written. Although this sounds like a cross between logic and functional programming (and there are strong parallels), there are several major differences:

- Entire relations are accepted as input to the expressions, and conceptually complete relations are delivered as their results.
- The contents of these relations change as a function of time.
- The relations are nearly always finite.
- Individual values in tuples are nearly always simple ground constants, with no variables or complex structures.

The following subsections describe formally the concepts of a relational schema, the basic operations that may be used in relational expressions, and how they together form a complete algebra.

20.2.1 Relational Schemas and Relations

A ***relational schema*** is a formal description of how certain relations, or tables, may be built. It is *not* a relation itself, only a description of what qualifies as a valid relation of that type.

Figure 20-4 summarizes the major components of a schema. It has a name **R,** and has associated with it a description of a tuple (row in the table form), where each component of a tuple has an ***attribute name*** $\mathbf{A_k}$ (the table's column identifiers), and a set of valid values for elements in

Relational Schema: $R[A_1,\ldots,A_n]$ where:

R = name of schema

$\{A_k\}$ = set of attribute names

A_k = name of k'th attribute

$D_k = \mathrm{dom}(A_k)$ = domain of values for k'th attribute

$\mathrm{dom}(R) = D_1 \times \ldots \times D_n = \{<d_1,\ldots,d_n> | d_k \in D_k\}$
= set of all permissible tuples

$D_R = D_1 \cup \ldots \cup D_n$
= set of all possible objects in any attribute in R

$\mathrm{adom}(A_k, r) = \{x | x \in D_k$ and x in some tuple of r$\}$

FIGURE 20-4
Formal components of a relational schema.

that column (its ***domain*** $\mathbf{D_k}$). In nearly all cases these domain sets are limited to simple constants such as numbers, booleans, character strings, etc. No complex structures are permitted. The total number of defined attributes is the ***degree*** of the schema.

The domain of the entire schema, $\mathbf{D_R}$, is the union of the individual attribute domains. It represents the set of all possible values out of which any component of any tuple might come. A variation of this, **dom(R)**, represents the Cartesian product of all these domains, and thus is the set of all possible tuples from which the tuples of an actual relation might be drawn.

A particular relation **r** is an instance of a schema; it is a subset of the set **dom(R)**. It is often written as **r(R)**, or $\mathbf{r}(\mathbf{A}_1,\ldots,\mathbf{A}_n)$ to show the schema used.

The ***active domain*** of an attribute for a particular relation, $\mathbf{adom(A_k,r)}$, is the subset of $\mathbf{D_k}$ that is actually used by the tuples in **r.** This changes as a function of time as tuples are added to, deleted from, or modified in the relation.

Another piece of notation treats a tuple *t* as almost a function symbol, where the only argument it can take is an attribute name from the schema governing the tuple's makeup. When treated as an expression, the value returned is the value of the component with the specified attribute name. Thus, if *t* is the tuple (Math, 390R, 6pm, 308) from the table of Figure 20-3, *t*(**Number**) returns "390R," while *t*(**Dept**) returns "Math."

Finally, a complete database usually has several schemas associated with it, one for each unique kind of table in it. There is nothing, however, that prevents a single schema from being used to describe more than one relation in the same or different databases.

20.2.2 Keys

In a relational model a ***key*** is some minimal subset of attributes (columns) of a table such that a tuple (row) can be uniquely identified just from its key attribute values. This means that, given some key values, scanning the key columns for a match will locate at most one tuple with these values. As before, knowing which attributes are key permit an implementation to "invisibly" (from the user's perspective) keep lookup tables or the like to speed access to individual rows when a key value is presented as part of a query.

The key columns are a programmer-specified property of the schema, not an individual relation built from that property. Thus, once a key is identified, it need never be rechecked for uniqueness as new tuples are added or deleted from a relation.

There is nothing to prevent more than one set of attributes that act like keys in a schema. In particular, a ***superkey*** is a set of attributes that includes some key as a subset. Such a key maintains information that is not needed for uniqueness identification. A ***designated key,*** or ***primary key,*** is a minimal attribute set that the schema designer has designated as the one to use first in indexing operations. Other keys which are also minimal (they are not supersets of the primary key) are called ***implicit keys.*** Finally, a ***foreign key*** of a schema is a key (not necessarily primary) some of whose attributes are some of the primary key attributes for another database. Knowing values for such attributes helps unlock corresponding tuples in several different relations.

20.2.3 Basic Operations

There are two general kinds of functions in the relational model: those that perform relatively simple operations on one or more relations, and those that perform operations whose time complexity can grow as rapidly as the product of the number of tuples in the relations provided as arguments. This section describes the former, which itself can be split into three categories:

- Functions that effect changes on the tuples in the relation (side effects)
- Functions that generate a new relation from the arguments, but with the same schema
- Functions that generate a new relation from the arguments, and with a new schema

Figure 20-5 lists the major functions of the first kind. In all cases the arguments consist of the name of a relation and either a tuple or part of a tuple. The notation "**A=v**" is called an ***attribute-value pair,*** and means that for whatever tuple is processed, the component corresponding to the attribute named **A** should have a value **v.** For ***insert,*** all the attributes making up a tuple are named, with their corresponding values making up the new tuple to be added to the relation. For ***delete*** and ***modify,*** one set of attributes (those marked $\mathbf{K_i}$ in the figure) form a ***pattern*** which should identify a particular tuple. The attributes in this pattern should encompass a key set, and the values thus select a unique tuple. The **delete** function will remove this tuple from the relation. The **modify** function selects the tuple and then uses the second set of attribute-value pairs to modify some of its components.

insert(r, $A_1=d_1,\ldots,A_n=d_n$) = Add tuple $<d_1,\ldots,d_n>$ to relation r.

delete(r, $K_1=d_1,\ldots,K_k=d_n$) = Delete tuple with designated key from relation r.

modify(r, $K_1=d_1,\ldots,K_k=d_n$; $C_1=c_1,\ldots,C_m=c_m$)
= Find tuple with designated key,
and change attributes $C_1,\ldots,C_m$ to matching values.

FIGURE 20-5
Basic state-change operations.

In all these cases the time complexity is at worst proportional to the number of tuples in the relation (all tuples are searched one at a time). In good implementations this might drop to the log of the number of tokens, or less.

The second class of simple operators (Figure 20-6) corresponds directly to standard functions over sets. One or two relations are accepted as input, and a new relation is generated (no side effects) where every tuple came from one of the inputs and thus has the same schema as the input relations.

Finally, Figure 20-7 diagrams three functions that accept relations of one schema as input and produce relations (often of another schema) as a result. Figure 20-8 diagrams some simple examples.

The first of these, ***rename*** (denoted "δ"), takes a relation of one schema and produces one over the other schema where the only difference is in the name of an attribute. The tuples in the relation itself are untouched copies. A subscript on δ of the form "A←B" indicates that the attribute with the name "A" in the original schema is called "B" in the resulting schema. Again note that the notation *t*(**B**) refers to the component of the tuple bound to *t* that has attribute name **A.**

The second function, ***select*** ("σ"), with subscript **A=a,** takes a relation **r** and returns a subset of that relation, over the same schema, which includes all those tuples whose components in the **A** attribute column have the value **a.** This is a lot like setting up a query in PROLOG where all the arguments but the one corresponding to **A** are variables, and **A** has an **a** in it. The function **select** has several useful properties:

- It commutes under composition with itself; that is, $\boldsymbol{\sigma_{A=a}(\sigma_{B=b}(r))=\sigma_{B=b}(\sigma_{A=a}(r))}$.
- Shorthand notation for compositions like that above collapse all the patterns into a single subscript, as in $\boldsymbol{\sigma_{A=a}(\sigma_{B=b}(r))=\sigma_{A=a,B=b}(r)}$.
- It distributes over **union, intersection,** and **update**; for example, $\boldsymbol{\sigma_{A=a}(r \cup s)=\sigma_{A=a}(r) \cup \sigma_{A=a}(s)}$.

The final function, ***project*** ("π"), takes a relation **r** over some schema **R** and a set of attributes **A** and, for each tuple in **r,** removes all but those attributes. This is equivalent to deleting all but a set of columns of a table. The schema of the resulting relation becomes a subset of the schema for the original relation. Note also that it is possible to get duplicate tuples out of this process, particularly when not all the key attributes are included in **A.** Most definitions of π have the function remove these duplicates. Those that don't, produce ***multisets,*** not relations.

Let r, s be relations on same schema, t any tuple:

Intersection: $r \cap s = \{t | t \in r \text{ and } t \in s\}$

Union: $r \cup s = \{t | t \in r \text{ or } t \in s\}$

Difference: $r - s = \{t | t \in r \text{ but not in } s\}$

Complement: $-r = \{t | t \in \text{dom}(r), \ t \text{ not in } r\}$

Active Complement: $\neg r = \{t | t \in \text{adom}(r), \ t \text{ not in } r\}$

FIGURE 20-6
Schema-preserving operations.

Rename: $\delta_{A \leftarrow B}(r) = \{t | \text{for some } t_1 \in r, \ t(B) = t_1(A)$
$\text{and } t(R-A) = t_r(R-A)\}$

Select: $\sigma_{A=a}(r) = \{t | t \in r \text{ and } t(A) = a\}$

Project: $\pi_A(r) = \{t | t_1 \in r \text{ and } t = t_1(A)\}$

Note: r is a relation over a schema R, which in turn is denoted as a set of its attribute names $\{A_1, \ldots, A_n\}$.

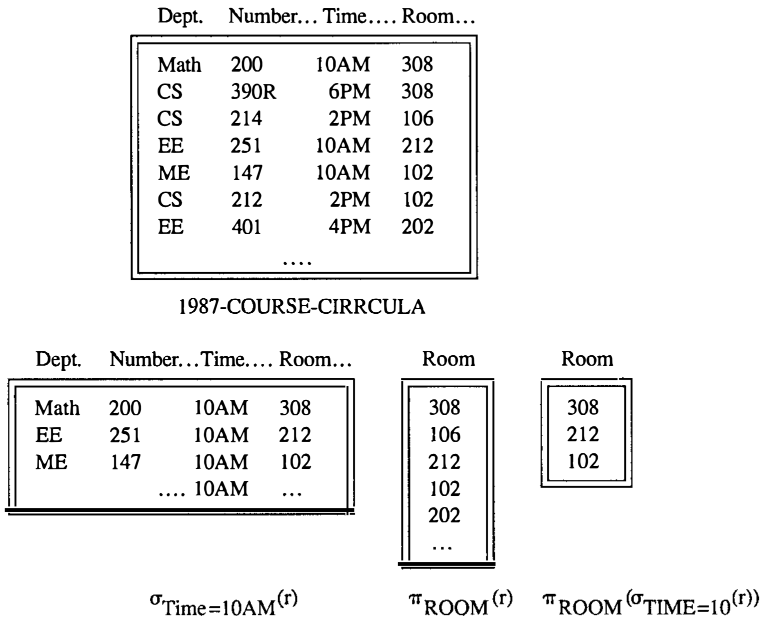
FIGURE 20-7
Schema-changing operations.

Simplifications of compositions of projections are possible, particularly when the outer projections are over subsets of columns of inner projections. In particular, if a set of attributes **B** is a subset of another attribute set **A,** then $\boldsymbol{\pi_B(\pi_A(r))=\pi_B(A)}$.

Finally, a very useful and common query is a projection of a selection. We select a subset of tuples from a relation where certain components equal some value and then create a new relation from a subset of its columns. In terms of Figure 20-8, $\boldsymbol{\pi_{Room}(\sigma_{Time=10am}(}$**1989-course-curricula**)) equals a relation we might entitle "**Rooms-in-use-at-10am.**"

20.3 COMPUTATIONALLY EXPENSIVE OPERATORS: JOINS

The operators of the last section of themselves provide a useful, but limited, array of database query and construction capabilities. There is also a rather wide range of straightforward implementation alternatives for them. In contrast, variations of the ***join*** operation give the relational model both its real expressive power and its implementation complexity. Basically, a **join** accepts two relations and "joins" them along columns whose attribute names and domains are common to both. This operation takes each tuple from one table and compares the values in some subset of columns common to the two relations to those columns in all the tuples of the other table. For each match a new tuple is generated for the result that consists of the concatenation of the two tuples, with the duplicate columns displayed exactly once. A single tuple in each relation can join with none, one, more than one, or even all of the tuples in the other relation. The result is a new relation with a schema that looks like the union of the two argument schemas. Figure 20-9 gives a simple example.



Dept.	Number...	Time....	Room...
Math	200	10AM	308
CS	390R	6PM	308
CS	214	2PM	106
EE	251	10AM	212
ME	147	10AM	102
CS	212	2PM	102
EE	401	4PM	202
....			

1987-COURSE-CIRRCULA

Dept.	Number...	Time....	Room...
Math	200	10AM	308
EE	251	10AM	212
ME	147	10AM	102
	....	10AM	...

$\sigma_{\text{Time}=10\text{AM}}(r)$

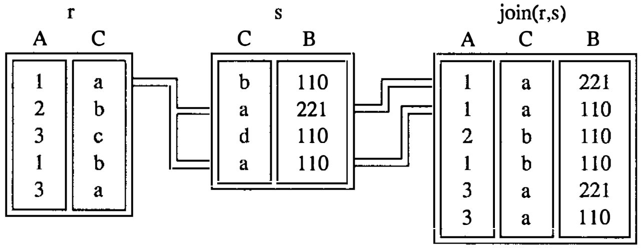
Room
308
106
212
102
202
...

$\pi_{\text{ROOM}}(r)$

Room
308
212
102

$\pi_{\text{ROOM}}(\sigma_{\text{TIME}=10}(r))$

FIGURE 20-8
Tabular view of **select** and **project.**



r

A	C
1	a
2	b
3	c
1	b
3	a

s

C	B
b	110
a	221
d	110
a	110

join(r,s)

A	C	B
1	a	221
1	a	110
2	b	110
1	b	110
3	a	221
3	a	110

FIGURE 20-9
Sample join.

20.3.1 Natural Joins

There are several kinds of joins, varying primarily in how the set of columns to match on are chosen. The most basic of these, the ***natural join,*** selects all possible columns that are shared between the two schemas. Notationally, the natural join of two relations **r** and **s** (with schemas **R** and **S**) is written **r**$\bowtie$**s.** If **C** is **R** ∩ **S** (the common attributes), the result of the join is:

$$\mathbf{r}\bowtie\mathbf{s}=\{t|t(\mathbf{R})=t_r\in\mathbf{r},\ t(\mathbf{S})=t_s\in\mathbf{s},\text{ and }t_r(\mathbf{C})=t_s(\mathbf{C})\}$$

If the degree of **r** is d_r, the degree of s is d_s, and they have c columns in common, then the degree of the result of the join is d_r+d_s-c. If there are n_r tuples in **r** and n_s tuples in **s,** then there are at most $n_r \times n_s$ tuples in the join. The maximum number corresponds to the situation when each tuple of one joins with all tuples of the other. It is also possible for no tuples to match, yielding an empty relation. Usually the result is somewhere in between. Note that, by definition, if there are no common columns between two relations, each tuple joins with each tuple in the other, and the result of the join is identical to the ***Cartesian product*** of the two relations.

The natural join has several nice mathematical properties. It is both associative and commutative, which means that complex queries that perform several natural joins in some nested fashion can be rearranged almost arbitrarily so that total computation time can be minimized. Further, any **select** of the form $\sigma_{A=a,B=b,C=c\ldots}(\mathbf{r})$ can be duplicated exactly by a join between **r** and the ***constant relation*** {(a,b,c)} with schema {A,B,C,...}.

Finally, because of its importance, we repeat here the mental ***tuple-at-a-time*** model for performing a natural join:

1. Identify all common attributes with the same name and domains.
2. Take some tuple *t* from one relation **r.**
3. Look at each tuple *u* in the other relation **s.**
4. If the two tuples have the same values in all common attribute positions, add a new tuple to the result relation consisting of the concatenation of the two tuples *t* and *u,* minus one copy of the common attributes from *u.*
5. Repeat steps 3 and 4 until all of s has been checked.
6. Repeat steps 2 through 5 until all of **r** has been processed.

20.3.2 Equijoin

An ***equijoin*** is a join where the columns along which the match is to be performed are specified in advance. If two relations **r** and **s** have schemas **R** and **S,** with $\{\mathbf{A}_1,\ldots,\mathbf{A}_m\}$ a subset of **R,** $\{\mathbf{B}_1,\ldots,\mathbf{B}_m\}$ a subset of **S,** and matching domains $\mathbf{dom}(\mathbf{A}_k)=\mathbf{dom}(\mathbf{B}_k)$, then the notation $\mathbf{r}[\mathbf{A}_1=\mathbf{B}_1,\ldots,\mathbf{A}_k=\mathbf{B}_k]\mathbf{s}$ joins the tuples in the two relations that have the same values in the indicated columns. We assume that there are no common attribute names across the the two schemas to begin with.

The main difference from the natural join is that both matching columns are kept in the tuples, and thus the schema of the output is the sum of the two schemas. Again, if m is 0 (we join across no attributes), we get the Cartesian product of the two relations again.

In some sense the equijoin is more fundamental than the natural join. Applying a **rename** to one relation before an equijoin guarantees unique attributes. Performing an equijoin on the columns that were just renamed computes the same tuples as the natural join would have.

Applying a **project** to the result can then eliminate the duplicate columns.

20.3.3 Related Operations

All of the joins are "multiplicative" functions; their result is related directly to the "product" (in the Cartesian sense) of the two inputs. If one were building an algebra, then, given a multiply operator, a good mathematician would look around for interesting inverse "divides." Such operations exist for the join, and are summarized in Figure 20-10.

The first of these, the ***divide*** operation **r** ÷ **s**, takes two relations **r** and **s** and looks for the largest subset **q** of **r** that, if joined with **s**, would produce a relation that is still a subset of **r**. This is analogous to a divide over integers, where the result **q** of **r** ÷ **s** has the property that it is the largest integer such that $\mathbf{q}\times\mathbf{s} \le \mathbf{r}$.

Actually, the schema for **q** is not all of **R**, but actually **R**−**S**. This reflects what the join with **s** will add back in. The relation **q** is thus a set of tuples from the schema **R**−**S**, where joining any tuple from it with any tuple from **s** yields a tuple from **r**.

In terms of a tabular interpretation, **r** ÷ **s** looks like this:

1. Cross out all the columns of the table **r** that have the same name and domain as **s**.
2. Toss out duplicate rows.
3. Pick the biggest subset of the result such that a join with **s** is still in **r**.

Note that this join has no common attributes and thus is a simple cross-product.

Assumption: r(R), s(S)

Cross Product: $r \times s = \{t|\ t(R)\in r \text{ and } t(S)\in s\}$

Natural Join: $r \bowtie s = \{t|t\in \text{dom}(R\cup S),\ t(R)\in r,\ t(S)\in s\}$

Equijoin: $r[A_1 = B_1,\ldots,A,_m = B_m]s = \{t|t(R)\in r,\ t(S)\in s,\ t(A_k) = t(B_k) \text{ for } 1\le k\le m\}$

Split: $\text{split}(r, p) = \{q_T,q_F\}$ where
  $q_T = \{t|t\in r \text{ and } p(t)\}$
  $q_F = \{t|t\in r \text{ and } \neg p(t)\}$

Factor: $\text{Factor}(r,A, L) = \{q(A\cup L),\ u((R-A)\cup L)\}$ where
  L is an attribute not in R or S
  $\pi_R(q\bowtie u) = r$

Division: $r\div s = \max\{q|q\in \text{dom}(R-S) \text{ and for all } t\subseteq S,\ q\bowtie t\in r\}$

FIGURE 20-10
Complex operations on relations.

Another complex operation is ***split.*** This function has two arguments, a relation **r** and a predicate function **p** over tuples in **r**, and returns two relations as a result. One of these consists of all tuples that satisfy **p**; the other is all other tuples from **r**.

The final complex operation in Figure 20-10 is ***factor.*** Its arguments consist of a relation **r** over a schema **R**, a set of attributes $\mathbf{A}=\{\mathbf{A}_1,\ldots,\mathbf{A}_m\}$ from **R**, and the name **L** for a new attribute that is not found in **R**. The result is two new relations, one with schema **A** ∪ **L** and the other with schema (**R**−**A**) ∪ **L**. Each tuple in **r** is split in two along these lines, with a unique "index" generated for the **L** component to link them. The basic operations proceed like this:

1. Project **r** along **A** and **R**−**A**.
2. Remove duplicates from the **A** projection only.
3. Add a new column **L** to each.
4. Generate a unique index value for each **L** component in the first.
5. Fill in the appropriate index value in the second that links together tuples that came from the same tuple of **r**.

The result of a join of the two outputs (minus the **L** column) should be the original **r**.

20.3.4 Comparisons Other Than Equality

The descriptions of all the above operations have assumed equality as the basic pattern match. In real life other predicates are useful ($<, \le, \ldots$). To handle this we say that two attributes **A** and **B** are **θ** ***comparable*** (read ***theta comparable***) if **θ** is a well-defined predicate over **dom(A)** and **dom(B)**. Then, without any loss in accuracy, we can define ***θselect, θjoin,*** etc., by replacing the equality test by the predicate **θ**.

20.4 RELATIONAL ALGEBRA

Codd's key contribution to database theory was the recognition that the above operations could be combined into a ***relational algebra*** that is both a firm algebra in a mathematical sense and a basis of a notation for formally posing queries of quite sophisticated complexity.

The algebra itself is based on a set of seven objects **{U,D,dom,RS,d,θ,O}**, where

- **U** is the set of all possible attributes.
- **D** is the set of all possible domain sets for attribute values.
- **dom:U→D** is a function between attribute names of **U** and domain sets from **D**.
- **RS** is the set of all possible relation schemas, namely, the set of all pos-

sible subsets of **U**, where each subset equals some set of attribute names.

- **d** is the set of all possible relations over the set of schemas **RS**, that is, all possible tables whose attribute names come from **U** and whose tuple values come from the appropriate elements of **D**.
- **θ** is the set of all valid comparison predicates, including at least "=" and " ≠ ."
- **O** is the set of operators from which expressions in the algebra may be formed.

Notice the strong relationship to an *interpretation* and the *Herbrand Universe*.

The set of operators **O** is carefully chosen so that they in fact form an algebra over the associated domains (they are closed, have inverses, identities, etc.). The minimal set of operators to make a system relational include:

- ***Union, intersection, difference***
- ***Active-complement, rename***
- ***Project, natural-join, divide***
- ***Select*** and ***θjoin*** with predicates from **θ**
- The standard logical connectives ***and, or,***... within the θjoin

Note that some other functions, such as **fact** and **split**, are not required for this minimal algebra.

A ***relational expression*** is any legal composition of these operators as applied to relations from the algebra or from constant relations from the domains. All such expressions produce single relation outputs.

20.5 IMPLEMENTATIONS

(Kitsuregawa and Tanaka, 1988)

This section describes several implementations of relational systems, both from the language and the machine side. In general, the variations reflect the different viewpoints taken toward storing and accessing tuples from relations. The simplest of these is the ***tuple-at-a-time*** view, which approaches each table as a file of records, one record per tuple, which can only be read sequentially from the beginning, one record at a time, and is extended by adding to the end of the file. This is a very simple model, with a good match to the user's perceptions and fast inserts, but slow selects (time proportional to length of file) and extremely slow joins (time proportional to the product of the lengths of the two files).

The next approach is a variation of this that keeps the records of each file in a sorted order, where the sort is along some attribute that is frequently used in expressions. This slows down inserts, since we must rewrite the entire file. However, it speeds up joins tremendously. The join becomes essentially a ***merge*** of the two files, with total time proportional to the sum of the two files lengths. For large files this can be an enormous savings. The sort speeds nothing, however, when other attributes are involved.

Another approach is to keep the files in random stored order but to maintain separate index arrays for one or more of the attributes. Speed of inserts is slower than the simple method (we have to rewrite each index file) but probably faster than the sorted approach (we do not erase the whole file). Speeds for selects and joins are almost as fast as for the sorted order, with the difference being in the need to read and rewrite the index arrays. Again, for nonindexed attributes, no speed gain is achieved.

Finally, files of tuples could be kept in random stored order but with ***hash tables*** available. A ***hash*** is a set of bins, where each ***bin*** corresponds to taking one or more attribute values through some ***hashing function*** that condenses the values to a few bits and then links together a list of all tuples which hash to the same value. This is similar to the ***indexing instructions*** on the WAM, and it speeds up selects and joins on multiple attributes when those attributes at least intersect the attributes involved with hash tables. Although the speedup is not as great as for the fully indexed case, the inserts are not slowed down as much.

Database problems seem to be ideal for the use of extensive parallelism. The databases themselves can be distributed over multiple media to increase access bandwidth. Multiple processors can likewise divide up the query processing, particularly the joins, where the time to process a section of a database can dramatically exceed the time to read it from storage. Several such machines will be described here. Note, however, that parallelism by itself is not always the best answer. Stone (1987), for example, gives a case where a good query-processing algorithm on a sequential machine can exceed the performance of a massively parallel machine.

20.5.1 Implementation in PROLOG

(Li, 1984; Yang, 1986, Section 8.5)

As mentioned in the introduction to this chapter, there is a strong resemblance between expressions in relational algebra and PROLOG statements. The individual relations are equivalent to sets of PROLOG facts. Individual tuples of the form ($\mathbf{a}_1$,...,$\mathbf{a}_n$) for relation **r** transform into a PROLOG statement of the form **r**($\mathbf{a}_1$,...,$\mathbf{a}_n$). Each attribute corresponds to a position in the PROLOG predicate's argument. Inserts and deletes are equivalent to **assert**s and **retract**s, respectively.

In classical PROLOG, control over the order of insertion is usually limited to either the top or bottom of the set of clauses that imple.nent the relation, so an underlying sorted order implementation is usually out. However, one of the basic WAM instructions is a hashed index that takes an argument value, hashes it, and branches through a table according to that value. Thus if we are designing a PROLOG to handle database relations more directly, we may wish to keep the hashing but replace the unique WAM code generated for such clauses by a more symbolic structure which is then searched by a general unification routine. A goal with the name of a database relation as its predicate would then invoke this general routine to branch through the hash table and search down a list of linked facts, using a general matching routine on each one. Ramamohanarao and Shepherd (1986) describe just such a system, in which a ***superimposed codeword*** scheme generates the hash code; Wong and Williams (1989) describe a hardware engine that handles this. Several popular implementations of PROLOG available today for personal computers and workstations implement something very similar in software.

A query to produce a new database relation **q** would convert into a PROLOG clause in which **q** is the head, its arguments are a list of variables, one per attribute, and the query expression is converted into a PROLOG left-hand side. This conversion might proceed as follows:

- For a selection of the form $\boldsymbol{\sigma}_{A=a}(\mathbf{r})$, where **A** is the k-th attribute, form a clause of the form:
  $\mathbf{q}(x_1,\ldots,x_{k\text{-}1},\ \mathbf{a},\ x_{k+1},\ldots,x_n){:}-\mathbf{r}(x_1,\ldots,x_{k\text{-}1},\ \mathbf{a},\ x_{k+1},\ldots,x_n)$.
- For a projection $\boldsymbol{\pi}_A(\mathbf{r})$ where **A** is the k-th attribute, form a clause of the form:
  $\mathbf{q}(x_k){:}-\mathbf{r}(x_1,\ldots,x_n)$.
- For an equijoin (along the k-th component in **r** and the j-th in **s**):
  $\mathbf{q}(x_1,\ldots,x_n,\ y_1,\ldots,y_{j\text{-}1},\ y_{j+1},\ldots,y_m){:}-\mathbf{r}(x_1,\ldots,x_n),\ \mathbf{s}(y_1,\ldots,y_{j\text{-}1},\ x_k,\ y_{j+1},\ldots,x_m)$.

Other operations would convert similarly.

With this approach, complex relational expressions usually subsume into single PROLOG clauses, with selects indicated by constants in arguments, projects simply ignoring argument positions, and joins proceeding via shared variables.

Processing in such PROLOG programs does not take the route of computing the entire relation at once. Instead, each time a PROLOG goal of the form **q**(...) is executed, the converted clauses are computed until a set of argument values is computed. If a backtrack is executed into the **q** goal, the next such set of argument values is computed. The process repeats until all tuples of **q** have been computed. This is essentially "tuple at a time."

This mechanism could be embedded in a **set-of** or **bag-of** predicate to accumulate all the components of a new relation and enter them into a hashed relation as described above.

20.5.2 Structured Query Language

(Blasgen, 1981; Astrahan et al., 1979; Ullman, 1982, chap. 6)

Structured Query Language (SQL) is an outgrowth of a research project (***System R***) at IBM in the mid-1970s to implement Codd's ideas in a practical database system. In structure, SQL is a combined data definition, data manipulation, and data query language with a very close tie to the underlying relational algebra. It has grown in popularity to the point where today many database programs are said to be "SQL-like."

Figure 20-11 diagrams a simple subset of syntax for SQL. The major statement form of interest here is the ⟨query⟩, which takes the form:

SELECT ⟨***attribute-list***⟩ FROM ⟨***relation-sets***⟩ {WHERE ⟨***condition***⟩}

The statement's semantic model is very close to the "tuple-at-a-time" model but may, invisibly to the user, support any of the other implementation techniques. Basically, a programmer assumes that from each relation mentioned in the FROM expression ⟨relation-sets}, each tuple (or set of tuples) is selected for processing one at a time. The WHERE condition then specifies the conditions under which the chosen tuple will participate in the final relation. At its simplest this can be of the form of a select test ⟨attribute⟩=⟨constant⟩ or a join of the form ⟨attribute⟩=⟨attribute⟩. Further, a wide range of predicates can be substituted for the equality in the condition test.

Finally, a projection can be performed on the result by specifying only certain attributes for output in the ⟨attribute-list⟩ following the SELECT keyword. Duplicate output tuples are not eliminated unless the keyword UNIQUE is part of the ⟨***attribute-list***⟩. Other options for this column list include expressions over columns involving ***aggregate operators*** such as **avg** (average), **max, min, sum,** and **count.** As an example, **avg(grade)** returns the average of all the **grade** attributes of the tuples that survived the WHERE condition.

```
<query> := SELECT <attribute-list>
           FROM <relation-sets>
           {WHERE <condition>}

<attribute-list> := {UNIQUE} <attribute> {,<attribute>}*
                    | <aggregate-function>(<attribute>)
                    | "*"

<attribute> := {<relation-name>.}<attribute-name>

<relation-sets> := <relation-name> {,<relation-name>}* <variable>*

<condition> := <value><predicate><value>
               | <value> IN <query>

<value> := <attribute> | <variable> | <constant>

<assignment> := ASSIGN TO <relation-name>: <query>
```

FIGURE 20-11
Simplified query syntax of a SQL subset.

In any of these expressions, if there are multiple relations with the same attribute name, the appropriate one can be specified by preceding the attribute name with the name of the desired relation and a ".". Thus if the attribute name "**id**" was used in both the **student** and **instructor** relations, and both relations were used in some query, then **student.id** or **instructor.id** would be used as appropriate.

A wide variety of complex test conditions can be specified in the WHERE condition. Simple tests, such as **instructor**="Kogge," are equivalent to selections in the relational algebra. Other tests, such as **instructor.id**=**student.id**, form joins (in this case the equijoin between the **id** attributes of the relations **instructor** and **student**—all students who are also instructors). Membership in a set can be tested by an IN expression, typically of the form ⟨***attribute***⟩ IN ⟨***query***⟩, where the ⟨***query***⟩ computes a relation of one-element tuples whose values are permissible. Leading NOTs may be used to indicate that a tuple passes only if it fails some test. Multiple tests can joined by AND, OR, etc., with the expected meanings.

Finally, new relations can be named and saved via ***assignment statements,*** which capture the output of a query. The schema of the new relation is inherited from the attributes in the ⟨***attribute-list***⟩.

The reader should note the highly functional flavor of the syntax of these expressions combined with their logic-oriented semantics.

20.5.3 QBE

(Zloof, 1977; Ullman, 1982, chap. 6)

One of the most productive advances in personal computing was the introduction of the ***spreadsheet***—a table into whose squares could be placed constants and formulas involving the values of other squares. Calculating the table involves displaying for each square its fully evaluated contents.

The same phenomenon happened with relational databases with the introduction of ***Query By Example*** or ***QBE.*** This too presents the user with a visual spreadsheet, called a ***table skeleton,*** into which the user can place expressions—in this case derived from the relational model. During a session a user can open and operate on many such skeletons. Figure 20-12 diagrams a blank table.

In the upper left square of a table skeleton the user places the name of some relation. If the relation is known to the system, then that table will reflect the contents of that relation. If the relation name is new, then that table will be built as a new one.

Across the rest of the top row go the names of attributes. If the relation is the name of an existing one, placing the attribute names here performs the equivalent of a **project.** For a new relation these names define the relation's schema.

Below the attribute names go rows of tuple descriptors. Within any row, placing a constant is equivalent to generating a **select** query, which only selects tuples from the original relation whose attribute-value pairs mirror those of the row tuple. Placing a partial expression such as ">3" in an entry passes only tuples whose values in that column are greater than 3. Placing a ***variable name*** (an underlined character string) in a row entry is equivalent to putting a variable in the right-hand side of a PROLOG clause. All tuples which pass all the other select commands in that row, and which also have values in that attribute position consistent with all other bindings of the same variable in other rows or tables, are still acceptable to that row. This is equivalent to performing an ***equijoin.***

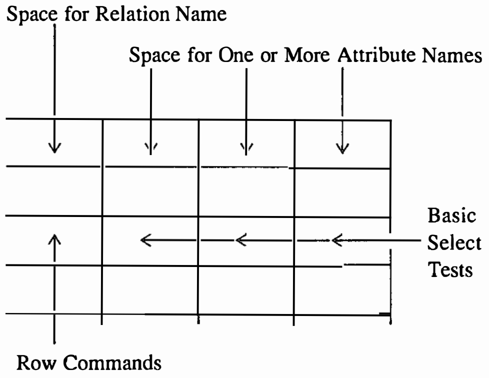


FIGURE 20-12
QBE table skeleton.

A ***row command*** may be entered in the far left column under the relation name. Such a command is applied to all tuples that pass the row tests. These commands are usually single letters that end with a ".". They include "D." for **delete,** "P." for **print,** and "I." for **insert** (used for new relations. The **print** command is equivalent to the "recalculate" command in a spreadsheet; it causes that row to be replaced by copies of all the rows that satisfy the row tests.

A "P." may also be placed in a row entry other than the first column. This causes just the values for that attribute from all tuples that pass the other tests to be displayed on the screen.

Finally, a ***condition box*** can be called up by a user to augment any query. In this box the user can place predicate expressions that must be satisfied for tuples to pass, but that are too awkward to build normally. An example might be "*income*+0.4×*capital-gains*>$10000" and involve arbitrary numbers of variables in arbitrary expressions.

20.5.4 The Teradata DataBase Computer

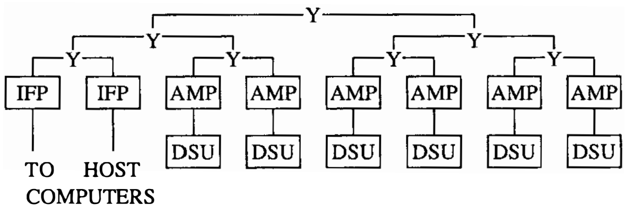
(Neches, 1984, 1986; Shemer and Neches, 1984)

The ***Teradata DBC/1012*** is a specialized computer system optimized to perform relational queries in parallel. Its design is based on the observation that a centralized point of access to a database makes sense for many applications, but there is no need to centralize the actual execution of queries. The primary goal of the design was to use many cheap microprocessors and disk drives in a parallel configuration to achieve very high performance and cost/performance. Additional goals were to permit incremental additions of performance power, coupled with increased fault tolerance. Versions have been in use since 1984, with a recent design employing anywhere from 6 to 1024 Intel 80286-based microprocessors.

The processor is connected as a back-end processor to a conventional host system. Queries to databases stored in the DBC/1012 are expressed in an SQL-like language, and come from the user through the host. These queries then go through syntax analysis and interpretation, with the results being shipped back to the host for relay to the user. An English-based natural-language front end is also available.

Figure 20-13 diagrams the architecture of the machine. It consists of two kinds of processor nodes, ***Access Module Processors*** (***AMPs***) and ***InterFace Processors*** (***IFPs***). Both have the same processor card but different I/O. There are usually two IFPs per system to communicate between the rest of the processor and the host machine. Each AMP has associated with it up to four local disks and a connection into a treelike interprocessor communication network called a ***Ynet***. The system tries to divide relations into equal-sized sets of tuples, and to distribute those sets across these disks.

The protocol of the Ynet permits guaranteed delivery of messages from any one AMP to some other specific AMP, a subset of AMPs, those in some predefined class, or even all of them. This guaranteed-delivery feature permits reliable implementation of both joins (where one part of a relation has to be distributed to all other parts) and synchronization operations. The basic process involves a byte-at-a-time transfer of information into the local Ynet connection. As the message goes up the tree at each node, it contends with traffic coming up from other AMPs. One message wins and continues; the other loses and drops off for retry. When the winning message reaches the root, it goes back down the tree in "broadcast" mode, splitting at each node. Individual responses come back up similarly, and are combined into a positive indication of what happened to the message.



AMP = Access Module Processor
DSU = Disk Storage Unit
IFP = Interface Processor
Y = Ynet Interconnection Node

FIGURE 20-13
Simplified teradata parallel computer.

20.5.5 A Specialized Relational Database Engine
(Kamiya et al., 1985)

As mentioned in the introduction to this section, the major problem with relational databases is the time complexity of the join, O(m×n) where m and n are the lengths of the files being joined. This can be reduced to O(m+n) *if* (a big if) the files happen to be sorted along the attribute being joined. The ***Relational DataBase Machine*** (***RDBE***) is an example of a machine designed to guarantee this. This machine was built as part of the Japanese ***Fifth-Generation Project*** as part of the larger ***DELTA*** knowledge-based machine prototype.

Figure 20-14 diagrams the basic architecture of this system. As with the Teradata machine, the RDBE is a back-end accelerator to a host, in this case a PROLOG-oriented ***Sequential Inference Machine.*** Queries from a PROLOG program for database access are routed to the RDBE for pro-

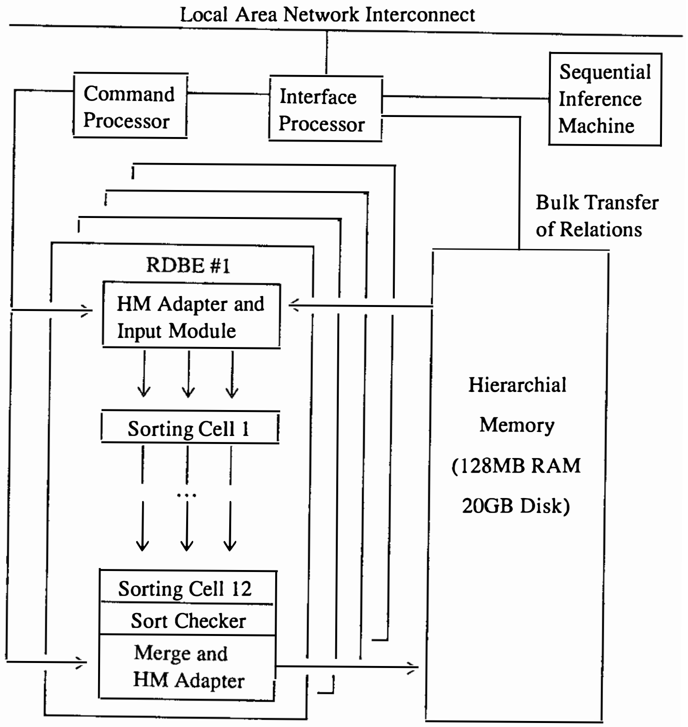

Local Area Network Interconnect
Command Processor
Interface Processor
Sequential Inference Machine
Bulk Transfer of Relations
RDBE #1
HM Adapter and Input Module
Sorting Cell 1
...
Sorting Cell 12
Sort Checker
Merge and HM Adapter
Hierarchial Memory (128MB RAM 20GB Disk)

FIGURE 20-14
Principal components of DELTA Machine.

cessing, with the results returned. Again as before, the RDBE has associated with it a large disk and storage array for containing the relations.

The basic hardware of the machine is essentially a set of multistage high-speed sorting networks capable of sorting tuples based on some attribute field, which may be integers, floating-point, or character strings. For simplicity of design, all tuples must be exactly 2048 words long as they are stored in the mass memory. Which attribute is sorted on is totally programmable.

Each of the sorting networks consists of an input module, 12 stages of sorting cells, and a final sort checker, merge, and output module. All told, a sorting module can sort up to 4096 tuples in each pass. These tuples are brought into the input module, where the attribute to be sorted on is used to select a subfield of each of them. From there, the 4096 values are compared two at a time in each layer of sorting cell. After the first layer of cells, every pair of tuples is sorted. After the second layer, every group of four cells is sorted. This increases by a factor of 2 at each level until all 4096 are sorted.

The basic relational query operations are handled by the merger unit. For a **select,** a sort on the attribute to be selected is followed by a discard of all elements that do not match (with a sorted order we know exactly when there are no more tuples that do not match). A **project** sorts on the attributes that will remain after the projection, with the merger discarding all but those attribute values. The reason for the sort is to allow duplicate tuples to be removed.

The real performance advantage comes with the implementation of **join.** The first relation is sorted on its join attribute and stored internally where the merger can access it. The second relation is then sorted on its join attribute. As these tuples leave the sorter, they are compared one at a time with the first tuple of the other sorted stream. Whichever one is too small (assuming that the predicate of the join is equality) is discarded. As long as both top tuples are equal, a joined tuple is generated and sent back to the memory. When they are no longer equal, the smaller one is discarded and the process is repeated.

Many selections and projections can be cascaded into this join process at no cost.

Relations larger than 4096 tuples are handled by multiple passes.

20.5.6 Datalog

As rich as the relational model is, it is not the end of the line for database systems. Just as PROLOG designers are working toward including support for databases in PROLOG, database designers have been looking at including many PROLOG features such as recursion and more powerful unification in their future systems. In fact, a common term for the next-generation database systems is ***datalog***—a PROLOG without function symbols, but with the completeness needed by databases. This latter comment refers to PROLOG's fixed order of attempting clauses, where if a solution to a goal exists by using clause k, but clause k-1 unifies with the goal and happens to go into an infinite loop, then PROLOG will never see the answer to the goal.

One of the most promising approaches to implementing such systems is called an ***extension table*** (see Dietrich, 1987) and resembles the ***memo functions*** and ***function caching*** described in Chapter 12. Such a facility captures all intermediate results for some goal in a table, and when some other goal occurs with the same predicate name, this table can be interrogated for at least a partial match. Thus, rather than tossing a potential set of bindings away at a backtrack, and then recomputing it at a later time, this approach accumulates the answers for reuse.

This has a direct effect not only on speed but also on completeness. If a particular clause is ***left recursive*** (the leftmost literal is a call to the same predicate—something that is of value in many applications), in PROLOG it would be liable for an infinite loop. With an extension table, the recursion would go through with the prior answers provided. When a new solution is found, it is added to the table, and can in fact participate directly in the recursion again. The completeness of the system has been increased.

In a recent work, Warren (1989) has proposed a new variant of the WAM that implements extension tables directly. This ***XWAM*** is based on the idea of copying environments when a choice point is created rather than reusing previous ones with a trail to handle recovery. New instructions are added to make this copying implement the ***tabling*** of any solutions found into these environments, along with a description of the goal arguments to which they correspond. Then a new instruction searches these tabled environments for ones that either match or are subsumed by the current goal. Next, new instructions also handle "improper failures" by previous goals that might now be retried with new solutions from the extension tables. Finally, instructions are added to support nested evaluations, particularly when predicates like **not** are to be handled.

20.6 PRODUCTION-RULE SYSTEMS

(Forgy and McDermott, 1977; Forgy, 1981, 1984; Brownston et al., 1985)

Over the last 15 years a series of related logic-based languages has been developed at Carnegie-Mellon University by Charles Forgy and his associates. These languages have gone by the name of OPS, OPS-2, OPS-4, OPS-5, and OPS-83, and have proved so useful that they have spawned a whole new class of languages called ***production-rule systems.*** In operation they are a cross between a database system (where all the tuples of the relationlike objects have been explicitly "produced" and kept in storage), and a logic system where clauselike "rules" govern when new tuples are created and old ones are deleted. Further, their major computational step is equivalent to a relational join, with a fast algorithm called a ***Rete net*** used for its implementation. Finally, as with other languages in this book, there is a simple abstract machine (the ***Production-Rule Machine***) that permits compilers to generate efficient code for their execution.

Although the rules have a resemblance to PROLOG, they are supported by an entirely different inference engine (Figure 20-15). First, the inference engine is ***forward-chained,*** with a "goal" of simply making inferences as long as possible. Next, it assumes ***modus ponens*** between facts and rules as its primary inference rule, with a decision procedure which is more breadth-first in nature than PROLOG. At each processing cycle each rule is essentially tested for satisfaction in all possible ways. Out of those rules that have satisfied antecedents, one is chosen for an actual inference, the set of facts is updated, and the process is repeated.

Additionally, whereas in PROLOG rules and facts have the same basic format and can be freely intertwined, in production-rule systems they are radically different in both structure and processing. Data looks like relational tuples, in which attribute values can have more structure that simple constants, while rules maintain their "if-then" structure throughout processing. They are not intermixed with the facts. The predicates used in the antecedents of rules are primarily standard comparators such as "=," ">," etc. Finally, whereas the PROLOG inference engine model is largely sequential, production rules have characteristics that seem suited to at least moderate levels of parallelism.

For simplicity, we will address only the original ***OPS*** language here. Most other production-rule systems are fairly clearly derived from it.

20.6.1 Working Memory Elements

In OPS, "facts" and "rules" are separate entities and have entirely different formats. Facts take the form of more or less conventional data objects, called ***working memory elements*** (written ***wme*** and pronounced "wimmie"), and are stored in the system's ***working memory.*** Rules are stored elsewhere, usually as some sort of compiled code.

Characteristic	OPS	Prolog
Fact	Data object	Single positive literal
Rule	"If-then"	Horn clause
Consequent	Multiple "change object" commands	Single positive literal
Predicates	Largely builtin	Defined by rules
Inference Rule	Modus ponens	Resolution
Decision	Breadth-first	Depth first
Procedure	Forward chained	Backward chained

FIGURE 20-15
Synopsis of differences between OPS and PROLOG.

Each of these wme objects is a member of some ***class*** of similar objects, and consists of one or more ***attribute-value pairs.*** An attribute "names" some characteristic of the object, and the matching value identifies the number, string, vector, or list to be associated with that characteristic. Although an object can have only one pair with the same attribute name, there is no constraint on the number of objects which can have pairs with the same attribute and the same or different values.

This notation is very similar to that of the relational model, with the class name indicating which relation the fact is in, and the attribute-value pairs indicating the tuple's components. It is also similar to the concept of ***records*** in conventional languages like PASCAL, with the OPS object class corresponding to the record type, the attribute names to the record's fields, and the values to the contents of those fields.

An individual object may have any specified attribute-value pair changed or deleted, or new ones added arbitrarily, at any time in the computation.

Other than membership in a particular class and possibly some specific attribute added by the programmer, there is no distinguishing name or key by which individual objects may be referenced. Instead, specific objects are identified during computation by the ***pattern matching*** used in the inferencing.

Figure 20-16 gives some simplified BNF rules for an OPS object. In keeping with early OPS implementations in LISP, the normal representation is as an s-expression list in which the first element is the class name of the object and the following elements are the attribute-value pairs. The attribute name always starts with a "^" and is followed in the list by the matching value.

```
<object> := ( <class-name> {^<attribute> <value>}+ )

<class-name> := <symbol-name>

<attribute> := <symbol-name>

<value> := <number> | <string> | <vector>...

Examples:
  (EXPRESSION ^NAME EXPRESSION-17
              ^LEFT-ARGUMENT EXPRESSION-18
              ^OPERATOR +
              ^RIGHT-ARGUMENT 0)

  (OBJECT ^NAME BLUE-CHEST ^AT SOFA ^ON PILLOW ^WEIGHT LIGHT
        ^CONTENTS BANANAS ^STATUS LOCKED ^KEY RED-KEY)
```

FIGURE 20-16
Simplified syntax of OPS data objects.

The first example in this figure is an object in the class "expression" that has four attribute-value pairs: the "name" of this specific object, its "left argument," its "operator," and its "right argument." The corresponding values indicate that it is "EXPRESSION-17" with overall value "EXPRESSION-18+0." The second example is an object of class "OBJECT" and has attributes dealing with its name, location, weight, contents, and so on.

20.6.2 OPS Rules

In OPS literature, rules usually go by the term ***productions,*** with a syntax (Figure 20-17) that is quite LISP-ish. A rule is expressed as an s-expression whose car is "P" and in which the rest of the list, up to an " ⇒ ," is the rule's antecedent, and the elements following it are the consequent. Further, the antecedent part can have an arbitrary number of literal-like terms called ***conditions,*** which are implicitly joined by **and** connectives. All the conditions in a rule must be satisfied before the rule can be used in an inference.

```
<rule>:= (P <rule-name><antecedent>⇒<consequent>)
<antecedent>:= <condition>+
<condition>:= <pattern> | ¬<pattern>
<pattern>:= (<object-class> {^<attribute> <test>}*)
<test>:= <basic-test> | <conjunction>
<conjunction>:= { <basic-test>+ }
<basic-test>:= <test-value> | <predicate> <test-value>
<predicate>:= = | ≠ | < | > | ≤ | ≥
<test-value>:= <number> | <string> | <variable>
<set-of-values>:= "<<" <constant>+ ">>"
<variable>:= "<" <variable-name> ">"
<consequent>:= <action>+
<action>:= (MAKE <object-class> {^<attribute> <value>}+ )
            |  (MODIFY <pattern-number> {^<attribute> <value>}+ )
            |  (REMOVE <pattern-number> )
            |  (WRITE <value> )

Example: Simplify symbolic expression "<variable> + 0" to <variable>
  (P ADDX0 (GOAL ^TYPE SIMPLIFY ^OBJECT <E>)
           (EXPRESSION ^NAME <E> ^LEFT-ARGUMENT <X>
                                 ^OPERATOR +
                                 ^RIGHT-ARGUMENT 0)
  ⇒ (MODIFY 2 ^LEFT-ARGUMENT <X>
              ^OPERATOR NIL ^RIGHT-ARGUMENT NIL))
     (MAKE GOAL ^TYPE SIMPLIFY ^OBJECT <X>)

Example: From Monkey and Bananas problem:
  (P RULE17 (GOAL ^ACTION HOLD ^OBJECT <OBJECT>)
     (OBJECT ^NAME <OBJECT> ^AT <PLACE> ^ON CEILING)
     (OBJECT ^NAME LADDER ^AT <PLACE>)
     (MONKEY ^AT <PLACE> ^ON LADDER ^HOLDS NOTHING)
⇒    (REMOVE 1 4)
     (MAKE (MONKEY ^AT <PLACE> ^ON LADDER ^HOLDS <OBJECT>)))
```

FIGURE 20-17
Simplified syntax of OPS rules.

As with a literal in formal logic, a condition represents either the result or the negated result of some predicate test. In OPS this test is called a ***pattern,*** and it identifies the class name of an object to be tested and smaller tests to be performed on the attribute-value pairs of any such object. The pattern is true only if there is some object whose attributes have all the desired characteristics in working memory.

There are two kinds of tests, a basic test and a conjunctive test. The latter is a series of basic tests that must all be true with regard to the same attribute.

In any individual basic test, a predicate symbol may appear to the left of the value to be used in the test. The predicates available are primarily the common comparators such as equality, inequality, greater/lesser, etc. If no predicate is specified, "=" is assumed.

As with PROLOG, variables may appear in the value part of these test specifications, and they play a very similar role. They are identified by a symbol string surrounded by "⟨. . .⟩." Variables are local to a rule and are assigned values during the matching process that satisfies a pattern. Further, once given a value, a variable keeps that value throughout the process of trying to satisfy the entire rule. This permits propagating data from one pattern to another, or from the antecedent to the consequent.

Variables may be local to a single pattern or common to several patterns. In the former case, when a wme is found that matches the pattern, local variables in the pattern are bound to the matching value component in the wme. In the latter case, a variable that is shared across two or more patterns indicates the equivalent of a join—two different wmes must be found that share a value across certain attributes.

The consequent of an OPS rule is radically different from that of PROLOG. Instead of specifying a single literal to use as a goal, it specifies one or more ***actions*** to be applied to data objects in the working memory. The four major types of actions include one that creates a new object with specified attribute-value pairs (***MAKE***), one that modifies an existing object's attribute-value pairs (***MODIFY***), one that deletes an object entirely (***REMOVE***), or one that outputs some value to the console or printer (***WRITE***). The values for these actions may be constants or variables given values by the antecedent. Given that complete objects have no real names, the specification of which object is to be modified or deleted is by a number n in the action which refers to whatever object matched the "n-th" from the left pattern in the antecedent.

Figure 20-17 gives a sample rule that might be found in a program for doing symbolic simplification of algebraic expressions built from tree-like descriptions. Each subexpression corresponds to a wme of class "EXPRESSION," and it has attributes of the subexpression's name, the name or value of its left and right operands, and the name of the function between them. The specific rule shown in the figure looks for an object of type "EXPRESSION" whose operator attribute is "+" and whose right argument is 0. If one such object is found, this rule directs the system to cancel out the operator and right argument of the object and simplify the left argument. Note that the variable ⟨E⟩ shows up in two patterns, meaning that a join between wmes of type goal and expression must be performed.

The second sample rule comes from the ***Monkey and Bananas*** problem, in which there are bananas hidden in a locked chest in a room and a hungry monkey is to search until he finds them. This particular rule handles the case where there is an object on the ceiling that can be reached using a ladder, and the monkey's current goal is to get that object. Note that in this case we have multiple variables spanning several conditions, with one of them, ⟨PLACE⟩, in three.

20.6.3 The OPS Inference Engine

The most basic form of the OPS inference engine is simple to understand but highly inefficient. As pictured in Figure 20-18, there are three major steps interconnected in a cycle. The ***match step*** essentially tests the antecedents of all rules against all possible combinations of wmes currently in working memory. Each of those rules that is satisfied is packaged with the objects that satisfied it into an ***instance*** and grouped into the ***conflict set.*** This set represents all possible inferences that could be made at the current time. Note that the same rule may appear in several instances, each time with a different set of matching wmes. Also, the same object may be used several times, both within the same instance and among different ones.

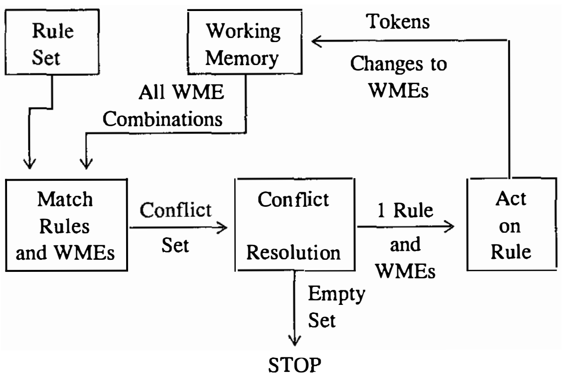


FIGURE 20-18
OPS interference engine.

The next step, ***conflict resolution,*** looks at this conflict set. If it is empty, there are no more inferences to be made, and the inference engine stops processing. If there is exactly one element, it is passed on. If, however, there is more than one instance in this set (the usual case), only one is chosen. How this choice is made is under some control by the programmer, and addresses some issues such as doing subgoals of a problem before larger goals, using older objects before newer ones, and preventing the system from getting caught in various kinds of inference loops.

The final ***action step*** takes the rule and matching objects selected by the conflict-resolution process and actually performs the inference. In the case of OPS, this inference involves executing the set of specified actions, which usually do some modifications to the working memory.

After modifying the working memory, the whole process repeats. The conflict set is conceptually recomputed from scratch. Thus it is possible for elements to drop out because of changes to objects they referred to, for new elements to enter the set because of the new values, for rules to stay in the set with the same objects but with different assignments to rule variables, or for the same combinations to exist again and stay in the set unchanged.

20.7 THE RETE ALGORITHM

(Forgy, 1982)

The potential for computational inefficiency in the prior model is obvious. If there are r rules, p patterns per rule, and w wmes, there are up to $r\times(w^p)$ combinations of rules and objects to try at each loop iteration. For any but the most trivial system, this number easily exceeds the capacity of any available computer (consider, for example, a moderate system of 1000 rules of 3 patterns each, with 1000 objects). Some very efficient kinds of optimization are necessary and have been developed in practice.

The most basic approach is called ***indexing,*** which stores with each object class name a list (or index) of only those rules whose antecedents might be affected by it. Now as changes are made to objects by inferences, only those rules on the list need to be reevaluated for inclusion or exclusion from the conflict set.

While this is a tremendous improvement over the brute-force approach, there is still the potential for a great many recomputations of patterns in the indexed rules for patterns other than those affected by the modified objects. Some way of saving the results of these other patterns could produce even more savings.

The ***Rete Match algorithm*** does exactly this by compiling from the rules a "network" that computes the antecedents directly, saves within itself the results of nearly all the pattern tests, including intermediate joins, and delivers as outputs just the changes to the conflict set from the previous iteration (see Figure 20-19). Consequently, when some action signals a change in some object, this change filters through the network, detecting where new patterns have been satisfied or where prior patterns that were satisfied have now become unsatisfied. Nodes in the network represent the various pattern tests and either pass or drop incoming changes on the basis of how they fare against the specified test. In addition, some nodes save the result of certain test sequences within themselves. Only when change signals manage to make it all the way through the network is any change made to the conflict set.

These networks are modeled after, and receive their name from, a neural network found in some animals.

Figure 20-20 diagrams how such a network fits into an OPS inference engine. Note that direct access to the working memory is no longer needed. All appropriate data can be stored inside the network.

20.7.1 Tokens

The operation of Rete networks has a great deal of similarity to the operation of ***data flow*** graphs. The descriptions of how wmes are to be changed are packaged in ***tokens,*** which are generated by the action step and absorbed by the Rete network. Individual nodes in the net wait for the arrival of appropriate tokens, save unprocessed combinations, fire as required, and generate new tokens as output.

All tokens have two components, a tag and a descripter. The latter is a description or pointer to one or more wmes. The tag normally takes on one of two values "+" or "−." The "+" tag indicates that the descripter represents something being added to the working memory, and it is generated by a MAKE or MODIFY action. A "−" tag indicates that something is being removed from working memory, and it is generated by a REMOVE or MODIFY action. Note that MODIFY actions send out two tokens in sequence, one "−" to remove the current values from the object being modified and one "+" to add in the new values.

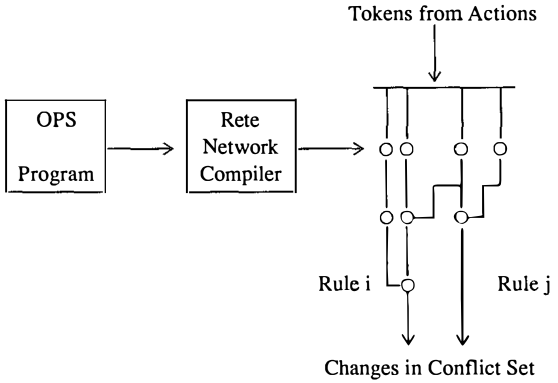


FIGURE 20-19
Construction of Rete networks.

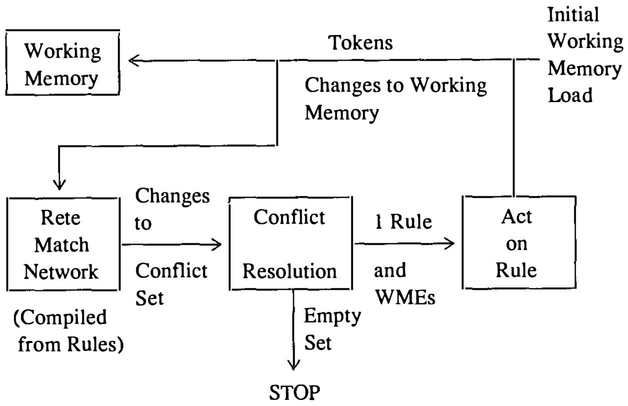
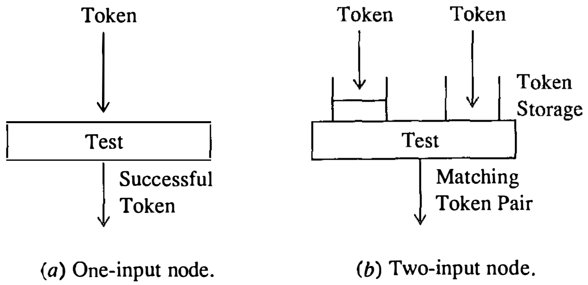


FIGURE 20-20
A Rete network-based OPS inference engine.

20.7.2 Basic Node Types

Within a Rete network there are several basic varieties of nodes, with an appropriate one chosen for each basic test in some rule's pattern. Figure 20-21 diagrams the structure of these node types. The ***one-input nodes*** correspond to tests that can be satisfied by looking at one object in isolation, and they operate like pure dataflow nodes. A token entering such a node represents a single object, and thus either passes or fails the test. If the test succeeds, a copy of the token is passed along the output of the node to successors in the net. With a failure, no token copy is passed. In either case, nothing is left behind in the node after firing. Note that the success of the test depends only on the descripter and not on the tag.



(*a*) One-input node. (*b*) Two-input node.

FIGURE 20-21
Principal types of Rete nodes.

Examples of the tests performed by one-input nodes include comparing (=, <, >,...) designated attribute values against specified constants, and/or verifying that multiple basic tests within a single pattern that use a common variable actually have compatible values. Figure 20-22 diagrams one such chain of nodes.

In relational model terms, a chain of one-input tests is exactly equivalent to a *select* operation.

The major tests that such nodes do not handle are those that involve variables used in more than one pattern. In such cases what must be found is a set of objects that individually pass the basic tests in their respective patterns, but that in addition have certain attribute-value pairs in which the values from the separate objects are related in some way (usually equality). This comparison is the responsibility of the ***two-input nodes***. Here each input comes from some string of one-input tests and represents objects that have successfully passed a series of simpler patterns. The test performed by the node is whether or not the values in the attributes that have a common variable do in fact have the same value.

Consider, for example, a pattern in some rule of the form "(Class1...^field1=⟨X⟩...)" and some other pattern in the same rule of the form "(Class2...^field2=⟨X⟩...)." Each pattern will pass objects from different classes with different attributes. However, jointly the two patterns in the same rule will pass a pair of wmes only if the value in "field1" of the first object equals the value in "field2" of the second.

Sample Antecedent Pattern:
(Class1 ^field1 = Expression ^field2 > 0 ^field3 <X>
^field4 ≠ <X> ^field5 <X>)

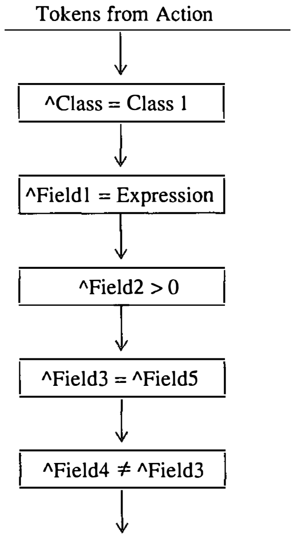


FIGURE 20-22
Sample chain of one-input nodes.

The resulting Rete net will have a separate string of one-input tests for each pattern to verify that all the tests other than the ones involving the shared variable ⟨X⟩ are passed. The output of these strings of tests then feeds a two-input node which will take objects from each input, select out the relevant attribute-value pairs, compare the values for equality, and output the pair of objects only if the values match. The resulting token has a descripter which includes both objects and which satisfies all the tests implied by the patterns, including the common variable.

If both input objects were guaranteed to appear at the same time, the construction of such two-input nodes would be only slightly different from that of the one-input varieties. However, in normal operations, objects that represent one input may appear long before, and in a different order from, equivalent objects for the second input. Further, it is entirely possible that an object for one input may match several different objects arriving at the second input over a long time period. This requires that there be storage at each input of all arriving "+" tokens, and that copies of these objects be removed only when tokens with "−" tags and appropriate descriptors arrive at the network. This storage will take the form of lists of tokens whose length can grow or shrink dynamically and arbitrarily as processing continues. Again, in terms of the relational model this is equivalent to a ***join,*** where the data for each relation arrives piecemeal.

20.7.3 AND Nodes

The most common type of two-input node is the ***AND node.*** Such nodes handle the case described above, where two separate patterns share a variable in equality tests. Objects passing the individual tests can satisfy the total antecedent only if the values they have in the specified attribute fields are the same.

For the detailed reasons given above, operation of these AND nodes depends on both the tags of incoming tokens and on what has arrived in the past at the other input. For "+" tokens arriving at a particular input, the steps involved are:

1. Add the token to the list associated with that input of the node.
2. Get the appropriate value from the token's descripter (i.e., the value specified by the shared variable).
3. Scan the list for the other input, looking for an entry whose appropriate attribute has the same value.
4. Each time the test succeeds, generate a "+" token as output, with a descriptor that points to the matching pair of objects. Then continue the search for more matches.
5. When all elements from the other input list have been tried, stop processing this node until another token arrives.

For incoming "–" tokens, processing in the node is somewhat different:

1. Scan the list for this input for the same object. Remove it, if found.
2. Scan the other list for any objects on it that had matched to the one just removed.
3. For each such match generate a "–" token as output, with its descriptor pointing to the matching pair.

20.7.4 NOT Nodes

The other common type of two-input node is the ***NOT node.*** This type handles conditions that combine both internal negations and variables shared with other patterns. In such cases only objects from the nonnegated patterns can pass through the NOT node. Passage is permitted only if there is no object in the negated side that has a shared variable of the same value as the object from the nonnegated side. The negated patterns thus act to establish a filter. Figure 20-23 diagrams this usage.

Operation of these nodes differs only slightly from that of the AND nodes, and detailed descriptions are left as an exercise to the reader.

One optimization trick that is often used in the implementation of these nodes is to keep with each entry in the left input a ***reference count*** of the number of matches that it currently has with the token stored in the right-hand side. Transitions to and from 0 signal the generation of output tokens.

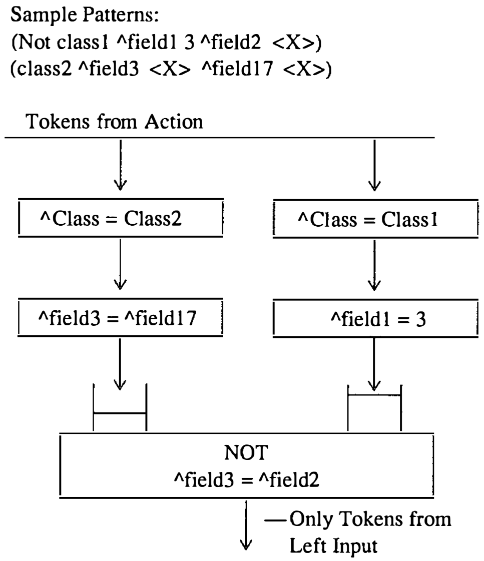


FIGURE 20-23
Use of a NOT node.

20.7.5 Other Node Types

In addition to the those above, there are other types of nodes that are needed to create complete networks, including:

- ***Root nodes*** to accept tokens from the action step and distribute them accordingly
- ***Two nodes,*** used as occasional place fillers
- ***Any nodes,*** used for combining conjunctive tests
- ***Terminal nodes*** to develop instances when appropriate for passage to the conflict-resolution stage

Finally, in many implementations the storage lists associated with the two-input nodes are actually split off as special node types of their own. This often permits sharing of lists between different strings of tests.

20.7.6 Optimization Techniques

For a single set of rules there are often many different Rete networks that one might build. Some, however, require less storage or less computation than others. For example, when compiling a single pattern, as many one-input nodes as possible should precede any two-input nodes. This reduces the total number of tokens that might have to be stored in the two-input node's input lists, reducing both storage and comparison time.

Additionally, within the string of one-input tests, those that tend to filter out the most tokens should be placed first. This reduces the number of tokens flowing farther down.

Similarly, whenever two patterns (potentially from different rules) share exactly the same test or subsequence of tests, only one set of nodes need be built, with the output fanning out to drive the rest of the individual networks for each pattern. This is called ***node sharing.***

A variation of this idea might handle the case where many patterns (again from potentially different rules) test the same attribute of a class of objects, but for different values. Here a new kind of multiple-output ***hash node*** might test all such objects once, and direct each token to the output corresponding to the particular value that the object had. Figure 20-24 diagrams a sample network using many of these techniques.

20.8 AN ABSTRACT MACHINE

(Forgy, 1982)

Just as was done with PROLOG, the major activities that occur within a Rete network can be expressed as relatively simple programs for a von Neumann-like abstract machine often called the ***Production-Rule Machine*** or ***PRM.*** This has been done, with interpreters for this machine built as programs for conventional computers. Compilers then can take an OPS program as input, compile the rules into programs for this abstract architecture, and run the resulting program on the abstract machine inter-

Sample Rules:

```
(ADDX0 (GOAL ^TYPE SIMPLIFY ^OBJECT <E>)
        (EXPRESSION ^NAME <E> ^RIGHT-ARG 0
                    ^OPR +    ^LEFT-ARG <X>) ——→ ...)
(TIMESX0 (GOAL ^TYPE SIMPLIFY ^OBJECT <E>)
        (EXPRESSION ^NAME <E> ^RIGHT-ARG 0
                    ^OPR *    ^LEFT-ARG <X>) ——→. ...)
```

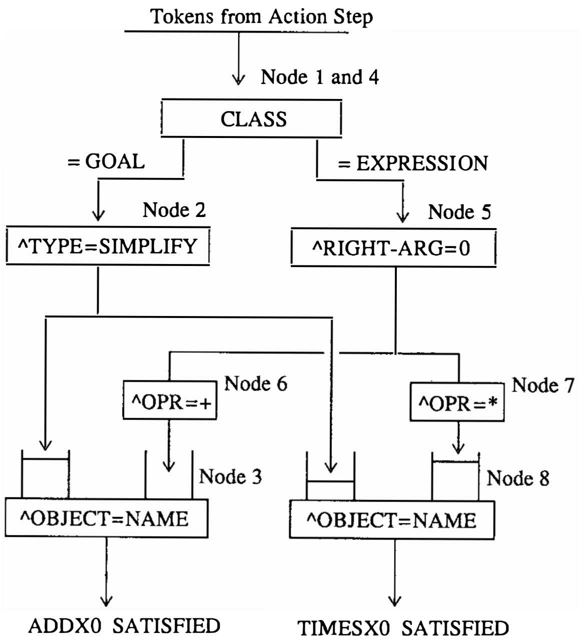


FIGURE 20-24
Sample Rete network using node sharing.

preter. Alternatively, each instruction can be ***open-coded*** as described before into a series of native machine instructions.

Figure 20-25 lists the general instruction classes for this architecture, with the data structures maintained during execution outlined in Figure 20-26 and sample code for the prior network shown in Figure 20-27. The ***Token Queue*** contains the tokens generated by actions, but not yet processed. The first of these is the token currently being processed by the net. The ***Program State Stack*** is a stack of labels and tokens representing various points in the network that must be resumed at some point. It is highly reminiscent of the ***Choice Point Stack*** in the WAM for PROLOG. The ***Token Lists*** for each AND instruction represent the operand tokens being stored by that instruction.

Execution begins by setting the Program Counter to the ROOT instruction and initializing the Token Queue with tokens from the prior cycle's actions. The top such token is processed first. The internal contents of token lists are initially unchanged from that left by the end of the last cycle.

Instruction Class	Operands	Function
TESTxxx	a,v	Test attribute "a" of current token against value v using predicate "xxx." Continue with next instruction only if match.
AND	a,xxx,b	Continue execution only if some token on left input has a value for attribute a that passes predicate xxx for attribute bs value in the right input.
NOT	a,xxx,b	Continue execution with some token from the left input only if no token from the right input passes the "xxx" test between attributes "a" and "b" respectively.
FORK	label	Pass current token to both the next instruction and instruction at address "label." Implemented by stacking the label and a token copy.
MERGE	label	Join with some other results at "AND" node found at "label." Typically used to direct tokens to right input of two-input nodes.
TERM	label	Flag rule found at "label" upon entering conflict set, with current token forming instance.

Note: "xxx" represents some test predicate like "=," ">," ...

FIGURE 20-25
Instruction set for PRM abstract machine.

As in a conventional computer, instructions are executed one at a time, in a specific sequence. FORK instructions indicate that the current token must be passed to at least two separate sets of code in the program, and at execution time the machine must stack the label in the instruction with a copy of the current token being processed. This corresponds to places in the network where the same token must be passed to several destinations.

One-input nodes correspond to TEST instructions which apply the specified test to the current token. On a successful test, control continues to the next instruction. On a mismatch, processing stops on the current chain of instructions. If the Program Stack is not empty, the top element is popped back into the machine state, and execution is picked up at that designated point. If the Program Stack is empty, control restarts at the ROOT instruction with the next token from the Token Queue.

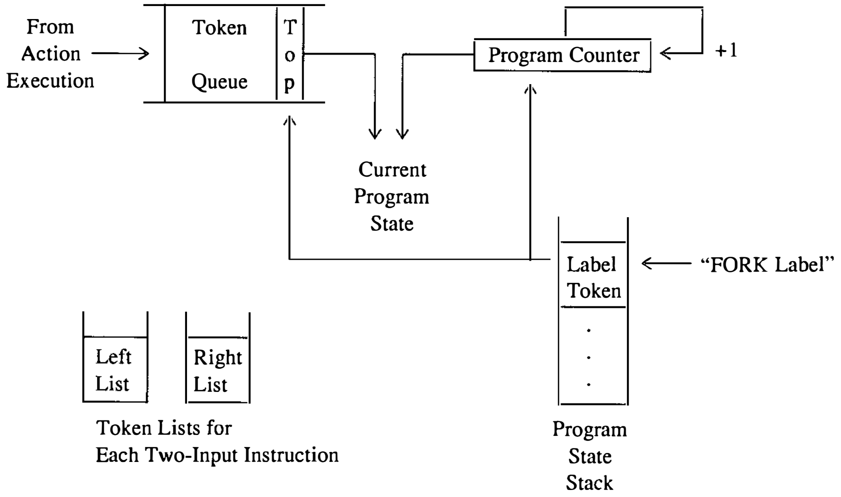


FIGURE 20-26
Data structures in PRM.

```
Root:  FORK    Node4
Node1: TEQ     "class,""goal"
Node2: TEQ     "type,""simplify"
       FORK    Node8
Node3: AND     "object,"=,"name"
       TERM    Addx0

Node4: TEQ     "class,""expression"
Node5: TEQ     "right-arg,"0
       FORK    Node7
Node6: TEQ     "opr,""+"
       MERGE   Node4

Node7: TEQ     "opr,""*"
       MERGE   Node8

Node8: AND     "object,"=,"name"
       TERM    TIMESX0
```

FIGURE 20-27
Code segment for previous network (no hashing).

The AND and NOT instructions handle the two-input nodes directly. Each such instruction has associated with it two lists of token inputs. When execution falls into it from the previous instruction, the current token is added to the left list and compared against the right list. A successful match continues execution as before, but with additional state information pushed on the program stack to resume the comparison process when this first match has run its course. Completion of the comparison process without any more matches triggers a backtrack similar to that for the TESTs.

MERGE instructions bring a chain of nodes into the right-hand side of a two-input node. They function as a normal JUMP-type instruction would in a conventional computer, but with the extension that if the target of the jump is an AND or NOT, then the current token goes into the right-hand list of that instruction, with the rest of the processing reversed from that discussed above.

Finally, TERM instructions include as an operand a pointer to the action code for some rule. Executing such instructions pairs the current token with the label as an "instance" for the conflict set. This is followed by a return to either label-token pair on top of the Program Stack or the ROOT as described above.

20.8.1 Measured Statistics

(Gupta and Forgy, 1983)

Earlier in Chapter 19, statistics on the execution of programs in the Warren architecture gave some significant insights into how PROLOG programs actually behave. The same type of study has been performed on the Rete net architecture, with similar results. In Gupta and Forgy's analysis, an OPS inference engine was instrumented and six separate OPS programs were analyzed. The results were summarized in three parts: properties of the programs themselves before compilation, properties of the constructed Rete networks, and properties exhibited during execution.

The numbers given here are averages of subsets of the numbers actually measured; the original paper should be consulted for more detail.

Figure 20-28 lists the properties of a rule as entered into the OPS system and averaged over all six programs. The column labeled "Variations" lists the smallest and largest numbers encountered. From these statistics, the average rule has approximately 4 conditions (10 percent of them negated), almost 3 actions (2 of which manipulate objects in working memory), and uses about 3.5 unique variables approximately 1.7 times each.

Figure 20-29 lists the properties of the Rete networks compiled from these rules. The numbers here reflect the optimization techniques discussed earlier, particularly node sharing. However, they do differ from Gupta and Forgy's numbers by not separating out as separate nodes the memory lists forming the input to the two-input nodes.

The final column of the table lists the ratio of nodes found in an unoptimized net to that for the optimized one. The results show clearly the advantages of optimization, with 2.5 times fewer nodes in the optimized version.

Figure 20-30 shows what happened when these networks were executed. The first table lists the average number of nodes (instructions) executed in an optimized network for each token generated by an action. Although the great majority of these are tests, the complexity of the other nodes more than compensates when total execution time is considered.

	Properties per Rule			Mix of Action Types	
	Average (%)	Variations (%)		Average (%)	Variations (%)
Conditions	4.1	3.1–5.8	MAKE	28	10–71
Actions	2.8	1.8–3.6	MODIFY	24	7–35
Total Variables	5.9	0.16–10.5	REMOVE	12	0–26
Unique Variables	3.47	0.42–7.52	WRITE	13	3–44
			OTHER	23	2–44

	Properties of Condition Element		
	Average	Variations	
% Negations	10%	1.3–14.2%	
Attributes tested	3.88	2.08–4.73	(including class check)
Variables	1.36	0.24–2.69	
Rete nodes	1.94	1.8–2	(Unshared Network)

*Above tables adapted from Gupta and Forgy (1983).

FIGURE 20-28
Properties of an average OPS program.

	Nodes per		Ratio of Unshared
	Program	Rule	to Shared Nodes
Total nodes	5120	5.44	2.5
Test nodes	1311	1.33	6.34
AND nodes	1845	2.17	1.21
NOT nodes	481	0.43	1.11
TERM nodes	909	1	1
Other nodes	574	0.51	1.39

FIGURE 20-29
Properties of an average optimized Rete network.

Since performance of AND and NOT nodes is a function of the length of their input lists, we also include a table of their average length and the number of times such lists are changed in length as a result of the arrival of a single token at the net's input. The two columns reflect the number of wmes that were joined together into a single token element. The normal case after a series of one-input tests is "1." When one two-input test feeds another, the queue at the input to the other has elements that have more than one wme referenced in them.

Also listed is the average number of objects in working memory for a subset of the six programs, and the average number of changes to working memory (tokens generated) over an average execution of each program.

In terms of machine instructions to execute these nodes when using something like the PRM of the previous section as an intermediate step, approximately 3750 instructions were required to process a "+" token, with about 940 of these used to construct entries in the buffers associated with the two-input nodes. Processing a "−" token required 1800 instructions. The actions averaged about 500 instructions per rule firing.

Node Type	Nodes Executed per Token Entering Net
Total	134.26
TEST nodes	79.77
AND nodes	29.6
NOT nodes	5.89
TERM nodes	1.98
Other	7.13

	#Tokens paired in token list entry.	
	1	>1
Average Length	25.17	20.3
Changes/Token	5.09	4.79

	Working Memory	
	Average	Variations
Number of objects	295	60–708
Number changes	3502	559–16839

FIGURE 20-30
Execution properties of a Rete net.

Finally, a more careful analysis of the data leading to all of these averages reveals wide variations in many terms, particularly those associated with two-input-node input lists. Thus, these numbers should be used only to provide a rough indication of activity within a Rete net, with more detailed simulations needed for real benchmarks.

20.9 MACHINES

The abstract machine of Section 20.8 is a typical approach to the implementation of production-rule systems on conventional machines. The following sections give brief descriptions of some of the other approaches that have been tried or suggested.

20.9.1 Opportunities for Parallelism

(Gupta, 1987; Forgy et al., 1984)

Production systems, particularly as implemented with Rete nets, seem at first glance to be rich in opportunities for parallelism. These include all three steps of the execution cycle: match, resolve, and act.

Match parallelism deals with the process of determining which condition elements are affected by new wmes and then how to update the internal state. There are at least four separate opportunities for parallelism here:

- ***Production parallelism,*** in which the rules are split into groups and the matching for these groups is done in parallel
- ***Node parallelism,*** in which multiple two-input nodes inside the Rete net are executed in parallel
- ***Intranode parallelism,*** in which activations of the same two-input node can be run in parallel
- ***Action parallelism,*** in which each of a set of changes to working memory is computed in parallel

Detailed simulation studies by Gupta indicate that production parallelism suggests a speedup as high as the number of production rules affected by a wme change (up to 26), but averages only about 1.9, mainly because of the variation in time that production rules take to process and the increase in redundant computations that occur when a Rete net is split to permit concurrent execution. Node parallelism is a little better, about 2.9, but is limited by the number of two-input nodes that are triggered by a wme. Intranode parallelism actually seems to be the richest single source, with a speedup of 3.9, mainly because of the fairly large number of cases where a ***cross-product effect*** triggers multiple matches from a single two-input node. Action parallelism is useful to increase the amount of parallelism available to the other three methods, and can boost their numbers by somewhere between 1.5 and 2.

Conflict-resolution parallelism refers to speeding up the selection of which of several instantiations in the conflict set should be fired. Although it is clear that much of this could be performed in parallel, only limited studies have been done because of the minimal time spent in the average program in this phase.

RHS parallelism (for right-hand side) attempts to execute the actions that produce the wme changes in parallel. As with conflict resolution, little effort has been spent trying to quantify achievable speedups because the time spent in this phase is so small relative to that spent in the match phase.

Overall, experiments to date indicate that a speedup factor of somewhere around 10 is possible on large production-rule programs when parallelism of the order of 20 to 30 processors is available.

20.9.2 The DADO Machine

(Stolfo, 1985; Miranker, 1984, 1990; Ishida and Stolfo, 1985)

The ***DADO Machine*** was a highly parallel machine with many small processing elements (PEs) arranged in a treelike hierarchy much like Mago's ***Cellular Tree Machine*** described in Chapter 12. Prototypes were running at Columbia University in the mid-1980s.

The original algorithm to be run on the DADO was one in which one or more production rules would be stored in intermediate levels of the tree, with the leaves below it used for storing wmes. These intermediate levels were called ***PM levels,*** for "pattern matching." As wmes were created in the *act phase,* they would be sent up the tree to the root, and then broadcast back down all paths. Each PM node would check it against its rules for possible use. If it was useful, a copy would be saved in a PE. Then, during the *match Phase,* each PE would broadcast one of the rule's condition elements to its slave PEs below it. If a match was reported back, its binding to variables would be collected and substituted into the next condition element. If a consistent set was discovered, the rule would fire and the cycle repeated.

Even though there was significant parallelism, the lack of internal state saving meant that a huge amount of recomputation was performed. Consequently, a new algorithm, halfway between the original algorithm and Rete's, was developed. This was called ***TREAT,*** for "Temporally REdundant Associative Tree." It saved internally all wmes that passed series of one-input nodes, but it did not save any outputs of internal two-input nodes. These would be recomputed much as described before. The big savings came from not having to scan all of working memory, but only those wmes that had already passed the initial tests.

Optimization algorithms would position the PM level of the tree so that there would be sufficient PM nodes to handle the average number of rules that might be tickled by a wme change. The OPS5 statistics indicated that this latter average would typically be in the range of 20 to 30. This would mean that a PM level 5 above the leaves would be appropriate. If an algorithm could be developed to scatter around the PMs at this level so that on average only one rule per PM would be active, then the maximum achievable parallelism would be achieved. Other optimizations consisted of scattering the wmes around the leaves so that at most one output of each one input string was stored per leaf. Again this translates into maximum parallelism during the match phase.

Analysis of this algorithm indicated that significant storage reductions would be achieved, but with at best moderate performance gain.

20.9.3 Associative Processing

(Kogge et al., 1989)

Chapter 8 introduced ***associative memories*** and their usefulness in storing lists. Chapter 19 surveyed approaches to using them to simplify PROLOG execution. Given this, it should be no surprise that associative memories also appear useful in implementing Rete nets. Figure 20-31 diagrams one approach. It assumes an associative memory word as long as the longest wme, and that these words can contain not only 1s and 0s but also a third code for "don't care." Thus a datum presented to the memory will match a word in the associative memory if its individual bits match exactly all those bits in the latter that are not don't cares.

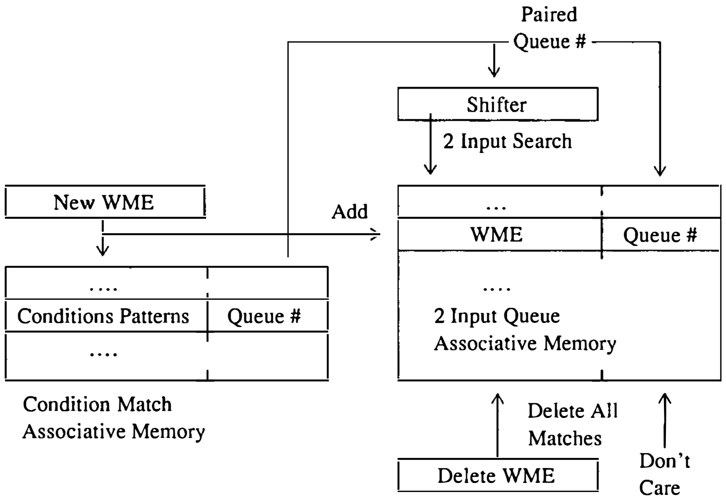


FIGURE 20-31
Simplified use of associative memory.

Two different data structures are kept inside the associative memory. One consists of one entry for each condition element in the program. The fields in each entry that are not "don't cares" correspond to attribute pairs that are tested via simple compares in the condition. All new wmes are compared against these entries. Each match signals that the wme should be included in the input queue for some two-input node. This is done by numbering each queue and then appending that number to the incoming wme.

This extended wme is then added into the second associative memory. All wmes in two-input nodes of the Rete net are stored there. After adding a new entry to a queue, a table lookup obtains the number of the queue on the opposite side of the two-input node. The appropriate field of the new wme is then selected and shifted over the position of the matching field in wmes for the other side. It is then tagged with the other queue's number, and an associative search is made for all entries that match just those two fields. All matches are extracted, paired with the original wme, and passed back through the same associative memory as the output of the two-input node.

The big performance gain is in the queue searches. If the statistics presented earlier are true, then instead of comparing the input to a two-input node with 20 to 30 elements on the other side, only a single associative cycle is needed.

In addition, on wmes being deleted, not even this search speedup is needed. We can simply look at *all* the elements of *all* queues simultaneously for components that match the wme. All those that match can be deleted and their storage marked as free for future use. Absolutely no node-by-node processing is needed.

The net effect is that if all other overheads are equal, then somewhere between a factor of 10 and 100 speedup in terms of machine cycles for the match process seems achievable. Ongoing experiments will verify this.

20.10 PROBLEMS

1. Prove that **select** distributes over union, intersection, and set difference.
2. Consider a simple form for a relation as an s-expression list of tuples, where all tuples are of the same length, and all are lists of simple constants. An example might be ((1 2 3 4) (3 2 43 4) (5 6 7 8) (9 10 11 12) (3 22 43 53) (3 24 7 99)).

   Devise a small s-expression-based language where the permissible query functions are **select, project,** and **join.** Show the Backus Naur form (BNF) for your language and design an abstract program interpreter for it. Assume that select patterns are of the form ({⟨constant⟩ | *}+) where "*" stands for no test, the project column selector is a list of integers, and the columns joined over are shown as a list of pairs of numbers.
3. Show what your interpreter from Problem 2 returns for the following:
   - **a.** A select of the form (3 * 43 *) applied to the sample relation
   - **b.** A project of the form (1 3) applied to the sample relation
   - **c.** A join of the form ((1.2) (3.3)) performed between the above and the relation ((12 1 3) (44 1 43) (6 3 43) (2 14 43))
4. Show for your language from Problem 2 how you would form a query which takes a select of one relation in which the first argument must equal 3, joins arguments 1 and 3 of the result with arguments 2 and 3 of some other relation, and projects the result along columns 1 2 4 5 7. (*Hint:* doing this for the example in Problem 2 would yield ((3 2 4 6 43) (3 22 53 6 43)).
5. Form the same query as in Problem 4 using PROLOG, where the way each tuple gets printed out is via a clause of the form: **solve-all(***db1, db2***)**:−**query(***db1, db2, tuple***)**, **write(***tuple***)**, **fail.**
6. Form the same query as in Problem 4 using SQL, where the attribute name for the i-th column is "column-i."
7. Expand the language and interpreter you have created to permit selects to compare columns within themselves. For example, a select pattern of the form (* 22 x x) would look for tuples where the second component is 22 and the third and fourth components are equal.
8. Describe how you might do a join of more than 4096 tuples on the RDBE.
9. For the databases of Figure 20-3 compute:
   - The natural join
   - An equijoin between the "Dept." columns
   - An equijoin between both the "Dept." and "Number" columns
   - A select of the first of the form "Time=10AM"
   - A projection of the previous result over "Room."
10. Build a Rete net for the second rule example of Figure 20-17 and then transcribe it into PRM code.

11. Define the semantics of a NOT node carefully.
12. Attempt to map the PRM instructions into WAM instructions. Could you build a Rete net in WAM code easily?
13. Assume a working memory consisting of wmes of class **person** with attributes **name, age, employment status,** and **employer.** Other wmes of the class **skills** have attributes **name** and **skill-type.** Develop a production-rule program which will modify any entry of the above when a person is between 20 and 35, has a skill as a programmer, is unemployed, and will be employed by "Vaporware, Inc.," with a new skill-type of "manager." For each such entry, develop a new wme of class **Vaporware-exec** and attributes **name, age,** and **title** (default is "manager"). Also, print out the name of the youngest such person, and make his or her title "company-president."
14. Develop the Rete net for Problem 13.
15. Develop a set of OPS rules which completes the symbolic simplification process of Figure 20-17 to take a collection of wmes representing a complex algebraic expression and simplify it. (Express all the standard simplifications involving 1 and 0 over +, −, ×, ÷, plus subtracting or dividing something by itself.)
16. Develop a Rete net for your answer to Problem 15.

CHAPTER 21

PARALLEL INFERENCE ENGINES

Parallelism in a computation occurs when two or more related steps are carried out at the same time. In function-based systems this includes applicative-order and "eager-beaver evaluation." ***Parallelism in a language*** refers to the capability of expressing opportunities for parallelism in computations defined by programs written in those languages. Again, in functional languages this includes explicit demand-driven and lazy-evaluation annotation. This chapter investigates the basic opportunities for parallelism in computation that might be found in logic-based systems that resemble PROLOG in certain ways—namely, statements based on Horn clauses in first-order logic and a backward-driven inference engine whose goal is to find a substitution that makes some initial query consistent with the axioms expressed by the logic program.

In particular, we introduce three common opportunities for parallelism: in the process of unifying two literals, in the choice of which clause to attempt to match against a goal (***OR parallelism***), and in the solution of multiple goals at the same time (***AND parallelism***). Variations of each opportunity are presented, with emphasis on the effects on implementation. One model in particular, the ***AND/OR Process Model,*** has attracted a great deal of attention because it combines much of the potential of both AND and OR parallelism.

Finally, there is a form of logic language called ***committed-choice, nondeterministic language,*** which combines a form of AND parallelism based on something very similar to ***streams*** in functional languages with an inference engine that never backtracks. Such languages, often called ***guarded Horn clause*** languages, represent a major new trend in logic programming and warrant careful attention.

Although this field of parallel logic computing is quite young, the amount of material being published on it is already so extensive that no single chapter can do it justice. It is highly recommended that the interested reader use this chapter as a brief introduction to the field, and then consult the references (and publications since this book was prepared) to really explore the subject.

21.1 TAXONOMY OF LOGIC PARALLELISM

(Douglass, 1985)

The primary reason for an interest in parallelism is, of course, speed of execution. Ideally, if one had N processors, it would be nice to solve problems in $1/N$th the time required to solve them on a single processor. Needless to say, the real world is never that simple, and most typical programs exhibit nowhere near an N-fold speedup.

There are several reasons for this. First is the inherent nature of a computation, which may limit parallelism regardless of how many processors are available. Two good examples of this are sorting or adding up N numbers. Neither can be executed in less than something proportional to $\log(N)$, regardless of the number of processors. Next is the propensity of conventional von Neumann-based programming languages to "mask" opportunities for parallelism behind so many sequential constructs that it becomes impossible, even for sophisticated compilers and support software, to reconstruct a parallel form. Finally, limitations on communication in real parallel computers often limit achievable parallel computation even when the language used permits its expression. If multiple processors have to queue up to use the same memory or select the next task to execute, or for data to go through multiple processors before reaching its intended destination, then what should take one step conceptually may actually take something proportional to the number of processors, negating their computational benefits.

Logic computing based on Horn clauses and PROLOG-like inference engines seem on the surface (and even deeper) to offer opportunities to beat many of these roadblocks. The creation and expansion of a deduction tree looks ideal: Different nodes could be expanded at the same time by different processors. In addition, requests for "all possible solutions" to a query are also common, and could trigger useful work in different processors by expanding the same nodes in different fashions (essentially building all possible closed deduction trees). Finally, as we saw with PROLOG, there are many cases where a simple sequential approach to potential inferences may end up in a very deep, or even infinite, "blind alley," while a real solution might be found almost immediately if one had only "looked down" some other path. A parallel model of a logic computation might look down multiple paths at the same time, guaranteeing a solution in cases where a sequential one might never terminate. In a real sense, a parallel model for logic computation is more *complete* than simpler sequential forms.

The following subsections describe several levels of parallelism that might exist in programs, and then maps three characteristics of logic programs into them. Succeeding sections of this chapter discuss each of these opportunities for parallelism in more detail.

21.1.1 Implementation Parallelism

Parallelism in implementations comes on many levels, and can roughly be divided into four major categories: microscopic, fine-grain, medium-grain, and large-grain. Figure 21-1 diagrams simple examples of each category.

Microscopic parallelism occurs when very small computational steps are performed in parallel. A trivial example of this is adding up two N-bit numbers by providing logic gates to add each of the N bits in parallel, and then combining the individual sums via a parallel tree of logic. Another example might be reading out N bits of information from a memory in one access by dividing the memory logic into N separate arrays of storage elements and accessing all in parallel. At this point in technology development we consider such parallelism commonplace, and will not consider it further. Even conventional von Neumann computers employ them extensively.

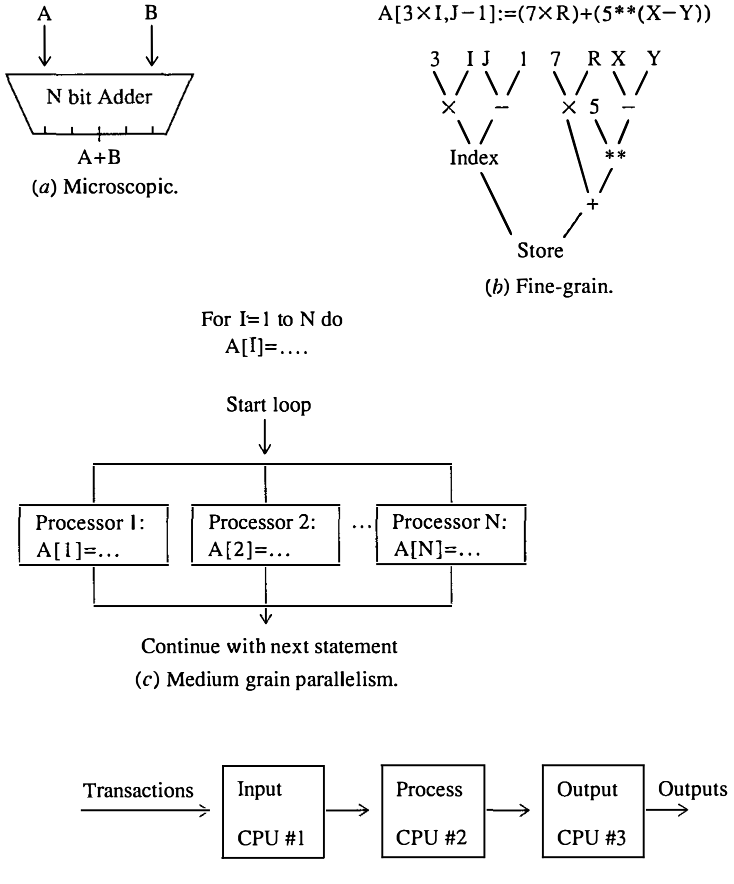
The other forms of parallelism are more significant. ***Fine-grain parallelism*** corresponds to multiple relatively small but self-contained operations being executed in parallel. In terms of conventional programming languages, it is roughly equivalent to executing parts of a single statement in parallel. For example, in the statement

$$A[3\times I,J-1]:=(7\times R)+(5/(X-Y));$$

the computation of some of the index terms $3\times I$ and $J-1$ could be executed concurrently with $7\times R$ and $X-Y$ (assuming that there are no side effects).

The major problem with fine-grain parallelism is contention for shared access to common variables and interlocks to guarantee proper execution (e.g., distributing all copies of a variable before updating it). This can often rival or exceed the time required to perform the operation to begin with and can thus wipe out any potential gains from parallel execution.

Medium-grain parallelism corresponds roughly to executing several complete statements in parallel. Here we still have problems with access and synchronization, but the major problem is often finding enough statements that are truly independent of each other to execute in parallel. It is



(*a*) Microscopic.

(*b*) Fine-grain.

(*c*) Medium grain parallelism.

(*d*) Large grain parallelism.

FIGURE 21-1
Types of parallelism.

also often difficult to judge the approximate execution time of a statement, particularly when there are embedded procedure calls or conditional operations in it.

Large-grain parallelism corresponds roughly to executing something the size of a whole block, procedure, or task of code in parallel with other such code segments. One example of this might involve executing multiple copies of the body of a loop in parallel with different index values for each. Another might be a reservation system of some sort, where requests come in, are analyzed for their contents in one processor, scheduled and allocated in another, with appropriate tickets, confirmation letters, bills, etc., prepared in a third. Synchronism in such cases is still a problem, but in conventional languages it is the inability to guarantee no interactions between different blocks of code that makes true parallelism hard to achieve in many real applications.

In the world of conventional computers and programming languages, all of these approaches have been tried. ***Single-instruction, multiple-data (SIMD) vector processors*** are good examples of fine-grain parallelism. The same function (usually at the level of a single arithmetic operation) is applied to all elements of one or more ***vectors*** of data. ***Closely coupled multiprocessors,*** particularly those with ***shared memory*** organizations, are well suited for medium-grain parallelism. Finally, ***distributed processors,*** in which the connection between individual computers is via something akin to an I/O channel, are a good match to large-grain parallelism, in which each processor can run autonomously for relatively long periods of time, with little data exchange of data with other processors.

21.1.2 Parallelism in Inference Engines

Within the execution of Horn clause logic systems there are at least three areas where parallel execution might be considered: unification parallelism, OR parallelism, and AND parallelism (Figure 21-2).

Unification parallelism corresponds to fine-grain parallelism and consists of trying to unify all the arguments of two literals at the same time. The basic operations are comparisons and whatever mechanisms are used to save computed substitutions. The most significant problems relate to guaranteeing consistency of results and use of common substitutions.

As one example, consider unifying **p**(3,*z*,5) with **p**(*x*,*y*,5), where *x* and *y* are unrelated variables with no current assigned values. All three argument unifications could go on in parallel, with a final synchronization needed to determine if any fail and to combine the generated substitutions. If, however, the second literal is of the form **p**(*x*,*x*,5), then the second unification test must be delayed at least partially until completion of the first. (Consider what happens when *z* does and does not have an initial value bound to it).

Theoretical results (see Dwork et al., 1984, for example) have placed bounds on this process. For two arbitrary literals with *n* arguments, the best speedup that can be guaranteed with something proportional to *n* processors is log(*n*). This is because of the potential need to

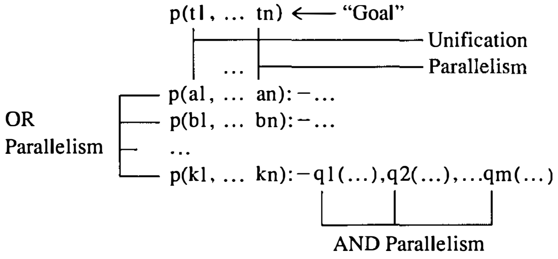


FIGURE 21-2
Opportunities for parallelism in logic programs.

unify all arguments in a treelike structure of log(n) levels. If substitutions go only toward variables in one literal (as is the case with production-rule systems of the last chapter), then a slightly better speedup of $\log^2(n)$ is possible with a number of processors that is some polynomial function of n (see Ramish et al., 1989). This latter case is sometimes called ***term matching***. Singhal and Patt (1989) describe some simulation experiments with a parallel unifier in the PLM machine discussed in Chapter 19. The speedup over several benchmarks was of the order of 1.7.

Because of this limited speedup and the fact that only part of the program's time is spent in unification, this level of parallelism will not be addressed further.

OR parallelism comes from the observation that there are usually multiple clauses with the same predicate symbol in the clause head. When a goal is created with that symbol as its predicate, any of these clauses may be the one that matches and contributes a solution. Further, if we are after all possible solutions, it may be necessary to try all such clauses. In this case it is often possible to try each clause independently on a separate processor. The major timing constraint is keeping track of individual results (success/failure indications and matching substitutions) as they come back, and returning them to the caller of the original goal in the order (if any) expected. This corresponds closely to medium-grain parallelism, in which individual statements are executed in parallel, although the total amount of work triggered by each statement may be larger and the amount of synchronization somewhat simpler.

AND parallelism comes about when one looks at the antecedent (right-hand in PROLOG) side of a clause and observes multiple literals each of which must be proved before the entire clause can be said to succeed. A parallel execution of this would attempt to solve each such literal separately. The major problems include:

1. Keeping track of the success/failure of individual literals
2. Making sure that substitutions are consistent across all literals
3. Determining when the clause fails as a whole

Finally, compounding these difficulties may be the desire to compute all possible proofs of the goal that triggered the clause in the first place. This requires that all possible solutions for individual antecedent literals be beat against each other to find all possible consistent matches.

Because each of the activities that are carried out in parallel resemble the overall goal (finding a deduction tree), AND parallelism is very definitely large-grain.

21.1.3 Representation: AND/OR Trees

AND and OR parallelism complement each other very well. Each process in an AND parallel step (a new goal to prove) may be solved by an OR parallel arrangement that tries all possible clauses that match it. In turn, a clause that successfully unifies under one OR process may spawn an AND parallel process to prove its right-hand side.

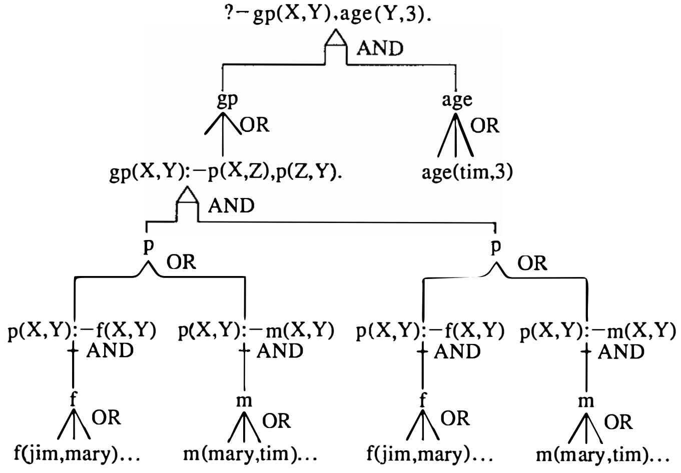
The definition of a ***closed deduction tree*** in Section 15.7 corresponds to a pictorial representation of one solution to some goal. Each subtree matches a clause that an OR parallel activity found to be suitable. The set of subtrees that joins to one subtree maps to the results of an AND parallel activity. This continues throughout all levels of the deduction tree.

It is possible to draw a diagram similar to a deduction tree which simultaneously shows all possible deduction tress and all opportunities for AND/OR parallelism. Such a diagram is termed an ***AND/OR tree***, and will be of use later in our detailed discussion of AND/OR parallelism. The major difference from a deduction tree is that we suppress the unification information. Figure 21-3 diagrams a simple example.

As with deduction trees, AND/OR trees are built from subtrees representing individual clauses. The root of the subtree is labeled with the predicate symbol of the consequent literal. Leaving this node are a series of arcs, one per antecedent literal, which lead to nodes labeled with the predicate symbol of the corresponding literal.

The root nodes of such subtrees are termed ***AND nodes*** because all the children nodes must be satisfied before the root node is satisfied. A horizontal line is often drawn between the arcs of such a subtree to reinforce the concept of this coupling.

The difference between AND/OR trees and deduction trees comes



Note: gp = grandparent
p = parent
f = father
m = mother

FIGURE 21-3
A simple AND/OR tree.

about when we expand the children of an AND node. In deduction trees we simply connect up the root of some other AND node. In AND/OR trees we instead make each child an ***OR node*** which expands downward to multiple AND nodes, one for each possible clause whose predicate symbol matches. A solution to any one of these lower nodes will satisfy the OR node.

AND/OR trees thus have levels that alternate between AND and OR nodes. If one level of the tree is all OR nodes, the next is all AND nodes, and vice versa.

Construction of an AND/OR tree is relatively simple. The root of the whole tree corresponds to the original query expression. If there is only one literal in the expression, this root is an OR node which expands to multiple children, one for each possible clause that might match. Each of these children are AND nodes. If there are multiple literals in the query, it is an AND node (all the literals must be satisfied), and it expands to a set of OR nodes, one for each query literal.

Expansion of the tree then continues recursively. Each unexpanded OR node expands out to a set of AND nodes, one for each possible clause with the same predicate symbol. Each unexpanded AND node corresponds to some clause in the logic program and is expanded out to a set of OR nodes, one per literal in the clause's antecedent. Note that a clause corresponding to a fact (no right-hand side in PROLOG form) does not need to be expanded further, and serves as a leaf on the tree.

For simplicity, when there is only one clause that might satisfy an OR node, we often suppress the single OR-level arc, and instead change the node to an AND node and expand accordingly. Likewise, when a clause has exactly one literal on the right-hand side, we often suppress the single AND arc and immediately show the OR alternatives.

Some additional comments about such trees are in order. First, in many real cases where clauses call each other recursively, the resulting trees can be infinite (draw out the tree for **append** as an example). This could be solved by making the tree a graph and routing an arc from lower in the tree back up to an earlier one. Usually, however, we are interested in only a small swatch of the computation, and thus limit our attention to perhaps two or three levels of a tree, leaving the lower ones unexpanded.

Also, we often have constants in query argument positions which can in turn eliminate beforehand many possibilities from an OR expansion. This means that the same predicate symbol appearing as an OR node in several parts of a tree may be expanded differently because of what we know beforehand about its arguments in each case.

Finally, the flow of substitutions in these trees should be discussed. In general, a substitution generated by a child of an OR node flows back up through the OR node toward its parent. If the parent is an AND node, the substitution is broadcast sideways across all other children of that parent. This is required to guarantee consistency of variable assignments across all the children.

If an OR node generates several possible substitutions (corresponding to different possible solutions), there must be a corresponding set of separate sideways broadcasts, each of which in turn could generate a separate substitution flowing up through the AND parent.

21.1.4 Parallelism in Language Constructs

Functional languages utilize a variety of constructs to permit expression of opportunities for parallelism, including various functionals, such as **map,** and data structures such as ***streams.***

Many logic languages have a similar, if not richer, set of constructs available to the programmer. First is a specification that not one but *all solutions* that satisfy some query are desired. This permits use of virtually all the parallel inference engine techniques mentioned above, with the emphasis on OR parallelism. Variations of the techniques discussed in Chapter 20 are also in order.

Next, even when only one solution is desired, it is possible to obtain increased parallelism by specifying that it "doesn't matter" which solution is found. For some goal, any clause that unifies with it and has its right-hand side satisfied is acceptable. This permits a set of clauses that might match a goal to be executed in parallel, with the first one to finish successfully being the one that provides the answer. This is in contrast to PROLOG, which "doesn't care" which solution is returned, as long as the order in which potential solutions are generated matches that specified by the order of the clauses as written by the programmer. While OR parallelism is clearly important here, efficient AND parallelism is more essential because of the emphasis on one solution. Such languages are termed ***nondeterministic*** because one cannot determine beforehand which solution will be returned, and consequently different executions of the same query can provide radically different answers.

The efficiency of such nondeterminism can be further enhanced by permitting the programmer to specify conditions under which to ***commit*** to a particular clause in advance of completing the entire right-hand side. Such a specification is called a ***guard,*** and is similar to a ***cut*** in PROLOG in indicating that as soon as all the literals to its left are satisfied, no other clause need be investigated for the same goal. Languages with these features are termed ***committed-choice, nondeterministic languages.***

Finally, inclusion of ***streams*** in logic languages offers opportunities for AND parallelism by permitting two goals that share a variable (and thus would be constrained from normal AND parallelism) to execute concurrently as soon as that variable is bound to a partial expression, even if that expression has within itself variables that are not bound until later.

21.2 VARIATIONS OF OR PARALLELISM

(Crammond, 1985)

OR parallelism is at heart an attempt to speed up the search for a clause or clauses that satisfy some goal. As discussed previously, it is useful in two situations: where all possible solutions are desired, and when it doesn't matter which solution is found first. In either case the main problem that must be addressed is how to handle the multiple bindings that might develop, particularly ***output substitutions*** that affect variables in the goal or its parents. This devolves into three major implementation problems:

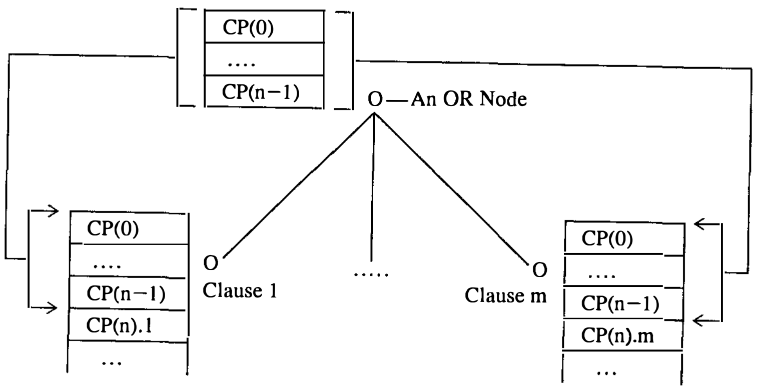
- How to record the different bindings of successful solutions
- How to create and access variable bindings so that there is no interference between separate OR processes
- How to create new OR parallel tasks

The first problem is usually solved by the programmer by designating a data structure such as a list to capture the solutions. The latter problems are more important, and will be addressed here.

We will assume for now that there is no AND parallelism; literals in the body of any clause are executed left to right just as in PROLOG. We will also assume that something like the WAM model is used as a target.

21.2.1 The Abstract Model

The simplest way of keeping bindings straight is to maintain them, and all their surrounding contexts, in separate copies. Figure 21-4 diagrams this. When a goal is encountered which can be solved with OR parallelism, the inference engine can simply make a unique copy of the entire choice-point stack for each processor that participates in the operation. Each such processor can then augment the stack with a choice point for its own clause and for any clauses that need to be solved as a result. When each processor completes, the values in its root choice point (the one for the query) represent a solution to that query.



Assume CP(i) is choice point for i'th inference.
CP(0) is the original query. CP(i).j is j'th versions of i'th Choice Point.

FIGURE 21-4
OR parallelism by copying.

COMPLICATIONS. Although the concept is simple, there are a few subtleties here. First, variables in the WAM exist not only in the stack but also in the heap, which means that it also has to be copied. Second, unless each processor has its own memory space, which is allocated exactly like everyone else's, special consideration has to be given to the representation of variables. Consider, for example, a simple copy of a choice point from one area of memory to another at a different address. If we use the representation for variables discussed earlier for the WAM—namely, that an unbound variable has as a value its own address—then in this copy we will not have new versions of variables from the original one. Instead we will have direct references to them. To fix this requires either a detailed examination of each word before it is copied or changing the form of an unbound variable to an index which, if added to the base address of the stack, gives the address of the cell. The latter is by far more efficient.

Finally, special consideration has to be given to backtracking. A choice point created as a result of an OR process [CP(n).i in Figure 21-4] must not be allowed to backtrack into CP(n−1) if it fails. The only one that should do this is the one that represents exactly the last possible clause to try when all previous ones have failed. All other OR processes that fail the clause they were established to try should simply terminate and release their storage. This requires special code in the backtrack process, plus synchronization information somewhere. The typical form of the latter is a single counter for each OR node of its outstanding child processes. This is decremented when such a process terminates because of a failure. Such a counter is often kept in the original choice-point stack, with pointers to it in the ones that represent copies.

OPTIMIZATIONS. The obvious inefficiency with this approach is the hugh amount of copying that goes on. There are a few optimizations that can reduce this. First, if care is taken, we need not copy the original stack for the leftmost OR clause in Figure 21-4. Instead the new choice point [CP(n).1], plus the synchronization information, can be built right on top. Only the other OR processes need to be copied. This stack might be called the ***favored alternative***.

Next, we can also avoid copying the entire stack and heap at some increase in complexity. For each argument generated by the topmost choice point for the call to the OR parallel set of clauses, the inference engine can trace it out and make a separate copy of everything (and only) that which it references. This is equivalent to a two-step unification process. First, the actual argument is unified with a dummy variable in the new OR process environment. This should run in ***read mode*** in the WAM mode, with a copy of everything in the argument reconstructed in the new processes' own heap. The kicker here is making sure that all references to the same variable in the goal's arguments are translated to references to a new common variable in the new processes heap. This often requires maintaining some kind of translation table between goal variables and already-created heap variables.

After this intermediate step, a set of conventional WAM unification instructions can unify these copies with the values expected by the clause.

With this approach there is far less copying to be done and absolutely no way one OR process can even touch the original variables in the caller, let alone cause conflicts with the other processes. Thus, when a process terminates unsuccessfully, all that need be done is signal the parent and free the storage. However, there is now the problem of how to handle the successful case, since all the answers are in the child processes' memory. The best approach to this is to perform yet a third unification process, this time between the arguments copied by the intermediate unification step above and the original arguments of the call. Again care must be taken to record when variables in the goal have been given values, so that this solution does not interfere with some possible future solution.

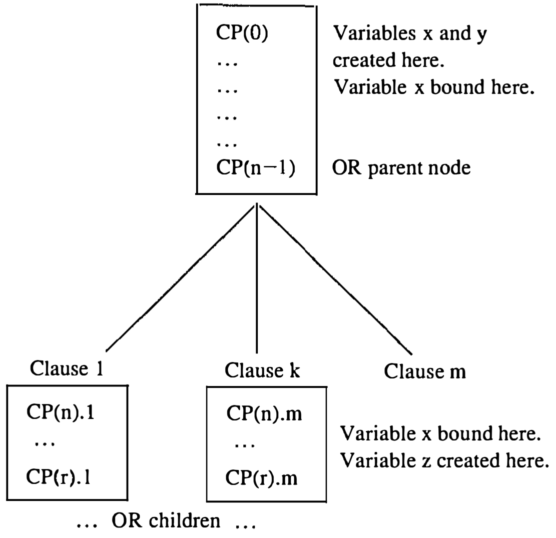
COMMENTS. While this copying process may seem grossly inefficient, there are cases where it may not be too bad. One is in a case where the parallel machine is a distributed one (such as a ***hypercube***) where there is no shared memory, only message-passing interfaces between processors. The other (suggested by Seif Haridi of the Swedish Institute of Computer Science) is where the parallel processors have their memories interconnected by a shared bus over which ***broadcasts*** of data to several memories can be accomplished at the same time. This latter is a decent match to several parallel processors that are commercially available today.

21.2.2 Conditional Bindings and the Naive Model

A better approach to this problem of sharing and accessing bindings is to identify just those unbound variables that may be shared among several OR processes and do something special about them.

The identification part is conceptually easy; we assume that at an OR node each OR child starts its own choice point, heap, and trail. A pointer in the topmost choice point so created refers backward to the parent OR node (Figure 21-5). Now any reference to a variable that is on the upper side of this pointer is to a variable that is potentially shared among the OR children. A reference to a variable that occurs below this reference is to one's own stack. Variables of the second kind are handled exactly like variables in the original WAM model. They were created by, and are referenced only by, the process that built the new stack segment. These are often called ***private variables.***

What a process does with a variable of the former kind depends on how the variable received its binding. If the OR parent (or one of its parents) has already bound a value to a variable, then all the children OR processes should see that same binding. These are called ***unconditional bindings.***



Variable	Status in OR Children
x	Unconditional—created and bound in parent.
y	Conditional—created in parent, bound in child.
z	Private—created and bound in child.

FIGURE 21-5
Classification of variables.

If a variable was created in the parent leg of the stack but was left unbound when the split into parallel OR processes occurred, then any attempt by an OR child to bind a value to it must be handled specially. Each OR child should see only its bindings and not anyone else's. Such bindings are called ***conditional bindings,*** because the values returned by a reference are conditioned by the referencer.

There is a slight problem with making a determination of which side of the parent–child boundary a variable is. A simple address comparison as was done for the trailing tests in the WAM is not always accurate because of the treelike nature of the stack. Individual choice points of the stacks may be scattered all over memory, with no ordering. This would be particularly true of a program after very many OR nodes had been created and then discarded.

A simple solution to this would be to associate an extra word of storage with each variable. This word would have in it some information about when the variable was created (e.g., by an allocate instruction). This could be the time of day when it was created, the depth of the stack in choice points, or something similar. Each process then also keeps with it a value in a compatible form indicating the time when its stack split off of an OR parent. Now whenever a variable is referenced, a comparison of its creation time with that of the process would reveal if it is private or not.

With this approach, both private and unconditional bindings are stored exactly where they would be in the WAM. Only the bindings for the conditionals are different. In the ***naive model*** of OR parallel implementation they are stored in a somewhat strange spot—the trail. As with the stack, each OR child gets its own trail, which is bound at its root to the top of the parent's trail. Now, whenever a binding to a conditional variable is to be made by a child process, instead of modifying the address in memory allocated to that variable, the address and the new value are stored as a pair in the processor's local trail. With this assumption, a dereference of a variable would proceed as in the following:

1. If the final address of the variable points to a private variable, the value in that memory location is the current binding.
2. Otherwise, search the pairs in the current processor's trail for one whose address component matches.
3. If a match is found, this was a conditional variable. Return the value in the pair.
4. If a match is not found, check the original variable's location in the parent's environment.
5. If that variable has a binding, use it (it's an unconditional binding).
6. If it is unbound, then the variable is unbound for the child also.

This approach clearly saves copying, but at the cost of an unpredictable amount of time for accessing a conditional variable. This means that when there are a lot of bindings to conditional variables, the time required to access them can grow arbitrarily long. In contrast, the time to access a variable in the WAM, once it has been fully dereferenced, is approximately constant (one memory access).

FAVORED BINDINGS. Just as was done with the abstract model, it is also possible to give one OR child ***favored binding*** status. Basically, one OR child (usually the first one established) is given an indication that it is special, and may make bindings directly into conditional variables in its parent. All that is needed is an extra tag bit in each value word which indicates whether the binding recorded is from a private binding or a favored conditional one. In the latter case, any reference to that variable from a different OR child will use the tag to indicate that the binding is not an unconditional one, and is thus not for it. In such cases these other OR children will search their trails and make conditional bindings as before.

This makes the accesses by the favored child constant at a slight increase in access time for everyone else.

21.2.3 The Argonne Model

(Shen and Warren, 1987)

A variation of this naive approach is to keep the conditional bindings not on the trail but in a special ***hash table*** or equivalent. This table, as pictured in Figure 21-6, would be associated with each OR child, probably with the first choice built after the division. Now, instead of placing bindings to conditional variables on the trail, the address for the original variable would be hashed and the hash value would be used to access the hash table. The combination of address and binding would be placed on a list associated with that entry.

The effect on performance can be quite significant. If there are n bins in the hash table, then the average time to search it is $1/n$-th that of the naive model. The cost is more complexity in memory management.

Simulation experiments seem to indicate that when the overhead for task switching is not excessive, a factor of near n speedups may be achieved with n of up to 10 or more. Increasing the overhead to perhaps four times the cost of a call degrades this significantly, to the point where a ceiling of perhaps 4 to 1 in speedup is reachable.

21.2.4 The SRI Model

(Warren, 1987)

If the problem with the above approaches is due to sharing environments, then perhaps one solution starts with explicitly fracturing the environment in the WAM from the choice-point stack and heap, and keeping all variables in a ***Processor Binding Array*** (***PBA***) associated with each

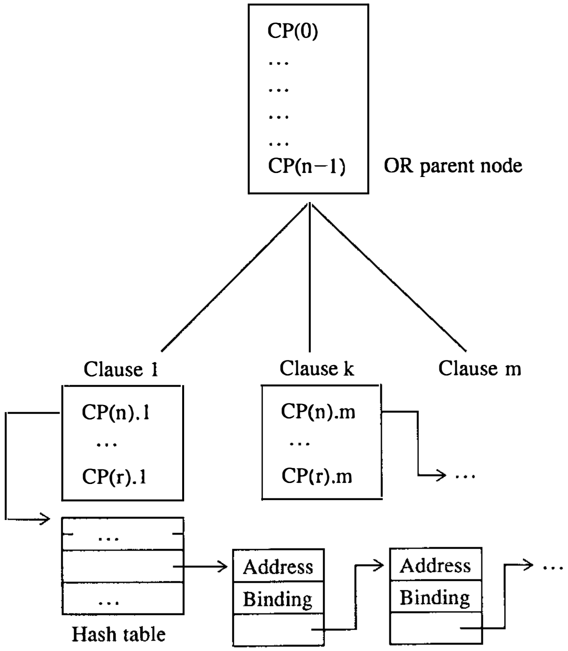


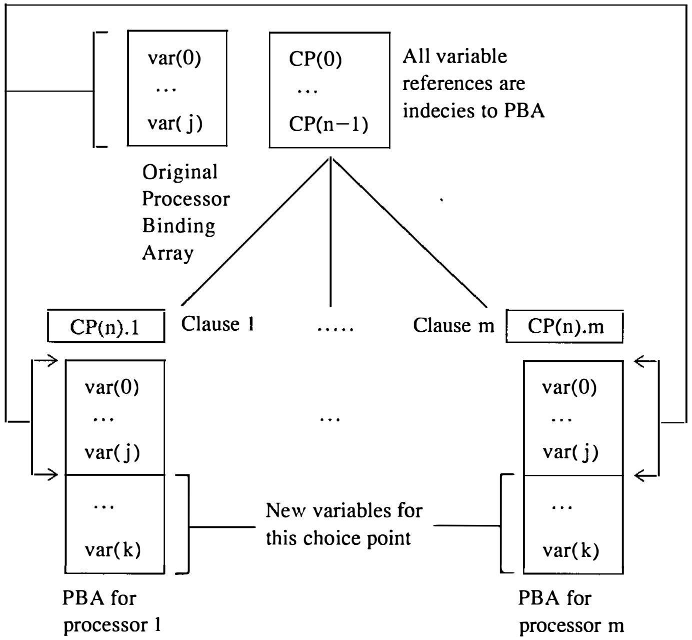
FIGURE 21-6
The Argonne model.

processor (Figure 21-7). All variable "values" in the choice point, the heap, or the trail are indices to entries in the PBA. All real bindings to variables are then recorded in the appropriate entries of this table. Performance cost of either a binding or a dereference is thus only slightly slower than that for conventional PROLOG. One extra indirect reference and an add to the PBA origin is needed per variable access to use the index found in the choice point as a pointer into the current processor's PBA. No search as in the naive model is needed.

With this approach, when an OR node is reached in a computation, only the current PBA need be copied to each new processor. Each processor associated with a new clause will build its own choice point for that clause and augment its own PBA with space for any new variables that it may need to create. The new choice points need only point backward to the prior ones and need copy none of their material.

As before, an individual processor will not backtrack beyond the choice point it created for the OR node unless it is the last such processor to fail. Also note that at a successful completion, the PBA contains (probably in its earliest entries) the bindings for the query variables. No back-unification is needed to recover them.

In the literature the improved form of this general approach is called the ***SRI model,*** after SRI International, the research establishment in California where David H. B. Warren was working when he first proposed it.

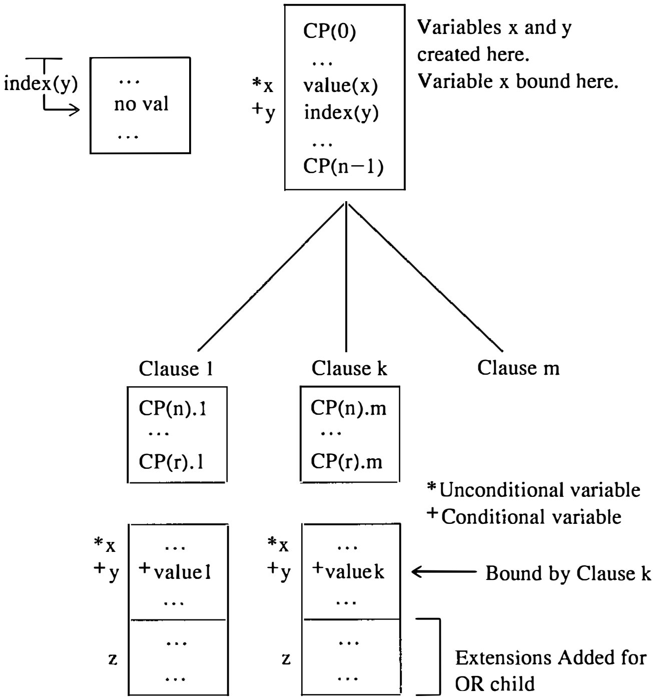


Assume CP(i) is choice point for i'th inference.
CP(0) is the original query.

FIGURE 21-7
Simplistic SRI model.

VARIABLE ACCESS OPTIMIZATIONS. There are several possibilities for minimizing overhead associated with copying and accessing the PBA in this model. First, as mentioned above, the ***favored processor*** that was executing the OR node at the beginning need not copy its PBA. It can simply augment it and keep going.

Second, and most important, not all variables need be bound in the PBA. Even though storage might be allocated for them in the PBA, private and unconditional variable bindings could be kept in their normal locations in the choice point and heap. Consider Figure 21-8 where there is exactly one OR parallel branch. All the choice points before it were built and executed sequentially as in conventional PROLOG, and each branch after it similarly acts sequentially. As in Figure 21-5, the two variables x and y are assumed to have been created in choice points before the OR branch, and z is assumed to have been created in one of the pro-



Note: .... indicates largely unused storage.

FIGURE 21-8
Complete SRI model.

cesses after the branch. *x* receives a binding before the OR; both *y* and *z* receive bindings after. Thus *x* is conditional, *y* is unconditional, and *z* is private.

Making this happen requires some additions to the WAM instruction set. When a variable is initialized for the first time, as in a get-var, a new cell is allocated to it in the PBA, and the index to that cell is saved in the choice-point definition (along with a special tag to that effect). When a binding is made, as in a get-val, if the binding is unconditional or private, then the value is saved in the original choice point, overwriting the index. If the binding is conditional, the index is used to access into the current processor's PBA and store the value there. Trailing must also be modified to include not only the address of a variable being trailed but also the original value, namely, the index to the PBA.

The net effect of all this is that accesses to bindings are almost as fast as in the original WAM, with at most an extra tag check, memory reference, and add to get into the PBA. In addition, the PBA can be in memory that is largely local to the processor owning it, meaning that accesses to it may be faster than to shared choice points, and, thus, in processors with caches, big chunks of the PBA can reside in the caches without fear of conflict with other processors.

SCHEDULING. One aspect of this whole process that has not been fully addressed yet is exactly how a processor decides where there are opportunities for useful work to perform and which should be taken. One could naively look around at all the open OR nodes for some node that still has unexplored clauses left to try, but this would require creating a complete copy of the PBA.

A better approach can be derived by assuming that we have expanded some OR node as fully as can be done with the processors that are available, and that one of these alternatives has just failed. If the normal WAM backtracking mechanism has been properly employed to this point, the PBA associated with this processor has been reset to exactly what it was when the alternative was started. Consequently, if there are any other OR alternatives that have not yet been tried [as recorded in CP(n−1) in Figure 21-8], then this processor can be dispatched to try it without any additional copying into its PBA.

If there are no further alternatives left to be tried at this node, but there are processors still active with other alternatives, the trail associated with CP(n−1) can be used to ***back up*** the processor's PBA to the choice point's parent, where a similar exploration for work can be performed. A failure to find work there could cause a backup to its parent, and so on. When work is finally found, only the minimal number of PBA entries have been reset to unbound, and no bulk copying has been performed.

Note that in this backup we *do not* reset the choice point's trail pointer. Nor do we unbind any unconditional variables in the choice point and heap. These will be left for the *last* processor to fail out of the OR node. We simply use the trail information to unbind the conditional variables specified by the trail.

Under many circumstances a simple backup as described above checks only for work between the current node and the root, and will miss opportunities for work that exist in other uncompleted OR nodes down other paths. Consequently, an elaboration of the above algorithm backs up one node [to CP(n−1)], and if there are no untried clauses there, looks to see if there are any still busy processors before backing up further. If there are, the processor can start down the choice-point stack associated with one of those processors, looking for OR nodes down that path. At each such transition downward, whatever additions were made to the active processor's PBA are copied to the "idle" processor looking for work. Again, if no work is found, we can continue to back up and go down other paths.

If no work whatsoever can be found, then the processor marks itself as completely idle. Then, whenever any other processor finds itself with a new OR node to expand, it can signal those processors in the idle pool. These in turn can copy the base PBA into their own, grab a clause alternative, and start off as before.

21.2.5 Version Vectors

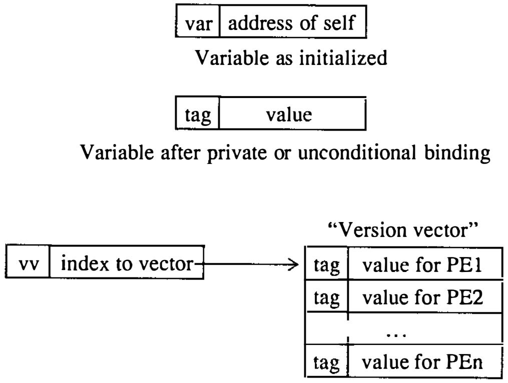
(Ciepielewski et al., 1989; Hausman et al., 1987)

An obvious problem with the SRI model is the large binding array associated with each processor. Given the optimizations possible with unconditional variables, many of these arrays are very sparse and become essentially wasted space; they are allocated and initialized, but they are never used.

An alternative called ***version vectors*** turns these arrays sideways. Rather than having each processor maintain an array with an entry for each variable, in the version-vector approach, each (and only) conditional variable has an array with an entry for each processor. This is implemented by tagging the original variable entry in the choice point or heap as a "version vector," and then replacing its value by a pointer to a vector of contiguous locations which is as long as there are processors in the processing system (Figure 21-9). Then, whenever the k-th processor accesses a variable that has a versions-vector tag, the associated version vector is accessed with an index of k. The net effect is a large savings in memory.

Virtually everything else in terms of optimizations and task scheduling mentioned above adapts naturally to this form. For example, the version-vector pointer can be kept in the word following the one containing the normal value. This permits a ***favored processor*** to find its bindings immediately, without concern for a version vector.

Perhaps the biggest difference in terms of implementation between



Variable as initialized

Variable after private or unconditional binding

Variable after conditional binding

FIGURE 21-9
Binding in the version–vectors model.

this and the pure SRI model has to do with synchronization. It is possible for multiple processors to attempt to bind a conditional variable at the same time. Depending on implementation, it is possible for each of these processors to believe that it is the first processor to attempt such a binding, and to go off and allocate a new version vector to it. The result could be chaos. Both wasted storage and lost bindings might result.

Preventing this requires some sort of ***lock-out*** between processors when a new version vector is to be allocated and installed. One implementation would be to allocate an extra bit in the tag field for synchronization. A processor sets this bit when it begins allocating a version vector, and resets it when the allocation and initialization are complete. Any other processor that wishes to dereference or bind to a variable must wait if it finds the bit set. This is not necessary with the other models, and could be a significant performance penalty when the number of processors is large.

Finally, the memory reference patterns from the different processors are different from the SRI model. For the latter, most variable references end up in an area of memory dedicated to that processor, which tends to foster good cache characteristics. With version vectors, all such references to the same variable from different processors are to the same area of memory. This may cause performance-degrading interference and cache thrashing, which has to be balanced with the savings on memory and copying.

21.3 VARIATIONS OF AND PARALLELISM

(Hermenegildo and Rossi, 1989)

AND parallelism occurs when two or more goals from the right-hand side of a clause or query are executed concurrently. Although certainly usable in an all-solutions context, AND parallelism (and particularly the stream AND form discussed later) is also helpful when only one solution is desired. In fact, a major variant of AND parallelism, ***stream parallelism,*** is an integral part of a new class of languages designed explicitly for exactly-one-solution problems.

21.3.1 Consistency Checks and Join-Based Parallelism

The major problem with implementing OR parallelism is handling the bindings and keeping them separate until solutions are found. With AND parallelism, the problem is almost exactly the opposite. Without care, it is possible for AND parallelism to run amok and actually take much longer than a sequential solution. The problem lies in the need to do ***consistency checks*** among the solutions that come back from the parallel AND goals, and make sure that the same bindings are applied to all occurrences to all variables. Only when this is done can an answer be propagated back.

As an example, assume that a parallel AND solver has been given the goal **p**(x, Hitech, Tim), and has chosen to expand the following clause in parallel:

$$\mathbf{p}(x,y,z){:}-\mathbf{p}(x,y,q),\ \mathbf{r}(y,q),\ \mathbf{s}(q,t),\ \mathbf{s}(t,z).$$

In a simple implementation, each of the four parallel processes would send back potential solutions for their specific literal consisting of substitutions for x and q, q, q and t, and t, respectively. Assume also that these solutions come back one at a time. The AND process will take the four sets of bindings and look to see if they have the same t and q values. When it finds such matches, it assembles the corresponding x as a solution to its parent. If there is no match, it must remember all four combinations of bindings and pick one or more literals for a backtrack to a new solution, if one exists. When these new solutions come back, they are again compared—both with themselves and with all the prior solutions.

While this would seem to maximize the potential parallelism, in reality it can have a killing effect on actual computational efficiency and total system throughput. Basically, the problem stems from the fact that no child process ever "sees" any potential solutions generated from other child OR processes and thus cannot "zero in" on potential solutions that have a real possibility of participating in the final consistent one. Further, there is no guarantee that the order of solutions from one child process has any resemblance to the order of solutions from another child, meaning that the parent AND process must remember ALL the solutions it receives from each child, and as new solutions come from other children, compare the relevant parts of this new solution with each of the previously stored ones.

This can cause an incredible explosion in storage and in computation time to perform all the comparisons. Figure 21-10, for example, gives a case where there are a huge number of potential solutions to the first literal (of which perhaps 100,000 are immediately available from a database), 100,000 solutions to the second, 1 billion to the third, and 2 to the fourth. If we solve each in parallel, Murphy's Law will guarantee that the entries we want are the last ones generated, by which time a potential of 2×10^{19} different triples of solutions will have been generated and checked for consistency. If each consistency check takes only 1 ns, the process would require a mere 6340 years.

For all practical purposes this approach is equivalent to computing the relations for all the goals in the query independently of each other, and then doing a relational ***join*** as described in the last chapter, but without the benefit of indices, hash tables, or the like. It is clearly counterproductive. AND parallelism will be of benefit only if the size of the joins can be reduced or eliminated. The following subsections describe several variants that attempt to do this.

21.3.2 Pipelined AND

Assume that, through compiler analysis or programmer notation, we know which goals are liable to require the smallest amount of computation under various circumstances of input arguments. Further, assume that we order the goals as shown in Figure 21-11, with each of the **Qi**s representing one of the goals left to solve, in the above order. Now assume that we start a single processor solving **Q1** and tell it to continue computing solutions as long as possible. The binding for the variables in the arguments of **Q1** are captured in tuples and stored in a buffer of some sort (queue, FIFO, etc.).

Clause given AND process:
p(x,y,z): – p(x,y,q),r(y,q), s(q,t),s(t,z).
with substitutions: Hitech/y, Tim/z.

where:
- p(x,y,z) = person z pays tuition x to attend university y.
- r(y,t) = person t attended university y
- s(t,z) = person t is parent of z

The clause states that a child will pay the same rate as his/her grandparent for a particular university, if that grandparent went to school there.

Assume that:
- 100,000 people have attended Hitech University,
- There are 1 billion parent relations,
- Peter's father paid $1000 to attend Hitech,
and Peter is a parent of Tim.

Total # of triples to check = $10^5 \times 10^5 \times 10^9 \times 2 = 2 \times 10^{19}$

FIGURE 21-10
Explosive growth in AND parallel computation.

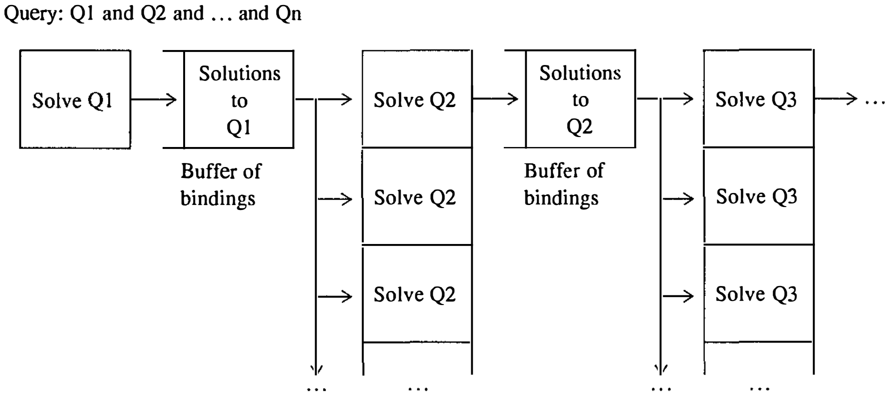


FIGURE 21-11
Pipelined AND parallelism.

As soon as a new binding solution from **Q1** enters its associated buffer, we have an opportunity to start a processor to try to solve **Q2** assuming the bindings just computed. It computes an extended set of binding and deposits those in its output buffer. This process repeats until the final **Qn** is computed. The answers that leave it represent consistent bindings to the whole problem.

Depending on the number of processors available in the system and how long the chain is, we have two options each time a new solution enters a buffer. Either a new processor can be started with a new copy of the code for the **Qi** or the solution can be left in the buffer until the current processor completes its current solution and is free to start another. Figure 21-12 diagrams a simple case. With a standard PROLOG inference engine running on one processor, the computation of the four solutions of **Q3:** D, E, I, and K takes 11 time units. The first solution of **Q4** (assumed to require binding K) appears at time 12.

If three processors are available and are allocated to each of the four goals, the time drops to 8 time units for this first **Q4.** Note that the time to the first **Q3** (binding D) does not change. If a processor is available every time some other processor produces a binding, then the time for K time drops to 6 time units and gets even better as the number of solutions increases.

The reason for this speedup on the first solution in the final case is that the system kicks off very early a set of computations based on the second solution from **Q1,** namely, G, long before it discovers that the first solution leads nowhere. This early start is where the real payoff comes from, and in fact can make an AND parallel solver more ***complete*** than regular PROLOG. Consider what would happen in regular PROLOG if the first solution A from **Q1** generated an infinite number of

Assume
Query = Q1 and Q2 and Q3 and Q4 solved in that order.
Each computation of a binding takes 1 unit of time.
Backtracks take 0 time.

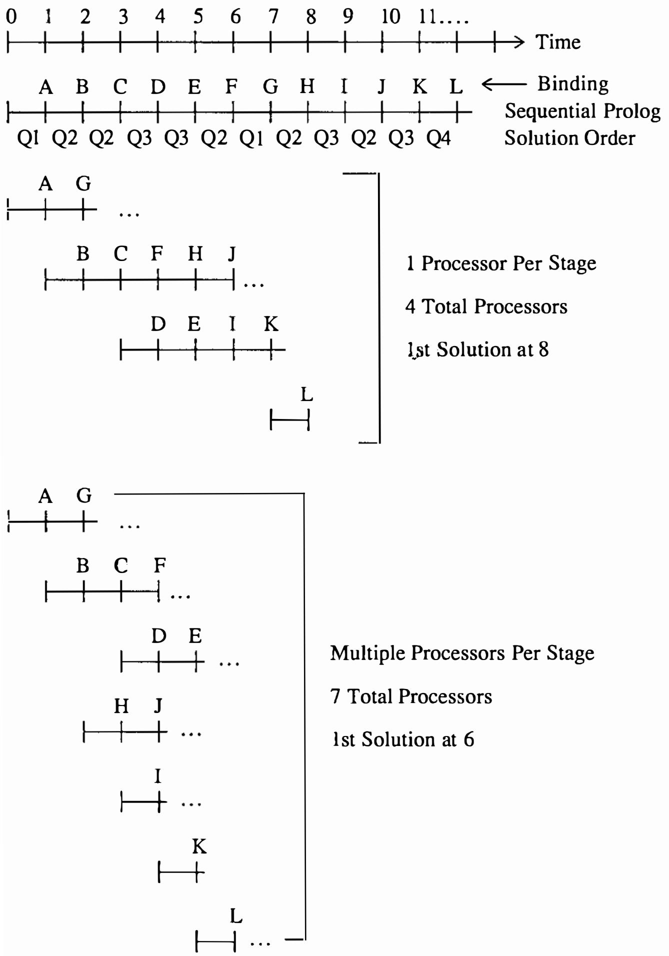


FIGURE 21-12
Sample AND parallel timings.

solutions from **Q2,** or got stuck in a recursive loop. The second solution would never be generated, even if it were trivially computable.

21.3.3 Dataflow AND Parallelism

Pipelined AND parallelism clearly helps with speeding the computation of multiple solutions to a query, but it is of less general use in speeding first (or single)-solution problems. The problem is that the way cross-checks are handled is via essentially a sequential flow.

The solution to obtaining more speedup without the risk of expensive joins is to run in parallel only those goals which are guaranteed not to interfere with each other. The computation is still pipelined between parallel sections of such goals, with minimal synchronization between sections needed only to guarantee that binding sets are in fact of a form that permits independent parallelism. Because of the arrangements of such systems, this model is often called ***dataflow AND parallelism.***

There are two kinds of goals that can be considered for parallel execution within a section:

- Goals that make no bindings whatsoever (called ***checker goals***)
- Goals whose output substitutions can be identified in advance (called ***generator goals***)

Checkers nearly always expect all their arguments to be ***ground terms*** with no embedded unbound variables. As a result, their effect is either to accept a binding as valid or to mark it as one which should be deleted from further consideration. Such goals can be run in parallel with any other goal as soon as the arguments are bound. Handling their response is limited to discarding from further consideration any binding that fails their test.

Generators have variables, usually in specific argument positions, that after execution are known to receive bindings. Determination of which arguments will be so generated can be done by either compile-time analysis or more frequently by programmer ***annotations*** on the ***modes*** of a predicate's arguments (see Chapter 18). An argument marked with an ***output mode*** is one that will receive a binding after a predicate's execution.

Generators can be run in parallel with other generators whenever the variables they bind are all different and they have values for all the argument positions marked as input positions. When a set of them is started in parallel from a single binding, the output will be a set of bindings consisting of the original set cross-produced with all possible combinations of the individual bindings provided as answers. For example, consider the query **p**(x,y), **q**(v,x,z), **r**(v,t). Assuming that variables x and v receive the binding [1,2/x,v], and that **p** generates bindings of 3 and 4 for y, **q** generates 5 for z, and **r** generates 6 and 7 as solutions for t, then running all three in parallel is acceptable. The synchronization at the end assembles possible bindings of solutions of the form {(1,2,3,5,6), (1,2,4,5,6),(1,2,3,5,7),(1,2,4,5,7)} for the variables x, v, y, z, and t, respectively.

The ***data dependency analysis*** algorithm from Chapter 18 can be used to compute which literals in a query are generators (and when), which are checkers, and which cannot be determined. Figure 21-13 gives an example consisting of eight calls to the predicate **q.** The mode of the first argument is input, and the mode of the second argument is both input and

Query: q(u,v), q(u,w), q(u,x), q(v,w), q(w,x), q(v,y), q(w,y), q(x,y).

Modes: q(+, ?).

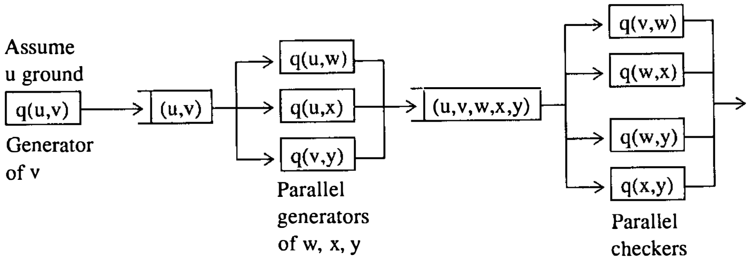


FIGURE 21-13
Dataflow AND parallelism.

output. The variable *u* is assumed to be bound at the start with some constant.

With these assumptions the leftmost goal must be done first. Any solutions it generates will have bindings for both *u* and *v*. Given these bindings, the literals **q**(*u*,*w*), **q**(*u*,*x*), **q**(*v*,*w*), and **q**(*v*,*y*) all become generators. Not all of them, however, can be executed simultaneously without potential conflict. The three literals chosen in Figure 21-13 represent one possible combination. If they run in parallel from the same (*u*,*v*) binding, they compute sets of bindings for *x*,*y*, and *z*. All combinations of these are joined with (*u*,*v*) (note: no consistency checks are needed) to produce bindings for all five variables. Since all these variables now have values, all the remaining four literals are now checkers and can run in parallel. Each receives the same binding set from the prior stage, and if all four of them pass the binding, it is a valid solution.

As in the last section, there is nothing that demands exactly one set of parallel processors to be allocated to a parallel section. If additional processors are available, each time a new binding becomes available at the input to such a section, a new set of parallelism can be started. The only constraint is that the processes executing in that section synchronize with each other before passing their final solutions on. This also requires careful synchronization at the input to the solution buffers shown in Figure 21-13; there may be quite a few sets of processors competing simultaneously to insert binding sets. Storage demands for all these partial solutions may also become quite high.

21.3.4 Restricted AND Parallelism

(DeGroot, 1984, 1985, 1987; Chang et al., 1985)

It is often impossible to predict at compile time the exact dataflow that all queries or clause right-hand sides might have. Either the information on bindings or appropriate mode information is simply not available. Thus, if AND parallelism is to be implemented, some sort of dataflow analysis must be done at execution time. Needless to say, a full-fledged analysis each time a goal is called is not only unbearably expensive, but also almost demands that the code be reinterpreted each time. This negates the advantages of tightly compiled code as we saw with the WAM and would seem to make AND parallelism of academic interest only.

A way out of this dilemma was invented by Douglas DeGroot in 1984, who developed a compile-time analysis that simply determines the set of "possible" dataflows that a clause's right-hand side may take on as a function of its argument bindings, and then produced relatively tight compiler code that at runtime determines which possibilities are applicable. To keep the runtime costs low, these tests are simple, but they sometimes err on the side of excluding parallelism when it might have been possible. After several years of experimentation, this restriction seems to be more than acceptable, and consequently this form of AND parallelism, termed ***restricted AND parallelism,*** is becoming the dominant approach.

To get an idea of what is involved, consider the following clause:

$$\mathbf{p}(x){:}-\mathbf{q}(x),\mathbf{r}(x),\mathbf{s}(x).$$

With no prior knowledge about modes of *x*, we cannot determine which parts of the right-hand side might be generators or checkers. However, suppose that code was included just as the body of the clause was entered to see if *x* was bound to a ground term or was still an unbound variable. If the former, then we could branch to code that specifies that all three literals may be executed in parallel (as checkers). In the latter case, there are no such guarantees and in the absence of further information no parallelism is possible. Assuming a PROLOG left-to-right order, we then generate code to execute **q**(*x*) in isolation. However, rather than then computing **r**(*x*) and **s**(*x*) in a sequential fashion, we can follow this call with a test similar to the first one. If *x* is now still unbound after a successful call, then there is nothing to do but execute the other two sequentially. If, however, *x* is bound to a ground term, then at least some parallelism can be rescued, and **r**(*x*) and **s**(*x*) can be executed as parallel checkers.

Although not all the potential parallelism has been captured, a great deal of it has, and in a relatively simple manner. Figure 21-14 gives a bigger example, where any one of five possible dataflow graphs might be chosen at runtime from the body of a clause. Note also that the pseudocode shown for the body's execution does not have anywhere near five times the amount of code of a single case. Most of the parallelism tests are cascaded.

The two functions ***ground*** and ***independent*** are shown in Figure 21-14 in a LISP-like format. Each has one argument which is a list of variables. Two or more expressions follow this list. In both cases a test is per-

```
p(x,y):-r(x), q(y), s(x,y), t(y).
Lisplike Pseudo code for body:
(ground (x y) (independent (x y) (r x) (q y))
              (ground (x y) (s x y) (t y)))
where:
  (ground a b c) =
    if all variables in a are bound to ground terms
    then do b and c in parallel
    else do b and c sequentially
  (independent a b c) =
    if all variables in a are either bound to ground terms or are
    all different unbound variables
    then do b and c in parallel
    else do b and c sequentially
```

Test	Result				
ground	yes	no	no	no	no
independent	yes	yes	yes	no	no
ground	yes	yes	no	yes	no

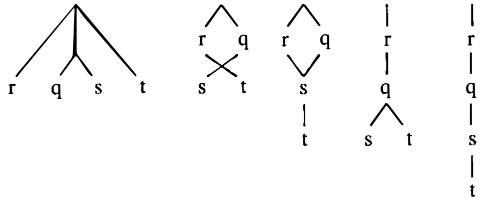


FIGURE 21-14
A larger restricted AND parallel example.

formed on the list of variables. If all pass, then all the subexpressions may be executed in parallel. If a test fails, then the subexpressions are executed sequentially from left to right.

The **ground** predicate is true only if all the variables in the list are bound to arguments which are totally ground—there are no embedded unbound variables. This can be a relatively simple test. In terms of a WAM-like machine, it involves dereferencing and testing the final tag. On any kind of a constant or unbound variable, the result is immediate. For a structure or a list, a thorough test would require dereferencing all its components. In most cases we opt for simplicity and simply declare these as nonground. While this will miss some parallelism, the gain in efficiency is usually worth it.

The typical implementation of **independent** is slightly more involved. It is true only if all the variables in its argument list are guaranteed to be independent, that is, no unbound variables are shared between any of them. Assume for simplicity that the argument list being tested has exactly two variables. If either one of them dereferences to a simple constant, then the pair are automatically independent. If both dereference to unbound variables, and the addresses are different, then they are independent. Equal addresses means that the variables are dependent.

As with **ground,** the difficult case is if either of them are bound to lists or structures. A rigorous test would look at each component of one and compare each unbound variable found there with all the unbound variables found in the other. This can get quite complex, especially for large structures. Instead, as before, for this case the simplest assumption is that they are dependent, and thus the subexpressions following the list are to be executed sequentially.

21.4 AND/OR PROCESS MODEL

(Conery, 1987)

One of the most exciting research directions today is in building language execution models (inference engines) that include both AND and OR parallelism in significant amounts. One of the most popular such models is the outgrowth of a Ph.D thesis by John Conery at the University of Oregon. It goes by the name ***AND/OR process model.***

In structure this model attempts to allocate a separate process for each possible node in the AND/OR tree for some problem, and then permits the concurrent execution of as many of these as possible at the same time (cf. Figure 21-15). Thus, the model can be used to compute either

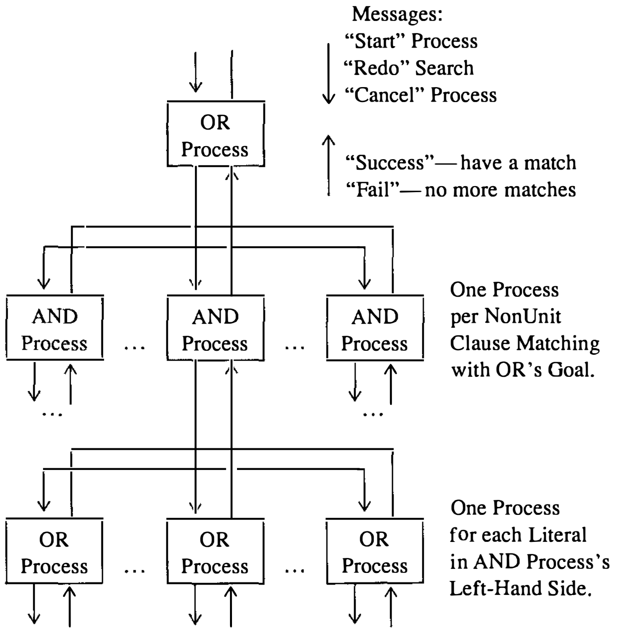


FIGURE 21-15
AND/OR process interactions.

one or all solutions to a query. The following subsections give a brief overview. Some liberties have been taken to simplify the presentation, but the general effect is in agreement with Conery's work.

21.4.1 OVERVIEW

In this model, an ***OR process*** corresponds to an OR node, and has the task of finding solutions which prove a single goal literal passed to the OR process by its parent. The process controls the simultaneous execution of a set of ***AND processes*** below it, each allocated to a separate clause whose head literal unifies with the OR's goal. In turn, each AND process attempts to find solutions which are consistent with all the literals on the right-hand side of the clause passed to it by its controlling OR process. This is done by spawning OR processes for each literal in the antecedent, and controlling them in a fashion which guarantees consistent solutions.

A key point about this model is that there is no centralized control to bottleneck the system. Instead, each process is in control of the processes below it in the tree, but only to the extent of creating them and scheduling when it is ready to receive a solution from them. Each process assumes that its parent will eventually want all possible solutions, and thus tries to keep busy as many nodes below it as possible. All communication between processes is through a standardized set of messages.

Although the model discusses only AND and OR parallelism, parallelism in the unification of individual literals is also possible, with no conceptual change to the basic description given here. Thus, with one exception, the model is an excellent representation for all possible sources of parallelism in a logic program. The one exception is in the AND process which handles multiple literals on the right-hand side of a clause. Even though multiple OR processes may be executed below it, the description here keeps them all focused on one solution at a time. The extension to handle multiple simultaneous solutions is much more complex and may not be worth the complexities in control needed to keep track of them. The section entitled ''Result Caching and Other Extensions'' will address this more fully.

21.4.2 Internal Process Details

Each process in this model is an independently executing program which communicates with other such processes via messages. At any point in time a process is in one of several states which control how it responds to incoming messages. Over time, a process will transition to and from these states in a well-defined fashion, with the transitions determining what messages to send out.

In addition, each process has some unique internal storage where it keeps not only an indication of what state it is in, but also who are its parent and children, what input information it received from its parent, and what solutions or partial solutions it has so far received from its children. These solutions are usually in the form of substitutions into variables in the literal or clause that the process is tasked to solve.

The following subsections describe the generic states and message types; details on internal storage are left to the discussions on the two process types.

PROCESS STATES. Each process is created by some single parent process, and may be terminated either when its processing is complete or under command of its parent. While active, it may be in one of three states (see Figure 21-16):

- Idle
- Waiting
- Gather

A process is in the ***Idle state*** only just after creation, and only until the parent has sent the process the necessary startup information.

A process is in the ***Waiting state*** when its parent has told it that the parent is now waiting for a solution from it, and that the child should respond with one as soon as possible. This state is entered after initialization from the Idle state, and after receipt of an appropriate command from the parent.

A process is in the ***Gather state*** when the parent is busy processing a solution handed it earlier by the process. In this state the child is free to try to find one or more additional solutions, so that it can respond rapidly the next time a parent asks for another solution.

INTERPROCESS MESSAGE TYPES. Transitions between states occur when messages are exchanged between a parent and a child. There are five such message types:

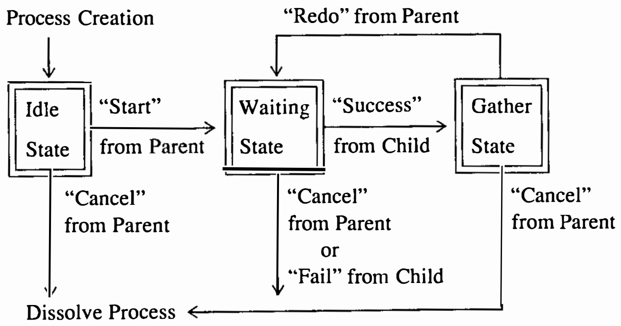


FIGURE 21-16
Generic state transitions.

- Start
- Cancel
- Redo
- Success
- Fail

The first three types go from the parent to the child process, and the latter two are responses from a child to its parent. The ***Start message*** signals a process that it is to transition from the Idle state to the Waiting state and includes the data needed by a child to start execution. In this Waiting state, the child is expected to respond as soon as possible as to whether or not it has found a solution. If a solution exists, the child responds with a ***Success message*** which includes the substitutions mandated by the solution it has found. If no solution exists, the child responds with a ***Fail message*** and dissolves itself.

This success message can be sent only when the child is in the Waiting state, i.e., when it knows the parent is waiting. Once such a message is sent, the child transitions to the Gather state, where it may continue to look for other solutions but will not forward them. The only time a child forwards any of these additional solutions is if the parent sends (at its leisure) a ***Redo message*** which commands the child back to the Waiting state.

The ***Cancel message*** may be sent by a parent to a child at any time, and causes that child to terminate both itself and all its children. This is used when a parent determines that no more solutions from the child would be useful.

The same message protocol is used by a child process in communicating with its children, although here the roles are reversed.

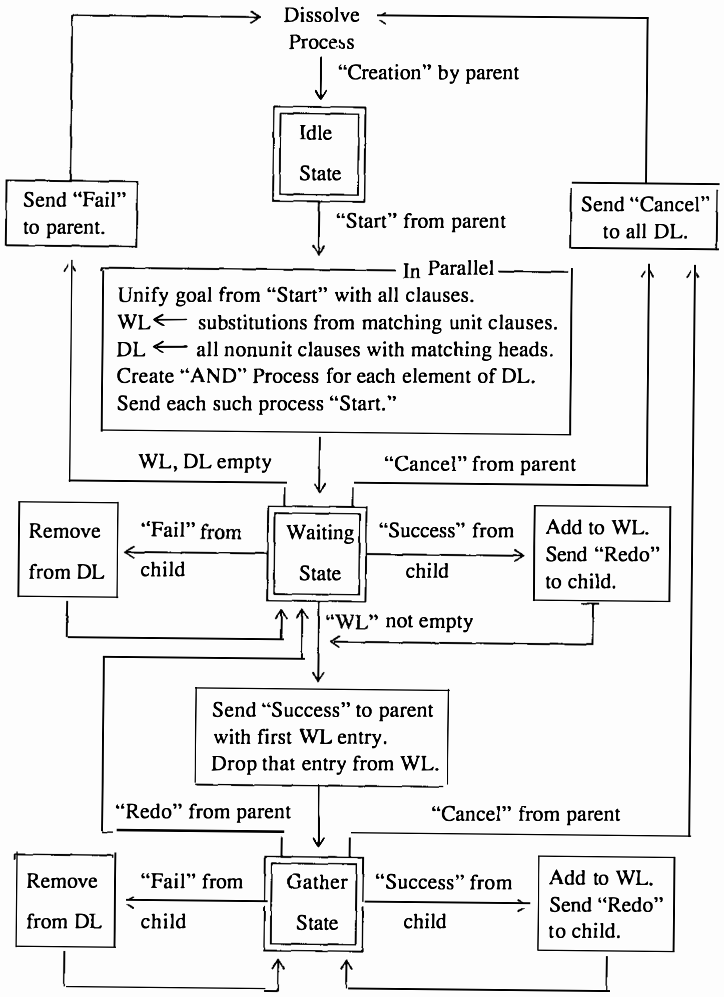
21.4.3 The OR Process

An OR process is created by a parent AND process for the express purpose of solving some single goal. The specific goal literal to be solved is passed to the OR process, along with any currently available bindings for variables in it, by the Start message from the parent. The OR process returns to its parent solutions to this literal one at a time in the form of sets of bindings to variables not bound in the original Start information. There is no presumed order to the sequence in which these answers may be presented. Thus two identical OR processes with the same initial goal are guaranteed to return the same (complete) set of answers, but it may return them in two different, unpredictable, orders. This is unlike PROLOG, where solving the same goal with the same program two different times gives the same solutions in the same order, but where it is possible that not all solutions will be found.

To help govern its computation, an OR process maintains for the life of its activity two data structures, a ***Waiting List*** (WL) of solutions to this goal which have been computed but not yet sent back to the parent, and a ***Descendant List (DL)*** of AND processes under its control that are still busy.

Conery's full model also includes several other data structures which help optimize the process. These are not necessary to an understanding of the basic process, and are discussed in the section entitled "Result Caching and Other Extensions."

Figure 21-17 diagrams in somewhat more detail what happens as an OR process transitions between the various states. Once created by its parent AND process, an OR process remains in the Idle state until re-


Dissolve Process
"Creation" by parent
Idle State
Send "Fail" to parent.
"Start" from parent
Send "Cancel" to all DL.
In Parallel
Unify goal from "Start" with all clauses.
WL ← substitutions from matching unit clauses.
DL ← all nonunit clauses with matching heads.
Create "AND" Process for each element of DL.
Send each such process "Start."
WL, DL empty
"Cancel" from parent
Remove from DL
"Fail" from child
Waiting State
"Success" from child
Add to WL. Send "Redo" to child.
"WL" not empty
Send "Success" to parent with first WL entry. Drop that entry from WL.
"Redo" from parent
"Cancel" from parent
Remove from DL
"Fail" from child
Gather State
"Success" from child
Add to WL. Send "Redo" to child.

DL = Descendant List of still busy AND processes
WL = Waiting List of bindings to go back to parent.

FIGURE 21-17
The generic OR process.

ceipt of a Start message. This tells the OR process what goal it is to solve and causes it to transition to the Waiting state. The OR process will remain in this state until it finds the initial solution (if any) to that goal.

During the first transition to Waiting, the OR process will perform several activities:

- Attempt to unify the given goal literal with all clauses in the logic program
- Make the WL point to a list of all variable substitutions resulting from successful unifications with unit clauses
- For each successful unification with a nonunit clause (i.e., one with one or more literals in the antecedent)
  - Create a separate child AND process
  - Send it a Start message with the name of the clause and the bindings from the head unification
  - Add the "name" of the AND process to the DL list

The basic model assumes complete flexibility in the order in which these activities are carried out. The set of head unifications could be done in parallel, along with creation of the AND processes for the nonunit matches. Further, each head unification could itself be done in parallel, in manners similar to that discussed earlier. Finally, the order in which items are added to the WL and DL lists is not specified. At one extreme, a PROLOG-like fixed ordering could be enforced. At the other, a fully parallel head unification could add items to the lists in order of their completion, with no guarantees of sequencing.

Once it has entered a Waiting state, an OR process remains there until one of several events occurs. First, and most common, if the process finds the DL list nonempty (i.e., it has one or more solutions in it), it removes one (again, which one is up to the system's designer), packages the bindings it contains into a ***Success message,*** forwards the message back up to its parent, and transitions to the Gather state.

While in the Waiting state, receipt of a message from a child AND process will cause some processing. If the message is a Success, then the bindings are removed from the message and sent to the parent, and the child is sent a ***Redo*** message. This will cause the child to start looking for yet another message. Having satisfied the parent momentarily, the OR process also moves to the Gather state.

If the message from the child is a ***Fail message,*** then that process has no (more) solutions that it can derive from its designated clause, and has quit. The OR process will then remove the child's name from the DL list of active processes. If this makes DL become empty, and if WL is also empty, then all possible solutions to the original goal have been generated and passed back to the parent process. The OR process will then send a ***Fail message*** back to the parent process and dissolve itself.

The final action that might affect an OR process in the Waiting state is receipt of a ***Cancel message*** from its parent. In this case the process should send Cancel messages to each of its children processes that are still active (as indicated by an entry on the DL list), and then dissolve itself. The Cancels sent downwards will have the eventual effect of canceling all descendants of the OR.

As described above, transmission of a solution to a parent causes an OR process to enter the Gather state. In this state the process responds to Success messages from children by placing the solutions on the WL list and sending back a Redo message. Likewise, a Fail message from a child causes the OR process to remove that child's name from the DL list. The net effect is that computation of alternative solutions continues as in the Waiting state, but that no Success or Fail messages are sent upwards. The process assumes that the parent is still busy handling the last Success.

In the Gather state an OR process is sensitive to two types of messages from its parent. A ***Cancel message*** is handled exactly as before, with Cancels going out to all children, and a dissolution of the process. A ***Redo message*** indicates to the child that the parent is ready for another solution. The OR process then transitions back to the Waiting state, where, if the WL is nonempty, the next solution is immediately sent upward, and the process transitions back to the Gather state.

The net effect of this model is that an OR process attempts to keep the computations of as many solutions as possible in action at the same time and stack up known solutions for immediate relay to the parent when asked for them.

21.4.4 The AND Process

An AND process is created by a parent OR process for the express purpose of finding solutions that satisfy some particular clause whose head has successfully unified with some goal. The clause and the unification information are passed to the AND process by a Start message from the parent. The AND process returns to its parent one (or more) solutions to this literal in the form of sets of bindings to variables not bound in the original Start information. There is no presumed order to the sequence in which these answers may be presented.

In operation, an AND process will create and manage several child OR processes, up to one for each literal in the clause's antecedent. Unlike the OR process, however, an AND process must be very careful about these children processes, for it is possible to put an AND process in a computational overload that totally overwhelms the gains due to parallelism. Figure 21-10 was an example of this.

Instead of the naive approach, assume that our AND process is more careful, and initially creates and starts an OR process only for the last literal of Figure 21-10 first. For each solution that this process generates, we create and start an OR process for the third literal, with the bindings given to each such process corresponding to a solution generated by the first. In turn, for each solution generated by this process we create and start OR processes for each of the initial two literals, again with initial bindings, reflecting the appropriate trail of solutions. Any time both of these processes return true for the same set of bindings, we get a solution to the entire literal. This is very similar to the ***dataflow AND parallel*** model discussed earlier.

The reduction in computation for the AND process is substantial. The first process will generate at most two solutions, each generating a process for the third literal. Each of these in turn will generate two solutions, which in turn will generate two pairs of OR processes for the other literals.

The net result is that we have exchanged an immense amount of AND process computation for a more controlled growth in parallelism with more total OR processes but only a trivial amount of internal computation.

Figure 21-18 diagrams a basic AND process. The three states are as before. The Idle state is the one entered upon creation of the process, and exited once the parent sends its first Start message. Each literal in the clause is then marked with one of three labels: ***solved, pending,*** or ***blocked.*** A ***solved literal*** is one for which a child OR process has reported a solution. A ***pending literal*** is one for which a child OR process is busy. A ***blocked literal*** is one whose child OR process cannot start yet because of data dependencies. The algorithm that does this starts by labeling the head literal of the clause (HG) as solved, and all others as blocked. It then performs a dataflow analysis as discussed in Section 21.3. The first time through this is based strictly on the bindings provided by the parent OR process. Given this dataflow, all literals that were blocked, but all of whose predecessors are marked as solved, have their markings changed to pending.

Now, for each pending literal a new OR process is created and initialized by a start message. The AND process then enters its Waiting state. A Success message from a child causes the literal associated with that process to be labeled solved (from pending) and the bindings to be recorded internally. If all literals are marked as solved, the AND process sends a Success message to its parent and enters its Gather state. (For simplicity here we assume that the AND process does not try to compute ahead as does the OR process.)

Upon receipt of a Redo message, if there are still unsolved literals, the dataflow analysis stage is reentered and the above process is repeated. This permits a dynamic modification of the dataflow depending on what really happened in terms of bindings and helps to maximize the amount of parallelism supported.

As long as success messages continue to come back from the child OR processes, this ***forward execution mode*** continues. As soon as one of the pending children answers back with a fail message, however, something very akin to the ***semiintelligent backtrack*** process discussed in Chapter 18 is initiated. Conery calls this ***backward execution.***

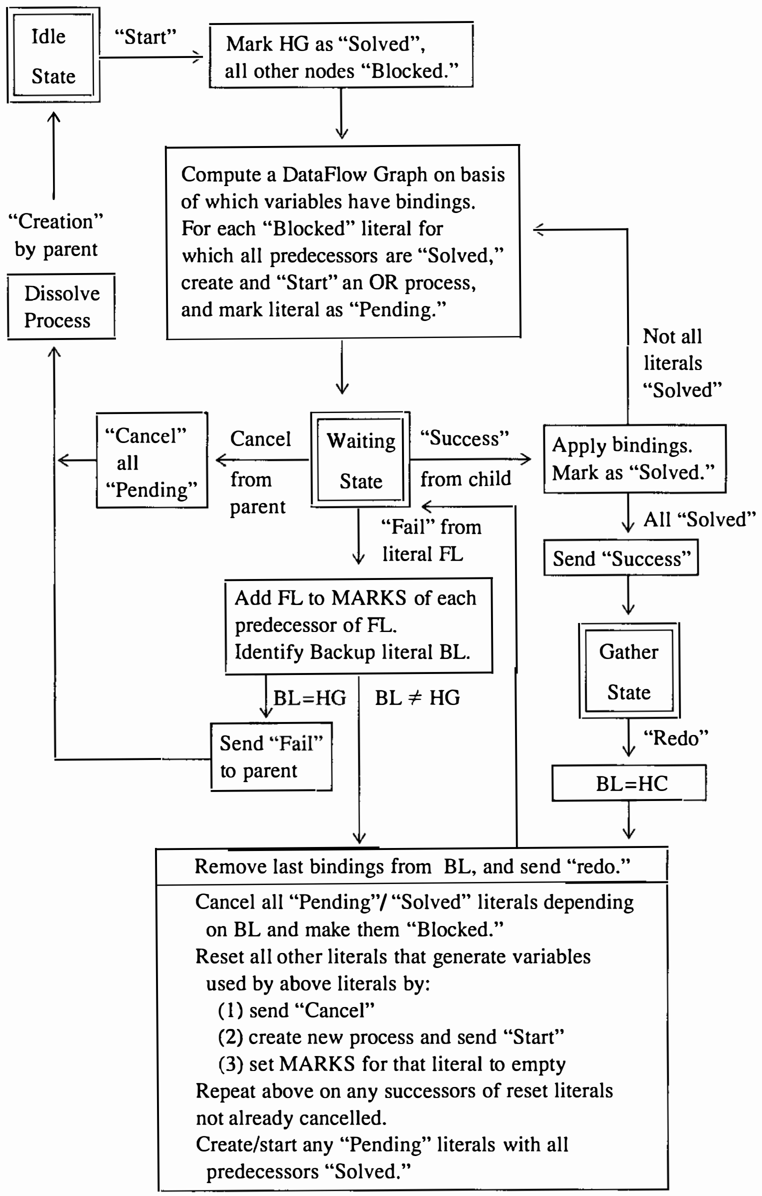

Idle State
"Start"
Mark HG as "Solved", all other nodes "Blocked."
Compute a DataFlow Graph on basis of which variables have bindings. For each "Blocked" literal for which all predecessors are "Solved," create and "Start" an OR process, and mark literal as "Pending."
"Creation" by parent
Dissolve Process
Not all literals "Solved"
"Cancel" all "Pending"
Cancel from parent
Waiting State
"Success" from child
Apply bindings. Mark as "Solved."
"Fail" from literal FL
All "Solved"
Send "Success"
Add FL to MARKS of each predecessor of FL. Identify Backup literal BL.
Gather State
BL=HG
BL ≠ HG
Send "Fail" to parent
"Redo"
BL=HC
Remove last bindings from BL, and send "redo."
Cancel all "Pending"/"Solved" literals depending on BL and make them "Blocked."
Reset all other literals that generate variables used by above literals by:
(1) send "Cancel"
(2) create new process and send "Start"
(3) set MARKS for that literal to empty
Repeat above on any successors of reset literals not already cancelled.
Create/start any "Pending" literals with all predecessors "Solved."

FIGURE 21-18
Simplified AND process.

In this mode, each literal in the clause that was a predecessor of the literal sending the fail message (named FL in Figure 21-18) is marked as a potential backtrack literal. This is done by keeping for each literal a list called **MARKS,** where its successors that fail are recorded. Then a backup literal (BL in the figure) is identified as being the ''latest'' of any literal marked with either the failing literal FL or one of its successors. As in Chapter 18, the latter is to guarantee that we do not backtrack too far after a chain of failures.

Now all bindings resulting from BL are deleted from the current solution, and all successors of BL in the dataflow which were marked as pending or solved are canceled. A new OR process is now started for each and the Waiting state is reentered.

If the backup literal BL is ever the head (HG), then this process has failed, and the parent is so informed.

RESULT CACHING AND OTHER EXTENSIONS. Figure 21-15 is a simplistic version of Conery's work. One of the extensions he discusses is the inclusion of a ***result cache*** to save intermediate results and prevent wholesale recomputation of answers on a backtrack. Each literal has associated with it a **New list,** where all solutions that have been delivered so far are recorded. Then, instead of canceling and restarting an OR process on a literal at backup time, these prior solutions can be used first, with the process left in the Gather start. Only when those solutions are exhausted would a Redo message to the child return the next answer.

Obviously this complicates memory management and can get caught in infinite-solution problems, but it can save significant computation.

A similar trick can be added to the Gather state processing. After finding one solution and delivering it to the parent, the AND process can go forward and induce a failure in the final clause of the dataflow, and try to compute an alternative answer. The equivalent of a WL could accumulate the results.

21.5 COMMITTED-CHOICE LANGUAGES

(Shapiro, 1989)

In functional programming a ***stream function*** is a function that returns an object that is not fully evaluated. Part of it is left as a ***closure,*** which is a promise to compute the rest at some point in the future. When the unexpanded component is finally needed, the requesting process suspends until the closure is evaluated. In highly parallel systems such closures can be computed eagerly, before they are needed.

The same idea can be grafted into logic languages. Assume that we have two goals which share a common variable. One will be a generator and one a checker. In all the AND parallel concepts up to now we have suspended the checker until the generator computes a binding for the shared variable. What if, however, we permit the generator to compute just part of the binding, leaving an unbound variable for the part that is not yet available. Assume also that as soon as this binding is made, the checker can start up, in parallel with the continuation of the generator. This is the essence of ***stream AND parallelism.***

The one difference between logic languages as we have discussed them to this point and the functional languages that originated the stream concept is that the latter are totally ***deterministic***—once they assign a value to a variable, they never retract it. Logic programs that never backtrack are also deterministic and thus would work nicely. Many PROLOG programs, however, are ***nondeterministic***—they can return several different bindings. Backtracking after sharing a binding with another literal would be difficult, to say the least. Consequently, logic languages that wish to support both stream AND parallelism and the possibility of multiple solutions must somehow have a way of specifying that after some point exactly one binding is to be used, and will never be retracted. The program must ***commit*** to one binding to permit all the goals sharing the variable to proceed in parallel. Such languages are called ***committed-choice, nondeterministic languages,*** and they are the subject of the following subsections.

21.5.1 Some Examples

Figure 21-19 gives some simple examples of stream AND parallelism in use. For simplicity we assume a PROLOG-like syntax, with the primary difference in semantics that all right-hand literals of a clause can be ex-

```
? – producer(data-stream),consumer(data-stream).

producer((datum.rest)): – produce(datum), producer(rest).
consumer((datum.rest)): – consume(datum), consumer(rest).
consumer(x): – consumer(x)
```

(*a*) Producer-consumer.

```
? – producer(data-stream),consumer(data-stream).

producer(((datum.response).rest)): – produce(datum),
                                      check(datum, response),
                                      producer(rest).

consumer(((datum.response).rest)): – consume(datum, response),
                                       consumer(rest).
```

(*b*) Two-way communication.

```
? – producer1(ds1), producer2(ds2), merge(ds1, ds2, ds3),
    consumer(ds3).

producer1, producer2, consumer as above.

merge((datum.rest1), ds2, (datum.ds3)): – !, merge(rest1, ds2, ds3).
merge(ds1, (datum.rest2), (datum.ds3)): – !, merge(ds1, rest2, ds3).
```

(*c*) A simple merge.

FIGURE 21-19
Examples of stream AND parallelism.

ecuted in AND parallel fashion as soon as the clause head unifies with a goal.

The first example is a generic form of a classic ***producer–consumer*** application. There are two processes, a producer and a consumer, that we would like to run in parallel. Assume that each is running on a separate processor but with some sort of shared memory between them. The producer should generate a stream of data that is accepted by the consumer and processed one item at a time. Generating one piece of data should occur in parallel with processing of previous data. In the example, executing the goal **producer** with a variable *data-stream* as its argument causes the computation of a new value by **produce** and a recursive call to **producer** for the rest. The variable representing the new value is dotted with a variable representing the rest of the stream, and bound to the initial variable.

The **consumer** goal shares *data-stream* with **producer** and should do nothing until something is bound to it. When something is bound to it, the car is passed to the function that handles it (**consume**) and the cdr is passed to a recursive call back to **consumer.** This latter **consumer** goal should again suspend until **producer** generates another binding. **Consume,** however, has been activated but will not proceed until the **produce** goal with the matching variable actually binds something to it. The net effect is that the producer can be busy propagating tasks to compute future elements of the shared stream while the consumer is working on spawning tasks to handle each element.

The odd second clause for **consumer** is there simply as a rather crude method of "suspending" **consumer** until a binding is made. As long as the argument is not a list, the second clause spins on itself. The next section will introduce a cleaner mechanism. Similar clauses for **consume** are assumed but not shown.

As with PROLOG, each time a clause executes, it gets a fresh copy of variables. Thus each of these recursions by **producer** really is independent of the others. Each of the *datum* variables is a distinct variable (one could assume an invisible "subscript" on each to designate the difference).

Figure 21-19(*b*) is an example of two-way communication. We assume here that the shared variable is again a list, but now each element is a pair. The car is a piece of data generated by the producer; the cdr is a response indication of some sort from the consumer. As before, the producer binds a list to the shared variable, but now this list has two unbound variables inside it. One of them is the rest of the stream. The other is a variable to be bound by the consumer as a response to the datum it received. Execution is as before, except that now the **check** goal also suspends until a response to a datum is generated.

Note that in this example there can be many processes active or suspended at the same time. The producer can recurse extensively, leaving a long line of active **produce** and **check** calls. On the other side, the consumer can spawn **consume** calls in close succession to the producer, with these calls remaining suspended until their *datum* 'variables are bound.

Finally, Figure 21-19(*c*) represents a merge of two streams. If only one producer has an output, only one clause of **merge** fires, and the datum is transferred to an output stream as before. If both producers have outputs simultaneously, then both **merge** clause heads are satisfied by the **merge** goal. Although it is not exactly right semantically, the intent of the **cut** in the clause bodies i\ to indicate that only one of them wins in such cases, with the choice of which one being totally random (***nondeterministic***). The body of the winning clause inherits references to the unprocessed stream from the losing clause. This establishes a new **merge** goal, which will then most probably unify with the clause which will process the other stream.

21.5.2 Guarded Horn Clauses

In the examples of Figure 21-19, a key assumption was made about the underlying language which was strikingly different from PROLOG. Only one binding to any particular variable is permitted. Once a goal "commits" to a clause, whatever single binding that clause returns to the goal's variables is it. There is no "backtracking" to find additional alternatives.

This assumption has the effect of limiting the ***completeness*** of such languages—they can miss solutions that other languages might find. However, it also permits introduction of AND parallelism without the headaches of consistency checking described earlier. With proper care, OR parallelism also becomes feasible without worrying about buffering solutions.

The one new construct needed for this is some way for a programmer to specify when a clause is satisfied to the point that the system should "commit" to that clause. In modern terminology this is designated by separating the right-hand side of a clause into two parts: the ***guard*** and the ***body.*** The latter is the set of literals that become new goals when a goal commits to the clause. The former is a set of goals which must be satisfied before the clause can be committed to.

Syntatically, the distinction between a guard and the rest of the body is via a built-in predicate much like a cut. The predicate, usually called the ***commit operation,*** indicates that everything between the head literal and the commit must be solved before this clause can be considered as the one to "commit" to deliver a solution (Figure 21-20). Once a system "commits" to this clause, other clauses that unified with the goal that started this clause are not allowed to proceed, and are deleted from any future consideration for the goal. No backtracking occurs.

As discussed in the examples, another characteristic of these languages is that a particular clause may have to ***suspend*** until an appropriate set of bindings is available for it to proceed. Signaling this suspension can

Assuming commit operator is " | ", Guarded clause looks like:
p(...):−g1(...),...,gm(...), | ,b1(...),...,bn(...).

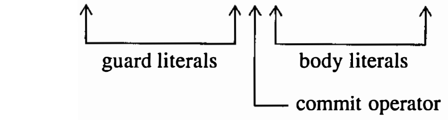
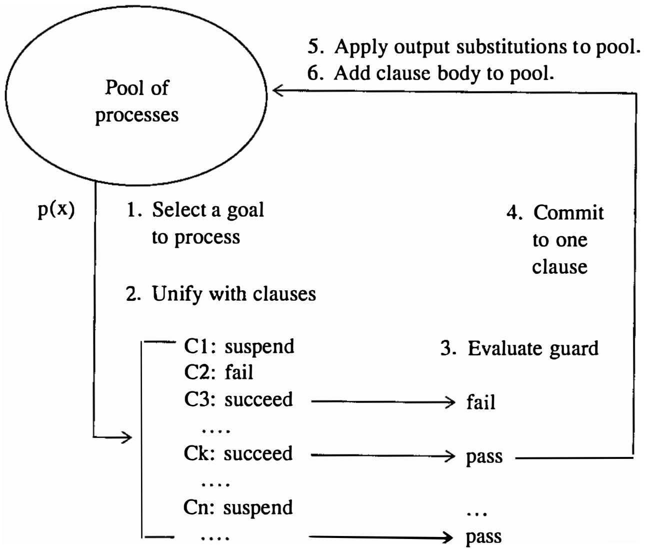


FIGURE 21-20
A guarded Horn clause.

be done in a variety of ways. Mode declarations for a whole set of related clauses can be used in advance. Annotations on the variables in a particular clause head can indicate which variables must be bound before succeeding. Finally, the language can simply make assumptions on the basis of how identifier symbols are used in a guard as to whether or not they need bindings before the clause can proceed.

In any case, Figure 21-21 diagrams a generic inference engine for such languages. Unlike PROLOG, where there is a specific order for attempting to solve goals, here there is simply a "pool" of them, with no intrinsic order. Each goal in this pool is often called a ***process,*** because it will trigger a potentially large-grain chunk of parallel processing. One or more of these goals can be extracted from this pool at any time.

Solving one particular goal requires several steps. First, the goal is unified with the heads of all clauses that have the same head predicate.



Note: Multiple goals may be processed in parallel.

FIGURE 21-21
A generic stream AND inference engine.

This could be done in parallel. There are three results from this unification. It could ***succeed***—unification works. It could ***fail***—unification is not possible. Finally, it could ***suspend***—there is a goal variable that must receive a binding from some other process before either success or failure can be declared.

As goal-head unifications succeed, the guard literals of the clause are attempted. Again, any of the three results above are possible. Eventually, however, one or more of the clauses will have both its unification and guard satisfied. At that point, one of these clauses will be chosen (nondeterministically), and all other clause processing for that goal will be terminated. This includes those that are suspended and those that are busily processing their guards.

The winning clause may then take the substitutions resulting from the unification and guard, and make them available across the whole pool. Further, if there is any body to the winning clause, all the substitutions from the left of the commit are applied to the body's variables, and the literals in the body are made into new goals in the pool.

Processing stops when the pool is empty. This occurs when the last goal has committed to a clause with no body.

21.5.3 Variations

The generic inference engine of Figure 21-21 still leaves many choices open to the designer of a logic language based on it. These include selecting answers to questions such as:

- Exactly when are output substitutions from the clause permitted to be applied to variables in the goal and the pool—at unification, after unification, during guard, after guard?
- How does each clause know which variables from a goal need bindings before the clause can leave suspension?
- When can bindings to local clause variables take place?
- How complex can the guard literals be?
- Exactly what can be done in parallel, and is there any form of backtracking?

Many of these problems are interrelated and directly affect implementation complexity and performance. For example, permitting output substitutions back into the goal before the clause is committed but during guard execution opens up the possibility of backtracking if the goal itself can be an arbitrarily complex set of literals. If guards can be based on arbitrary programmer-defined relations, then it may be necessary to support a deep parallel AND/OR tree in each clause's guard, with complex signaling and environment management needed to terminate one of these trees when some other clause for the same goal wins commitment (something akin to that for the AND/OR process model). If the only guards permitted are system-defined built-ins which cannot spawn subgoals, then there is still a question of whether or not they are allowed to be generators for either goal or local variables, or simply checkers. The former requires at least some allocation and management of binding environments, although not as much as for the ***deep guards*** of the general case.

Languages which permit only built-ins as guards are often called ***flat guarded languages*** (there is no "tree" needed to support them). Languages in which guards can only be checkers are said to have ***safe guards.***

21.5.4 Languages

Languages with any of these characteristics are less than a decade old in their development and are still undergoing tremendous development. Figure 21-22 lists the major characteristics of some of these.

Figure 21-23 gives some very brief programs in three of these languages. The notation used is from the languages and not that which we have used for PROLOG. In no sense do these examples, or the summaries in Figure 21-22, do any of the languages justice. The original references should be seen for more completeness.

Language	Features
IC-Prolog (Clark, McCabe, and Gregory, 1982)	Dataflow coroutining, pseudo-parallel evaluation, guards, read-only variables, streams, runtime mode checking, backtracking, logically correct but incomplete.
Parlog (Gregory, 1987; Clark and Gregory, 1986)	Separate AND and OR parallel evaluation, one, all, and commited choice solutions, guards, streams, fixed argument mode procedures, compile-time mode checks, only one global binding environment.
Concurrent Prolog (Shapiro, 1985, 1986, 1987)	Committed choice nondeterminism, guards, streams, compile-time mode checks, read-only variables, annotation on calls, separate environment for each guard evaluation.
Flat Concurrent Prolog (Mierowsky, 1985; Shapiro 1987b)	Concurrent Prolog without user-defined guard predicates. Avoids multiple environments.
Flat Guarded Horn Clause/KL-1 (Fuchi et al., 1987)	Stream AND parallelism, primitive guards, compile-time checks, suspension on attempt to substitute to goal variable.
Strand (Foster and Taylor, 1989)	Stream AND parallelism, primitive guards, assignment builtin for output substitution, compile-time mode checks, matching for unification, single environment.

FIGURE 21-22
Some current stream AND parallel languages.

```
mode ots(?,^,?).
ots(key, value, tree(left, node(key, value), right)).
ots(key, value, tree(left, node(nkey, nvalue), right))
   ← key<nkey: ots(key, value, left).

ots(key, value, tree(left, node(nkey, nvalue), right))
   ← key>nkey: ots(key, value, right).
```

(*a*) Parlog ordered tree search.

```
quicksort([x|xs],ys): − partition(xs?, x, smaller, larger),
                        quicksort(smaller?, ss),
                        quicksort(larger?, ls),
                        append(ss?, [x|ls?], ys).

partition([y|in], x, [y|smaller], larger)
    : − x≥y|partition(in?, x, smaller, larger).

partition([y|in], x, smaller, [y|larger])
    : − x≥y|partition(in?, x, smaller, larger).

partition([ ], x, [ ], [ ]).
```

(*b*) Flat concurrent prolog quicksort.

```
? − buffer: = [slot1, slot2, slot3, slot3|stream],
    producer(1000, buffer), consumer(buffer, stream).

producer(n,[datum|rest]): − n>0| n1 is n − 1, datum: = "message,"
producer(n1, rest).
producer(0,[datum|rest]): −  datum: = "done."
consumer(["message"|rest), buffer): −  buffer: = [x|bs],
consumer(rest, bs).
consumer(["done"|rest], ).
```

(*c*) Strand bounded buffer program.

FIGURE 21-23
Some examples of stream AND languages.

The ***ordered-tree-search*** Parlog program demonstrates the programmer-specified modes for all arguments for a relation: input, output, and input in this case. The relation searches through a complex data structure where each node consists of a left and right subnode, and a key and value for the node itself. When a key match is found, the output substitution will bind the matching value to the middle argument. Two of the clauses have guards which commit the solution process down either the left or right subtree (a ":" is used to separate guard from body).

The ***quicksort*** program for Concurrent PROLOG demonstrates another way of specifying modes for variables. The arguments ending with a "?" are ***read-only variables***; the clauses suspend until those arguments are bound to values by other processes. Again, simple guards are present in some clauses (in this case a "|" is used to separate guards and bodies).

Finally, the STRAND (***STream AND language***) example is an elaboration of the producer–consumer problem. In this case the producer will generate 1000 messages but with at most four of them outstanding at a time. If the consumer is slow, the producer will recurse on itself only as long as the rest of the buffer is a list. As soon as it becomes an unbound variable, the producer suspends until the consumer chews through them. Note also the ***assignment relation,*** which uses the infix ":=" between a variable and a structure; in STRAND, this statement is the only way that substitutions back into the caller's arguments can be made.

21.6 IMPLEMENTATIONS

This section covers very briefly several research projects under way today that demonstrate various forms of the parallel models and languages described above. Again, because of the extensive work being done in the area today, this discussion is in no way comprehensive. The projects covered were chosen primarily because they had real measurements available or had unique features. Other projects of interest include:

- ***PPP,*** an extension to the PLM machine (Fagin and Despain, 1987)
- An abstract machine to support ***Carmel-2,*** a RISC to support Flat Concurrent PROLOG (Harsat and Ginosar, 1988)
- An AND parallel system running on a local area net (Carlton and Van Roy, 1987)
- A method for using the vector facilities of many conventional supercomputers to contain sets of solution bindings for a near-PROLOG language (Kanada and Sugaya, 1989)
- An implementation of Parlog on the ALICE machine described in Chapter 12 (Lam and Gregory, 1987)
- A method for using associative memories to hold the binding arrays for AND-OR parallel systems (Ribeiro, 1988)

21.6.1 Parallel Inference Engines at ICOT

Chapter 19 addressed some of the logic-language machines being developed as part of the Japanese Fifth-Generation Project. This work is also being extended to include a range of parallel implementations. The language being supported by these machines is denoted ***KL-1,*** for ***Kernal Language 1*** and is a form of flat, guarded, Horn clause (FGHC) logic.

Two different machine architectures are being developed. The first, ***Multi-PSI*** (Taki, 1986; Ichiyoshi at al., 1987), is a collection of 16 to 64 PSI-IIs (See Chapter 19) arranged in a two-dimensional mesh. Communication is via message passing. The second, the ***PIM*** [Parallel Inference Machine, (Goto and Uchida (1986)] is built around a tight cluster of eight processing elements that share memory. Multiple clusters may be tied together to form larger machines.

The architecture of both machines is a modification of the WAM to support KL-1 efficiently (Kimura and Chikayama, 1987). The major differences are:

- Goal arguments are always ***temporary,*** meaning that they can be kept in registers.
- Unification is almost always one-way—from goal arguments into clause variables. This corresponds to the WAM's ***write mode.***
- Most data structures are allocated off the heap, complicating memory reclamation. The equivalent of choice points can be managed explicitly, but variable cells are more difficult, necessitating a different style of garbage collection based on reference bits (Chikayama and Kimura, 1987).

As with the WAM, variable cells are tagged. However, to support the suspension concept of FGHC languages, a new tag called ***hook*** is supported to indicate that the cell represents an unbound variable for which one or more goals is suspended. The value field in this case points to the first of a list of ***goal records*** describing the suspended goals. When a binding is made to the variable, these records represent goals which may now be ready for processing.

Such goal record includes copies of the arguments and a pointer to the associated code for that predicate. This resembles the upper part of a WAM choice point. The record also includes a pointer field to chain together multiple goals for the ready queue.

When queued in a suspension list, each record also has associated with it a count of the number of variables which need binding before the record can be made ready. Each time a binding is made to a hooked variable, this counter is decremented for each goal record in its suspended list. Only when the count reaches zero is the record released for processing. Because of the multiple-processor environment, decrementing and testing of these count fields must be done atomically.

The unification instructions reflect this suspension mechanism. For most gets and unifys there are variants that run in write mode, and when they find that the goal argument dereferences to an unbound variable, they stack the variable's address on a suspension stack and branch to an instruction-specified location. At this label is usually found a ***suspend instruction,*** which sets the tags of the variables on the suspension stack to hook, sets the waiting count, and suspends the process. When the process is reawakened, this instruction has an address (usually the start of the clause code) at which execution should begin.

New goals are recorded with code that resembles a sequence of puts and a call in the WAM. First is a ***create-goal*** instruction, which allocates storage for a goal record from the heap. A sequence of ***set-xxx*** instructions are the equivalent of puts except that they put their new argument values in the new goal record. A final ***enqueue-goal*** takes this new goal record and queues it for processing.

21.6.2 The PEPSys Model
(Westphal and Robert, 1987)

The PEPSys (Parallel ECRC PROLOG System) is an attempt to combine OR and AND parallelism with sequential backtracking and shallow binding to minimize structure copying. The programmer specifies opportunities for parallelism in the program via *pragmas*. If it is not marked as potentially parallel, it is executed sequentially. Depending on processor resources at runtime, these opportunities may or may not be exploited.

A pragma with a predicate symbol specifies whether or not that predicate, when called, should be a candidate for OR parallelism. A hash table of addresses and values is kept with each processor as it starts an OR parallel child. A time stamp is used to help distinguish on which side of the OR boundary a particular variable was created.

AND parallelism is signaled in the body of a clause by using "#" instead of "," between two literals. The goals represented by these literals must be independent, with no variables that are read by one and written by another. The # is assumed to be right-associative, so that a string of literals separated by it will be considered for parallel execution. The management of bindings from two AND parallel nodes is asymmetric. All the solutions from the left literal are accumulated in a solution list until the right literal responds with its first solution. A join between the two then proceeds, with the solutions that succeed being passed on. In either case the solution list from the left literal is frozen, and all later solutions arriving for it are kept in a separate list.

A backtrack into the right-side literal from further on in the clause causes a new solution for it to be generated and compared against the frozen list from the left literal. This process continues until there are no more solutions possible from the right literal. Then all the solutions computed since the the first set was frozen are themselves made the frozen set, and the generation of solutions from the right-hand literal is repeated. Over time this will generate the complete join of the solutions from the two literals.

21.6.3 The Reduce-OR Model

(Ramkumar and Kale, 1989)

The reduce-OR model exploits both AND and OR parallelism in a system with similarities to Conery's AND/OR process model. It was designed to permit a variety of memory models from distributed to shared memory. This is possible because of the choice for developing and passing binding environments. The unification between a goal and a clause head occurs in two stages. In the forward phase unification is normal. If it succeeds, pointers are set up between the new environment for the clause and that for the goal. In the second phase, the new bindings are checked for any references to unbound variables in the goal. If any are found, the objects containing them are copied into the new clause's environment (this is called *importing*). At a successful conclusion to the clause, the clause's arguments are back-unified with the goal's arguments to *export* the bindings back again.

Special tags on data structures are used when it is known that the whole underlying object is ground—i.e., contains no unbound variables. Such objects need not be tested or copied further.

A compiler generates code for a WAM-like intermediate machine. The major differences in the unification instructions is the more extensive use of ***structure sharing*** as described above, versus the normal WAM's ***structure-copying*** behavior. Also, additional instructions called sendfactresp and sendclauseresp are used at the end of a clause's code to transfer the results back.

The process control instructions assume the existence of an underlying runtime system called ***Chare Kernal*** to manage tasks, messages, and memory. New instructions of the form getxxxenv set up binding environments. A firearc instruction is similar to a call except that it permits the specified subgoal to run in parallel. A suspend instruction manages waiting for the return messages.

Performance on real machines has been very successful. On shared-memory machines with up to 20 processors, almost linear speedups have been achieved. On hypercube machines, near linear performance up to about 8 processors has been observed, with a tailing down to speedups of perhaps 20 to 1 with 32 processors.

21.6.4 The RAP-WAM

(Hermenegildo, 1986; Chang and Chiang, 1989)

The ***RAP-WAM*** is a modification to the WAM to support restricted AND parallelism. Each processor executing the overall program looks like a standard WAM except that there is a new ***goal stack*** to handle goal frames and a new ***parcall frame*** to be stored on the processor's local choice-point stack. Each time a parallel call is to be made, a parcall frame is established. This frame has within it one slot for each goal to be run in parallel, along with such things as the number of goals left to schedule, the number of goals that have succeeded so far, and information to manage backtracking if necessary.

A new instruction, allocate-pcall-frame, builds a new parcall frame, and push-call generates a new parallel goal to include in that frame, and stacks the goal on the local goal stack. A pop-pending-goal then either pops a goal off the goal stack and executes it (with return information stored as in the WAM's call) or continues with the next instruction if there are no more goals. If a goal frame has been popped, completion returns execution to the pop-pending-goal instruction.

In analogy to the try-xxx instructions in the WAM, the RAP-WAM includes instructions like check-me-xxx, which are used to guide execution through a set of code variants for a single clause. Each variant tests for and then executes a different dataflow through the literals. One test instruction takes the form check-ground and verifies that a register dereferences to a ground term. The other is check-independent and takes two arguments, which must be independent for the instruction to pass. For either instruction, a failure of the check causes something akin to a ***shallow backtrack*** to branch to the next set of code designated by the check-me-xxx instruction.

When a processor runs out of work of its own to do, it looks around at other processors' goal stacks. When it finds a nonempty one, it executes it.

21.7 PROBLEMS

1. Assuming that a single processor can unify only two simple terms in one time unit, how many processors could be used, and how long would it take, for each of the following:
   a. **p**(1) and **p**(*x*)
   b. **p**(1,1) and **p**(*x*,*x*)
   c. **p**(1,1,1,1) and **p**(*x*,*x*,*x*,*x*)
   d. **p**(1,1,1,1,1,1,1,1) and **p**(*x*,*x*,*x*,*x*,*x*,*x*,*x*,*x*)
   e. **p**(**g**(**f**(*x*)),**h**(**f**(*x*))) and **p**(**g**(**f**(**a**)),**h**(**f**(**a**)))
2. Give some precise semantics for how favored bindings would work for the naive OR parallel model during both dereferencing and binding.
3. Trace out an OR parallel solution to Figure 16-3, query 4, assuming the program of Figure 16-2. Indicate for each variable if it is private, unconditionally bound, or conditional. If this changes as a function of time, indicate when the changes occur.
4. Pick as a second-stage set of generators the literals **q**(*u*,*w*) and **q**(*v*,*y*) and redraw Figure 21-13 accordingly. Convert the query of Figure 21-13 to a restricted AND-parallel expression. Use the notation of Figure 21-14.
5. Develop an AND/OR process model for Proplog, the propositional logic language of Figure 16-4 (Note: many things get simpler). Solve the picnic problem of Figure 15-16 using it.
6. Develop a stream-AND program for handling streams of binary-valued objects that pass through a relation built to represent a digital logic circuit. In particular:
   a. Define clauses for **AND, OR,** and **NOT.**
   b. Use the above in a program that simulates an Exclusive OR circuit.All arguments to these predicates are streams.

REFERENCES

Ackerman, William B., 1982. "Data Flow Languages," *IEEE Computer,* Vol. 15, No. 2, Feb., pp. 15–25.

Akimoto, Maruo, Shinichi Shimizu, Akio Shinagawa, Akira Hattori, and Hiromu Hayashi, 1985. "Evaluation of the Dedicated Hardware in Facom Alpha," *Proceedings Spring COMPCON,* pp. 366–369.

Allen, Donald C., S. A. Steinberg, and Lawrence A. Stable, 1987. "Recent Developments in Butterfly LISP," BBN Advanced Computers, Cambridge, MA.

Allen, John, 1978. *The Anatomy of LISP,* McGraw-Hill, New York.

Appel, Andrew W., 1987. "A Standard ML Compiler," *Lecture Notes in Computer Science #274,* Proceedings Functional Programming Languages and Architecture, Portland, OR, Springer-Verlag, New York, pp. 301–324.

Appel, Andrew W., 1989. "Runtime Tags Aren't Necessary," *Journal of LISP and Symbolic Computation,* Vol. 2, No. 2, June, pp. 153–162.

Arvind, and Kim P. Gostelow, 1982. "The U-Interpreter," *IEEE Computer,* Feb., pp. 42–49.

Arvind, K. P. Gostelow, and W. Plouffe, 1980. *An Asynchronous Programming Language and Computing Machine,* Department of Information and Computer Science, University of California—Irvine, June 1.

Arvind, and Robert A. Iannucci, 1981. *Instruction Set Definition for a Tagged-Token Data Flow Machine,* Computation Structures Group Memo 212, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Dec. 10.

Astrahan, M. M. et al., 1979. "System R: A Relational Database Management System," *IEEE Computer,* Vol. 14, No. 5, May, pp. 43–48.

Augustsson, Lennart, 1984. "A Compiler for Lazy ML," *Proceedings ACM Conference on LISP and Functional Programming,* pp. 218–227.

Backus, John, 1978. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM,* Vol. 21, No. 8, Aug., pp. 613–641.

Backus, John, 1981. "Function Level Programs as Mathematical Objects," *Proceedings ACM Conference on Functional Programming and LISP,* Portsmouth, New Hampshire, Oct. 18–22, pp. 1–10.

Backus, John, 1985. "From Function Level Semantics to Program Transformation and Optimization," *IBM Research Report RJ4567,* IBM, San Jose, CA, Jan. 89.

Baer, Jean-Loup, 1980. *Computer Systems Architecture,* Computer Science Press, Rockville, MD.

Bailey, Roger, 1985. "A HOPE Tutorial," *Byte,* August, pp. 235–258.

Baker, Henry G., 1978. "List Processing in RealTime on a Serial Computer," *Communications of the ACM,* Vol. 21, No. 4, April, pp. 280–294.

Barklund, Jonas, and Hakam Millroth, 1986. "Garbage Cut for Garbage Collection of Iterative PROLOG Programs," *Proceedings IEEE Symposium on Logic Programming,* Salt Lake City, Utah, Sept. 22–25, pp. 276–283.

Bartley, David H., and John C. Jensen, 1986. "The Implementation of PC Scheme," *ACM Conference on Functional Programming Language and Computer Architecture,* Aug., Boston, pp. 86–93.

Bavel, Zamir, 1982. *Math Companion for Computer Science,* Reston Publishing Co., Reston, VA.

Ben-Ari, Mordechai, 1984. "Algorithms for on-the-Fly Garbage Collection," *ACM Transactions on Programming Language and Systems,* Vol. 6, No. 3, July, pp. 333–344.

Benker, H., J. M. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffré, A. Pöhlmann, J. Noyé, B. Poterie, A. Sexton, J. C. Syre, O. Thibault, G. Watzlawilk, 1989. "KCM—A Knowledge Crunching Machine," *Proceedings 16th International Conference on Computer Architecture,* Jerusalem, Israel, June, pp. 186–194.

Berkling, Klaus, 1975. "Reduction Languages for Reduction Machines," *Proceedings 2nd International Symposium on Computer Architecture,* Houston, TX, Jan.

Berkling, Klaus, 1986. "Head Order Reduction: A Graph Reduction Scheme for the Operational Lambda Calculus," Syracuse University CASE Report 8613, Nov.

Blasgen, M. W., 1981. "System R: An Architectural Overview," *IBM Systems Journal,* Vol. 20, No. 4, pp. 41–62.

Bloss, Adrienne, Paul Hudak, and Jonathan Young, 1988. "Code Optimization for Lazy Evaluation," *Journal of LISP and Symbolic Computation,* Vol. 1, No. 2, Sept., pp. 147–164.

Bobrow, D., and B. Wegbreit, 1973. "A Model and Stack Implementation of Multiple Environments," *Communications of the ACM,* Vol. 16, No. 10, Oct., pp. 591–603.

Borriello, Gaetano, Andrew Cherenson, Peter Danzig, and Michael Nelson, 1987. "RISCs vs CICSs for PROLOG: a Case Study," *Proceedings Second International Conference on Architecture Support for Programming Language and Operating Systems,* Palo Alto, CA, Oct., pp. 136–145.

Bowen, K. A., K. A. Buettner, I. Cicekli, and A. K. Turk, 1986. "The Design and Implementation of a High-Speed Incremental Portable PROLOG Compiler," *Proceedings 3rd. Conference on Logic Programming,* London, July, Springer-Verlag, New York, pp. 650-656.

Brooks, Rodney, 1984. "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," *ACM Conference on LISP and Functional Programming,* Austin, TX, Aug., pp. 256–263.

Browning, S. A., 1980. *The Tree Machine: A Highly Concurrent Computing Environment,* Ph.D. Dissertation, Department of Computer Science, California Institute of Technology, Pasadena, CA.

Brownston, Lee, R. Farrell, E. Kant, and N. Martin, 1985. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming,* Addison-Wesley, Reading, MA.

Bruynooghe, M., and L. M. Periera, 1985. "Deduction Revision by Intelligent Backtracking," in *Implementations of PROLOG,* J. A. Campbell, Ed., John Wiley, New York, pp. 194–215.

Burge, W. H., 1975. *Recursive Programming Techniques,* Addison-Wesley, Reading, MA.

Burstall, R. M., D. B. MacQueen, and D. T. Sannella, 1980. *HOPE: An Experimental Applicative Language,* CS Report CSR-62-80, University of Edinburgh, Edinburgh, Scotland, May.

Burton, F. W., and M. R. Sleep, 1981. "Executing Functional Programs on a Virtual Tree of Processors," *Proceedings ACM Conference on Functional Programming Language,* Portsmouth, New Hampshire, Oct. 18–22, pp. 187–194.

Campbell, J. A. 1984. *Implementations of PROLOG,* John Wiley, New York.

Carlton, M., and Peter Van Roy, 1987. "A Distributed PROLOG System with AND-Parallelism," EECS Department, University of California—Berkeley.

Chang, Chin-Liang, and Richard Char-Tung Lee, 1973. *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, New York.

Chang, Jung-Herng, and Alvin Despain, 1985. "Semi-intelligent Backtracking of PROLOG Based on Static Data Dependency Analysis," *Proceedings Symposium on Logic Programming,* Boston, MA, July 15–18, pp. 10–21.

Chang, Jung-Herng, Alvin Despain, and Douglas DeGroot, 1985. "AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis," *Proceedings Spring COMPCON,* San Francisco, CA, Feb. 25–28, pp. 218–225.

Chang, Si-En, and Y. Paul Chiang, 1989. "Restricted AND-Parallelism Execution Model with Side Effects," *Proceedings North American Conference on Logic Programming,* Cleveland, OH, Oct., pp. 350–368.

Chikayama, T., and Y. Kimura, 1987. "Multiple Reference Management in Flat GHC," *Proceedings 4th International Conference on Logic Programming,* Melbourne, Australia, MIT Press, Cambridge, MA, May, pp. 276–293.

Church, A., 1951. "The Calculi of Lambda-Conversion," *Annals of Mathematical Studies 6,* Princeton University Press, Princeton, NJ.

Ciepielewski, Andrezej, Seif Haridi, and Bogumil Hausman, 1989. "A Sequential Abstract Machine for Flat Concurrent PROLOG," *Journal of Logic Programming,* Vol. 7, No. 2, Sept., pp. 125–147.

Clark, Douglas W., and C. C. Green, 1977. "An Empirical Study of List Structure in LISP," *Communications of the ACM,* Vol. 20, No. 2, pp. 78–87.

Clark, Keith, and Steve Gregory, 1986. "PARLOG: Parallel Programming in Logic," *ACM Transactions on Programming Language and Systems,* Vol. 8, No. 1, Jan., pp. 1–49.

Clark, Keith, F. G. McCabe, and Steve Gregory, 1982. "IC-PROLOG Language Features," in *Logic Programming,* K. L. Clark and S. A. Tarnlund, Eds., Academic Press, London, pp. 253–266.

Clark, K. L., and S. A. Tarnlund, Eds., 1982. *Logic Programming,* Academic Press, London.

Clarke, T. J., P. J. Gladstone, C. D. Maclean, and A. C. Norman, 1980. "SKIM—the S,K, I reduction machine," *Proceedings LISP-80 Conference,* Stanford, CA, Aug.

Clinger, William, 1988. "Semantics of Scheme," *Byte,* Feb., pp. 221–227.

Clocksin, W. F., and C. S. Mellish, 1984. *Programming in PROLOG,* 2nd ed., Springer-Verlag, Berlin.

Codd, E., 1970. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM,* Vol. 13, No. 6, June, pp. 377–387. See also *Communications of the ACM,* Vol. 26, No. 1, Jan. 1983, pp. 64–69.

Cohen, Jacques, 1981. "Garbage Collection of Linked Data Structures," *ACM Computing Surveys,* Vol. 13, No. 3, Sept., pp. 341–367.

Cohen, Jacques, 1985. "Describing PROLOG by Its Interpretation and Compilation," *Communications of the ACM,* Vol. 28, No. 12, pp. 1311–1324.

Cohen, Paul, Avron Barr, and Edward Fiegenbaum, 1981. *The Handbook of Artificial Intelligence,* Vols. 1, 2, 3, William Kaufman, Los Altos, CA.

Colmerauer, Alain, 1985. "PROLOG in 10 Figures," *Communications of the ACM,* Vol. 28, No. 12, pp. 1296–1310.

Colomb, R. M., 1985. "Use of Superimposed Code Words for Partial Match Data Retrieval," *Australian Computer Journal,* Vol. 17, No. 4, Nov., pp. 181–188.

Colomb, R. M., and Jayasooriah, 1986. "A Clause Indexing System for PROLOG Based on Superimposed Coding," *Australian Computer Journal,* Vol. 18, No. 1, Feb.

Conery, John S., 1987a. *Parallel Execution of Logic Programs,* Kluwer Academic Pub., Boston, MA.

Conery, John S., 1987b. "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors," *Proceedings Symposium on Logic Programming,* San Francisco, CA, Aug., pp. 457–467.

Cox, P. T., 1985. "Finding Back Track Points for Intelligent BackTracking," *Implementations of PROLOG,* J. A. Campbell, Ed., John Wiley, New York, pp. 216–233.

Crammond, Jim, 1985. "A Comparative Study of Unification Algorithms for OR-parallel Execution of Logic Languages," *Proceedings IEEE International Conference on Parallel Processing,* Aug., pp. 131–138.

Curry, H. B., and R. Feys, 1958. *Combinatory Logic,* Vol. 1, North Holland, Amsterdam.

Curry, H. B., J. R. Hindley, and J. P. Seldin, 1972. *Combinatory Logic,* Vol. 2, North Holland, Amsterdam.

Darlington, John, and Michael Reeve, 1981. "Alice—A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," *Proceedings ACM Conference on Functional Programming and LISP,* Portsmouth, New Hampshire, Oct., pp. 65–73.

Date, C. J., 1983. *An Introduction to DataBase Systems,* Addison-Wesley, Reading, MA.

Date, C. J., and Colin J. White, 1988. *A Guide to DB2,* 2d ed., Addison Wesley, Reading, MA.

Davies, D. Julian, 1984. "Memory Occupancy Patterns in Garbage Collection Systems," *Communications of the ACM,* Vol. 27, No. 8, Aug., pp. 819–825.

Davis, Al, 1989. *Mayfly Parallel Processing System,* Technical Report HPL-SAL-89-22, Hewlett-Packard, Palo Alto, CA, Mar. 16.

Davis, A. L., and S. V. Robison, 1985. "The FAIM-1: Symbolic Multiprocessing System," *Proceedings COMPCON,* San Francisco, Feb., pp. 370–371.

Davis, M., and H. Putnam, 1960. "A Computing Theory for Quantification Theory," *Journal of the ACM,* Vol. 7, No. 3, pp. 201–215.

Debray, S. K., 1986. "Register Allocation in a PROLOG Machine," *Proceedings IEEE Symposium on Logic Programming,* Salt Lake City, Utah, Sept. 22–25, pp. 267–275.

Debray, Saumya, and David S. Warren, 1986. "Automatic Mode Inferences for PROLOG Programs," *Proceedings IEEE Symposium on Logic Programming,* Salt Lake City, Utah, Sept. 22–25, pp. 78–88.

DeGroot, Douglas, 1984. "Restricted AND Parallelism," *Proceedings International Conference on Fifth Generation Computer Systems,* Tokyo, Japan, Nov., pp. 471–478.

DeGroot, Douglas, 1985. "Alternate Graph Expressions for Restricted AND Parallelism," *Proceedings Spring COMPCON,* San Francisco, CA, Feb. 25–28, pp. 206–210.

DeGroot, Douglas, 1987. "Restricted AND-Parallelism and Side Effects," *Proceedings Symposium on Logic Programming,* San Francisco, CA, Aug., pp. 80–91.

Dennis, J. B., 1980. "Data Flow Supercomputers," *IEEE Computer,* Nov., pp. 48–56.

Dennis, J. B., 1982. *VAL,* MIT Computation Structures Group Memo #213, Massachusetts Institute of Technology, Cambridge, MA, Mar.

Despain, A. M., and D. A. Patterson, 1978. "X-TREE: A Tree Structured Multiprocessor Computer Architecture," *Proceedings Fifth International Symposium on Computer Architecture,* Apr., pp. 144–151.

Dietrich, S. W., 1987. "Extension Tables: Memo Relations in Logic Programming," *Proceedings International Symposium on Logic Programming,* San Francisco, CA, Aug., pp. 264–272.

Dijkstra, E. W., L. Lumport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, 1976, 1978. "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Lecture Notes in Computer Science,* Vol. 46, Springer-Verlag, New York. See also *Communications of the ACM,* Vol. 21, No. 11, Nov., pp. 966–975.

Diller, Antoni, 1988. *Compiling Functional Languages,* John Wiley, Chichester, England.

Dobry, T. P., 1987a. *A High Performance Architecture for PROLOG,* Report UCB/CSD 87/352, University of California—Berkeley, May.

Dobry, T. P., 1987b. "A Coprocessor for AI: LISP, PROLOG and Data Bases," *Proceedings COMPCON,* San Francisco, Feb., pp. 396–407.

Dobry, T. P., A. M. Despain, and Y. N. Patt, 1985. "Performance Studies of a PROLOG Machine Architecture, *12th International Symposium on Computer Architecture,* Boston, MA, June, pp. 180–190.

Douglass, Robert J., 1985. "A Qualitative Assessment of Parallelism in Expert Systems," *IEEE Software,* May, pp. 70–81.

Dybvig, R. Kent, 1987. *The SCHEME Programming Language,* Prentice-Hall, Englewood Cliffs, NJ.

Dwork, Cynthia, Paris C. Kaneliakis, and John C. Mitchell, 1984. *On the Sequential Nature of Unification,* Report MIT/LCS/TM-257, MIT Laboratory for CS, Massachusetts Institute of Technology, Cambridge, MA, Mar.

Eisenbach, Susan, 1987. *Functional Programming: Languages, Tools, and Architectures,* Halsted Press, John Wiley, New York.

Fagin, Barry S., and Alvin M. Despain, 1987. "Performance Studies of a Parallel PROLOG Architecture," *Proceedings 14th International Symposium on Computer Architecture,* Pittsburgh, June, pp. 108–116.

Fitch, J. P., and A. C. Norman, approx. 1977. "A note on compacting garbage collection," *Software Practices and Exp.,* Vol. 7, No. 6, p. 713.

Forgy, Charles L., 1981. *The OPS5 User's Manual,* Carnegie-Mellon University Computer Science Department Report CMU-CS-81-135.

Forgy, Charles L., 1982. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence,* Vol. 19, Sept., pp. 17–37.

Forgy, C. L., 1984. *The OPS 83 Report,* Carnegie-Mellon University Computer Science Department Report CMU-CS-84-133, May.

Forgy, Charles, and J. McDermott, 1977. "OPS: A Domain-Independent Production System Language," *Proceedings 5th International Joint Conference on Artificial Intelligence,* pp. 933–939.

Forgy, Charles, Anoop Gupta, Allen Newell, and Robert Wedig, 1984. "Initial Assessment of Architectures for Production Systems," *Proceedings American Association on Artificial Intelligence,* Austin, TX, Aug., pp. 116–120.

Foster, Ian, 1989. "Strand: A Practical Parallel Programming Language," *Proceedings North American Conference on Logic Programming,* Cleveland, OH, Oct., pp. 497–512.

Foster, Ian T., and Stephen Taylor, 1989. *Strand: New Concepts in Parallel Programming,* Prentice-Hall, Englewood Cliffs, NJ.

Fuchi, K., and K. Furukawa, 1987. "The Role of Logic Programming in the Fifth Generation Computer Project," *New Generation Computing,* Vol. 5, No. 1, pp. 3–28.

Gabriel, Richard P., 1986. *Performance and Evaluation of LISP Systems,* MIT Press, Cambridge, MA.

Garey, Michael S., and David S. Johnson, 1979. "hpl.Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, San Francisco.

Gee, Jeff, Stephen W. Melvin, and Yale N. Patt, 1987. "Advantages of Implementing PROLOG by Microprogramming a Host General Purpose Computer," *Proceedings Fourth International Conference on Logic Programming,* Melbourne, Australia, May, MIT Press, Cambridge, MA, pp. 1–20.

Gellert, W., H. Kustner, H. Hellwich, and H. Kastner, 1975. *The VNR Concise Encyclopedia of Mathematics,* Van Nostrand Reinhold, New York.

George, Lal, 1989. "An Abstract Machine for Parallel Graph Reduction," *Proceedings Functional Programming Language and Computer Architecture,* London, Sept.

Georgeff, M., 1982. "A Scheme for Implementing Functional Values on a Stack Machine," *ACM Symposium on LISP and Functional Programming,* Pittsburgh, Aug. 15–18, pp. 188–195.

Gibson, Jack C., 1970. "The Gibson Mix," IBM Corp., Systems Development Division, TR00.2043, Poughkeepsie, NY, June 18.

Gilmore, P. C., 1960. "A Computing Procedure for Quantification Theory," *Journal of the ACM,* Vol. 7, pp. 201–215.

Goldberg, Adele, and David Robison, 1983. *Smalltalk-80: The Language and Its Implementation,* Addision-Wesley, Reading, MA.

Gordon, Michael, 1979. *The Denotational Description of Programming Languages*, Springer-Verlag, New York.

Goto, A., and S. Uchida, 1986. "Toward a High Performance Parallel Inference Machine—The Intermediate Stage Plan of PIM,". ICOT Technical Report TR201, ICOT, Tokyo, Japan.

Gregory, Steve, 1987. *Parallel Logic Programming in PARLOG*, Addision-Wesley, Workingham, England.

Greenblatt, Richard, D., Thomas F. Knight, Jr., John Holloway, David A. Moon, and Daniel L. Weinreb, 1984. "The LISP Machine," in *Interactive Programming Environments*, McGraw-Hill, New York, chap. 16.

Griss, Martin L., 1983. *The Portable Standard LISP User's Manual—Revised Version 3.1.7*, Department of Computer Science, TR-10, University of Utah, Salt Lake City, Utah, Feb.

Gupta, Anoop, 1987. *Parallelism in Production Systems*, Morgan Kaufmann Publishers, Los Altos, CA.

Gupta, Anoop, and Charles L. Forgy, 1983. *Measurements on Production Systems*, Carnegie-Mellon University Computer Science Department Report CMU-CS-83-167, Dec.

Gurd, John, and Ian Watson, 1980a. "Data Driven System for High Speed Parallel Computing—Part 1: Structuring Software for Parallel Execution," *Computer Design*, June, pp. 91–100.

Gurd, John, and Ian Watson, 1980b. "Data Driven System for High Speed Parallel Computing—Part 2: Hardware Design," *Computer Design*, June, pp. 91–100.

Guttag, John, James Horning, and John Williams, 1981. "FP with Data Abstraction and Strong Typing," *ACM Proceedings List and Functional Programming*, Portsmouth, New Hampshire, Oct., pp. 11–24.

Halstead, Robert H., 1985. "MultiLISP: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Language and Systems*, Vol. 7, No. 4, Oct., pp. 501–538.

Halstead, Robert H., 1986. "Parallel Symbolic Computing," *IEEE Computer*, Aug., Vol. 19, No. 8, pp. 35–43.

Harrison, P. G., 1982. "Efficient Storage Management for Functional Languages," journal unknown.

Harrison, Peter G., and Hessam Khoshnevisan, 1985. "Functional Programming Using FP," *Byte*, Aug., pp. 219–232.

Harsat, Arie, and Ran Ginosar, 1988. "CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent PROLOG," *Proceedings Fifth Generation Computing Systems*, Tokyo, Japan.

Hausman, Bogumil, Andrezej Ciepielewski, and Seif Haridi, 1987. "OR-Parallel PROLOG Made Efficient on Shared Memory Multiprocessors," *Proceedings Symposium on Logic Programming*, San Francisco, Aug., pp. 69–79.

Haynes, Christopher T., D. P. Friedman, and Mitchell Wand, 1986. "Obtaining Coroutines with Continuations," *Computing Languages*, Vol. 11, No. 3/4, pp. 143–153.

Henderson, Peter, 1980. *Functional Programming—Application and Implementation*, Prentice-Hall, Englewood Cliffs, NJ.

Henderson, Peter, and James Morris, 1976. "A Lazy Evaluator," *3rd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Language*, Atlanta, Jan., pp. 95–103.

Henderson, P., and D. A. Turner, 1982. *Functional Programming and Its Applications: An Advanced Course*, Cambridge University Press, Cambridge, England.

Henderson, Peter, Geraint A. Jones, and Simon B. Jones, 1983. *The LISPKit Manual*, Vols. I and II, Technical Monographs PRG-32(1) and PRG-32(2), Oxford University Computing Laboratory, Oxford, England.

Hennessy, John L., and David A. Patterson, 1989. *Computer Architecture: A Quantative Approach*, Morgan Kaufmann, Palo Alto, CA.

Hermenegildo, M. V., 1986. "An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs," *Proceedings 3rd International Conference on Logic Programming*, London, pp. 25–54.

Hermenegildo, Manuel, and Francesca Rossi, 1989. "On the Correctness and Efficiency of Independent AND-Parallelism in Logic Programs," *Proceedings North American Conference on Logic Programming*, Cleveland, OH, pp. 350–368.

Hickey, Tim, and Jacques Cohen, 1984. "Performance Analysis of on-the-Fly Garbage Collection," *Communications of the ACM*, Vol. 27, No. 11, Nov., pp. 1143–1154.

Horowitz, Ellis, 1983a. *Fundamentals of Programming Languages*, Computer Science Press, Rockville, MD.

Horowitz, Ellis, 1983b. *Programming Languages: A Grand Tour*, Computer Science Press, Rockville, MD.

Hudak, Paul, 1989. "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, Vol. 21, No. 3, Sept., pp. 359–411.

Hudak, Paul, and P. Waldler, Eds., 1988. "Report on the Functional Programming Language Haskell," Yale University Computer Science Department Report YALEU/DCS/RR656.

Hughes, John, 1982. "Graph Reduction with Super-Combinators," Report PRG-28, Oxford University Computing Lab, Oxford, England, June. *See also* " Supercombinators—A New Implementation Method for Applicative Languages," *ACM Symposium on LISP and Functional Programming*, Pittsburgh, pp. 1–100.

Huynh, Tien, Brent Hailpern, and Lee W. Hoevel, 1986. "An Execution Architecture for FP," *IBM Journal of Research and Development*, Vol. 30, No. 6, Nov., pp. 609–616.

Ichiyoshi, N., T. Miyazaki, and K. Taki, 1987. "A Distributed Implementation of Flat GHC on the Multi-PSI," *Proceedings 4th International Conference on Logic Programming*, Melbourne, Australia, MIT Press, Cambridge, MA, May, pp. 257–275.

Ishida, Toru, and Salvatore J. Stolfo, 1985. "Towards the Parallel Execution of Rules in Production System Programs," *Proceedings IEEE International Conference on Parallel Processing*, Aug., pp. 568–575.

Jamsek, D., K. J. Greene, S. K. Chin, and P. R. Humeun, 1988. "WINTER—WAMS in TIM Expression Reduction," *Proceedings North American Conference on Logic Programming*, Oct., Cleveland, OH, pp. 1013–1029.

Janssens, Gerda, Bart Demoen, and Andre Marien, 1988. "Improving the Register Allocation in WAM by Reordering Unification," *Proceedings Fifth International Conference on Logic Programming*, Seattle, WA, MIT Press, Cambridge, MA, pp. 1388–1402.

Johnsson, Thomas, 1983. "A G-machine: An Abstract Machine for Graph Reduction," Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, Aug.

Johnsson, Thomas, 1984. "Efficient Computation of Lazy Evaluation," *Proceedings ACM-SIGPLAN Conference on Compiler Construction*, June, Montreal, pp. 58–69.

Jones, Neil, and Steven Muchnick, 1982. "A Fixed-Program Machine for Combinator Expression Evaluation," *ACM Symposium on LISP and Functional Programming*, Pittsburgh, Aug. 15–18, pp. 11–20.

Lucas, P., and K. Walk, 1969. "On the Formal Description of PL/I," *Annual Review of Automated Programming*, Vol. 6, No. 3, pp. 105–182.

Kamiya, Shigeo, Susumu Matsada, Kazuhide Iwata, Shigeki Shibayama, Hiroshi Sakai, and Kunio Murakami, 1985. "A Hardware Pipeline Algorithm for Relational Database Operation and Its Implementation Using Dedicated Hardware," *Proceedings 12th Symposium on Computer Architecture*, Boston, pp. 250–257.

Kanada, Yasusi, and Masshiro Sugaya, 1989. "A Vectorization Technique for PROLOG without Explosion," *Proceedings 11th International Joint Conference on AI*, Detroit, MI, pp. 151–162.

Katevenis, M., 1985. *Reduced Instruction Set Computer Architecture for VLSI*, MIT Press, Cambridge, MA.

Keller, Robert M., 1983. *FEL (Function Equation Language) Programmer's Guide*, AMPS

Technical Memo #7, Department of Computer Science, University of Utah, Salt Lake City, Utah, Apr. 8.

Keller, R. M., and F. C. Lin, 1984. "Simulated Performance of a Reduction-Based Multiprocessor," *IEEE Computer,* July, pp. 70–82.

Keller, R. M., and M. R. Sleep, 1982, 1986. "Applicative Caching," Technical Report UUCS 82-014, Computer Science Department, University of Utah, Salt Lake City, Utah; also *ACM Transactions on Programming Language Systems,* Vol. 8, No. 1, 1986, pp. 88–105.

Keller, R. M., G. Lindstrom, and S. Patel, 1979. "A Loosely-Coupled Applicative Multiprocessing System," *Proceedings AFIPS NCC,* June, pp. 613–622.

Kennaway, J. R., and M. R. Sleep, 1983. "Novel Architectures for Declarative Languages," *Software and Microsystems,* Vol. 2, No. 3, June, pp. 59–70.

Kessler, R. R., S. T. Shebs, and W. F. Galway, 1986. "A Portable Common LISP Subset with High Performance," *ACM Conference on LISP and Functional Programming,* Aug., Boston.

Kieburtz, Richard B., 1985. "The G-Machine: A Fast, Graph-Reduction Evaluator," *Proceedings IFIP Conference on Functional Programming Language and Computer Architecture,* Nancy, France, pp. 400–413.

Kieburtz, Richard B., 1988. "Performance Measurement of a G-Machine Implementation," Oregon Graduate Center, University of Oregon, Beaverton.

Kieburtz, R. B., and J. Shultis, 1982. "Transformations of FP Program Schemas," *Proceedings ACM Conference on Functional Programming Language and Computer Architecture,* Pittsburgh, pp. 41-48.

Kimura, Yasunori, and Takashi Chikayama, 1987. "An Abstract KL1 Machine and Its Instruction Set," *Proceedings Symposium on Logic Programming,* San Francisco, Aug., pp. 468–477.

Kitsuregawa, Masaru, and Hidehiko Tanaka, 1988. *Database Machines and Knowledge Base Machines,* Kluwer Academic Publishers, Boston.

Kluge, Werner E., 1983. "Cooperating Reduction Machines," *IEEE Transactions on Computers,* Vol. C-32, No. 11, Nov., pp. 1002–1012.

Knight, Kevin, 1989. "Unification: A Multidisciplinary Survey," *ACM Computing Surveys,* Vol. 21, No. 1, Mar., pp. 93–124.

Kogge, Peter, 1981. *The Architecture of Pipelined Computers,* McGraw-Hill, New York.

Kogge, Peter, 1982. "An Architectural Trail to Threaded Code Systems," *IEEE Computer,* Mar., pp. 22–32.

Kogge, Peter, 1985. "Function-Based Computing and Parallelism: A Review," *Parallel Computing,* Fall, pp. 243–253.

Kogge, Peter, John Oldfield, Mark Brule, and Charles Stormon, 1989. "VLSI and Rules Based System," in *VLSI for Artificial Intelligence,* J. Delgado-Frias and W. R. Moore, Eds., Kluwer Academic Publishers, Norwell, MA.

Kowalski, Robert, 1979. *Logic for Problem Solving,* North Holland, New York.

Krueger, Steven, and Patrick Bosshart, 1987. "High LISP Performance on a High-Level Language Processor," *Proceedings COMPCON,* San Francisco, Feb., pp. 120–122.

Kumar, Vipin, and Yow-Jian Lin, 1988. "A Data-Dependency-Based Intelligent Backtracking Scheme," *Journal of Logic Programming,* Vol. 5, No. 2, June, pp. 165–180.

Kumon, Kouichi, Hideo Masuzawa, Akihiro Itashiki, Ken Satoh, and Yukio Sohma, 1986. "Kabu-wake: A New Parallel Inference Method and Its Evaluation," *Proceedings COMPCON,* San Francisco, pp. 168–172.

Kurosawa, K., S. Yamaguchi, S. Abe, and T. Bandoh, 1988. "Integrated Architecture for a High Performance Integrated PROLOG Processor IPP," *Proceedings Fifth International Conference on Logic Programming,* MIT Press, Cambridge, MA, pp. 1506–1530.

Lam, M., and S. Gregory, 1987. "PARLOG and Alice: A Marriage of Convenience," *Proceedings 4th International Conference on Logic Programming,* Melbourne, Australia, MIT Press, Cambridge, MA, May, pp. 294–311.

Landin, P. J., 1963. "The Mechanical Evaluation of Expressions," *Computer Journal,* Vol. 6, No. 4, pp. 308–320.

Landin, P. J., 1966. "The Next 700 Programming Languages," *Communications of the ACM,* Vol. 9, No. 3, Mar., pp. 157–164.

Lewis, P. M., D. J. Rosenkrantz, and R. E. Stearns, 1978. *Compiler Design Theory,* Addison-Wesley, Reading, MA.

Li, Deyi, 1984. *A PROLOG DataBase System,* Research Studies Press, Letchworth, England, and John Wiley, New York.

Lieberman, Henry, and Carl Hewitt, 1983. "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM,* Vol. 26, No. 6, June, pp. 419–429.

Mago, Gyula, 1979. "A Network of Microprocessors to Execute Reduction Languages—Parts 1 and 2," *International Journal of Computer and Information Science,* Vol. 8, No. 5, pp. 349–385, 435–471.

Mago, Gyula, 1980. "A Cellular Computer Architecture for Functional Programming," *Proceedings COMPCON,* pp. 179--187.

Mago, Gyula, 1985. "Making Parallel Computation Simple: The FFP Machine," *Proceedings COMPCON,* San Francisco, Feb., pp. 424-427.

Mago, Gyula, D. F. Stanat, and F. A. Koster, 1981. "Program Execution on a Cellular Computer: Some Matrix Algorithms," University of North Carolina at Chapel Hill.

Maier, David, 1983. *The Theory of Relational Data Bases,* Computer Science Press, Rockville, MD.

Marti, J., and J. Fitch, 1982. "The Bath Concurrent LISP Machine," School of Math, University of Bath, England.

Matthews, Gene, Robert Hewes, and Steve Krueger, 1987. "Single-Chip Processor Runs LISP Environments," *Computer Design,* May 1, pp. 69–76.

McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, 1965. *LISP 1.5 Programmer's Manual,* The MIT Press, Cambridge, MA.

McGraw, James R., and Stephen K. Skedzielewski, 1982. "Streams and Iteration in VAL—Additions to a Data-Flow Language," *3rd International Conference on Distributed Computer Systems,* Oct.

Mellish, C. S., 1981. "The Automatic Generation of Mode Declarations for PROLOG Programs," DAI Research Paper 163, University of Edinburgh, AI Department, Edinburgh, Scotland, Aug.

Mendelson, Elliot, 1964/1979. *Introduction to Mathematical Logic,* Van Nostrand, New York.

Mierowsky, C., S. Taylor, E. Shapiro, J. Levy, and S. Safra, 1985. "The Design and Implementation of Flat Concurrent PROLOG," Technical Report CS85-09, July, Weizmann Institute of Science, Department of Applied Math, Rehovot, Israel.

Mills, J. W., 1986. "A High Performance LOW RISC Machine for Logic Programming," *Proceedings IEEE Symposium on Logic Programming,* Salt Lake City, Utah, Sept. 22–25, addendum.

Mills, J. W., 1989. "A High Performance LOW RISC Machine for Logic Programming," *Journal of Logic Programming,* Vol. 6, Nos. 1 and 2, Jan./Mar., pp. 179-212.

Milner, Robin, 1984. "A Proposal for Standard ML," *Proceedings ACM Conference on LISP and Functional Programming,* Austin, TX, Aug., pp. 184–197.

Miranker, Daniel, 1984. "Performance Estimates for the Dado Machine: A Comparison of TREAT and Rete," *Proceedings International Conference on Fifth Generation Computer Systems,* ICOT, Tokyo, Japan, Nov., pp. 449-457.

Miranker, Daniel, 1990. *TREAT: A New and Efficient Match Algorithm for AI Production Systems,* Morgan Kaufmann, San Mateo, CA.

Mitchell, John C., and Rober Harper, 1988. "The Essence of ML," *Proceedings ACM Symposium on Principles of Programming Language,* San Diego, Jan., pp. 28-46.

Moon, David A., 1984. "Garbage Collection in a Large LISP System," *ACM Conference on LISP and Functional Programming,* Austin, TX, Aug., pp. 235–246.

Moon, David A., 1985. "Architecture of the Symbolics 3600," *12th Symposium on Computer Architecture,* Boston, June 17–19, pp. 76–83.

Morioka, M., S. Yamaguchi, and T. Bandoh, 1989. "Evaluation of Memory System for Integrated PROLOG Processor," *Proceedings 16th International Conference on Computer Architecture,* Jerusalem, June, pp. 203–210.

Mulder, Hans, and Evan Tick, 1987. "A Performance Comparison between the PLM and an MC58030 PROLOG Processor," *Proceedings Fourth International Conference on Logic Programming,* MIT Press, Cambridge, MA, pp. 59–73.

Naganuma, Jiro, Takeshi Ogura, Shin-Ichiro Yamada, and Takashi Kimura, 1988. "High-Speed CAM-Based Architecture for a PROLOG Machine (ASCA)," *IEEE Transactions on Computers,* Vol. 37, No. 11, Nov., pp. 1375–1383.

Nakajima, K., H. Nakashima, M. Tokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto, and M. Mitsui, 1986. "Evaluation of PSI Microinterpreter," *Proceedings COMPCON,* San Francisco, Feb., pp. 173–177.

Nakashima, H., and K. Nakajima, 1987. "Hardware Architecture of the Sequential Inference Machine: PSI-II," *Proceedings IEEE Symposium on Logic Programming,* San Francisco, CA, Aug., pp. 104–113.

Nakazaki, Ryosei, A. Konagaya, S.-i. Habata, H. Shimazu, M. Umemura, M. Yamamoto, M. Yakota, and T. Chikayama, 1985. "Design of a High Speed PROLOG Machine," *12th Symposium on Computer Architecture,* Boston, June 17–19, pp. 191–197.

Naish, Lee, 1985. *Negation and Control in PROLOG, Lecture Notes in Computer Science No. 238,* Springer-Verlag, Berlin.

Neches, Philip, 1984. "Hardware Support for Advanced Data Management Systems," *IEEE Computer,* Nov., pp. 29–40.

Neches, Philip, 1986. "The Anatomy of a Data Base Computer System—Revisited," *Proceedings COMPCON,* San Francisco, Feb., pp. 374–377.

Newton, Michael O., 1987. "A High Performance Implementation of PROLOG," TR:5234:86, Computer Science Department, California Institute of Technology, Pasadena, CA, Apr. 10.

Nilsson, Nils J., 1980. *Principles of Artificial Intelligence,* Tioga, Palo Alto, CA.

Niwa, Masashi, Masanobu Yuhara, Kooji Hayashi, and Akira Hattori, 1986. "Garbage Collection with Area Optimization for FACOM ALPHA," *Proceedings Spring COMPCON,* San Francisco, Feb., pp. 235–240.

Patterson, D. A., and C. H. Sequin, 1982. "A VLSI RISC," *IEEE Computer,* Vol. 15, No. 9, Sept., pp. 8–21.

Patterson, D. A., et al., 1979. "Design Considerations for the VLSI Processor of X-TREE," *Proceedings Sixth International Symposium on Computer Architecture,* Apr., pp. 90–100.

Peyton Jones, Simon L., 1987. *The Implementation of Functional Programming Languages,* Prentice-Hall, London, Great Britain.

Pitman, Kent M., 1983. *The Revised MACLISP Manual,* MIT/LCS TR-295, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Pountain, R., 1985. "Parallel Processing: A Look at the Alice Hardware and HOPE Language," *Byte,* May, pp. 385–395.

Ramamohanarao, Kotagiri, and John Shepherd, 1986. "A Superimposed Codeword Indexing Scheme for Very Large PROLOG Databases," *Proceedings Third International Conference on Logic Programming,* London, pp. 569–576.

Ramesh, R., R. M. Verma, T. Krishnaprasad, and I. V. Ramakrishnan, 1989. "Term Matching on Parallel Computers," *Journal of Logic Programming,* May, Vol. 6, No. 3, pp. 213–228.

Ramkumar, Balkrishna, and Laxmukant Kale, 1989. "Compiled Execution of the Reduce-OR Process Model on Multiprocessors," *Proceedings North American Conference on Logic Programming,* Cleveland, OH, pp. 313–332.

Ramsdell, John D., 1986. "The CURRY Chip," *ACM Conference on LISP and Functional Programming,* Massachusetts Institute of Technology, Cambridge, MA, Aug., pp. 122–131.

Ribeiro, Joao, 1988. "CAMAndOr: An Implementation of Logic Programming Exploring Coarse and Fine Grain Parallelism," CASE Center Report 8815, Syracuse University, Syracuse, NY, Dec.

Ribler, Randy L., 1987. "The Integration of the Xenologic X-1 Artificial Intelligence Coprocessor with General Purpose Computers," *Proceedings Spring COMPCON,* San Francisco, Feb., pp. 403–407.

Robinson, J. Alan, 1965. "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM,* Dec., pp. 23–41.

Robinson, J. A., and K. J. Greene, 1987. "New Generation Knowledge Processing: Final Report on the SUPER System," CASE Center Report TR 8707, Syracuse University, Syracuse, NY.

Robinson, J. A., and E. E. Sibert, 1980. "Logic Programming in LISP," Technical Report, School of Computer and Information Science, Syracuse University, Syracuse, NY.

Robison, A. D., 1987. "Illinois Functional Programming: A Tutorial," *Byte,* Feb., pp. 115–125.

Rosser, J. Barkley, 1982/1984. "Highlights of the History of the Lambda-Calculus," *ACM Symposium on LISP and Functional Programming,* ACM No. 552820, Pittsburgh, Aug. 15–18, 1982, pp. 216–225. Also see *Annals of the History of Computing,* Vol. 6, No. 4, Oct. 1984, pp. 337–349.

Sannella, D. T., 1981. "HOPE Update: February 1981," CS Report LA-2593, University of Edinburgh, Edinburgh, Scotland, Feb.

Schoenfinkel, M., 1929. "Ueber die Bausteine der Mathematischen Logik," *Mathematische Annalen,* Vol. 92, pp. 305–316.

Schorr, H., and W. M. Waite, 1968. "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures," *Communications of the ACM,* Vol. 10, No. 8, p. 501.

Schwartz, J. T., R. B. K. Dewar, E. Dubinsky, and E. Schonberg, 1986. *Programming with Sets—An Introduction to SETL,* Springer-Verlag, New York.

Seldin, J. P., and J. R. Hindley, 1980. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism,* Academic Press, New York.

Sergot, M. J., F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory, 1986. "The British Nationality Act as a Logic Program," *Communications of the ACM,* Vol. 29, No. 5, May, pp. 370–386.

Shankar, Subash, 1988. "A Hierarchical Associative Memory Architecture for Logic Programming Unification," *Proceedings 5th Conference on Logic Programming,* MIT Press, Cambridge, MA, pp. 1428–1447.

Shapiro, Elud Y., 1983. "A Subset of Concurrent PROLOG and Its Interpreter," Technical Report TR-203, ICOT, Minato-ku, Toyoko 108, Japan.

Shapiro, Elud Y., 1986. "Concurrent PROLOG: A Progress Report," *IEEE Computer,* Aug., pp. 44–58.

Shapiro, Elud Y., 1987. *Concurrent PROLOG Collected Papers,* Vols. 1 and 2, MIT Press, Cambridge, MA.

Shapiro, Elud, 1989. "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys,* Vol. 21, No. 3, Sept., pp. 412–510.

Shemer, Jack, and Philip Neches, 1984. "The Genesis of a Database Computer," *IEEE Computer,* Nov., pp. 42–56.

Shen, Kish, and David H. D. Warren, 1987. "A Simulation Study of the Argonne Model for OR-Parallel Execution of PROLOG," *Proceedings IEEE Symposium on Logic Programming,* San Francisco, Aug., pp. 54–67.

Shustek, L. J., 1978. "Analysis and Performance of Computer Instruction Sets," Ph.D. thesis, Stanford University, Stanford, CA.

Singhal, Ashok, 1989. "A High Performance PROLOG Processor with Multiple Function Units," *Proceedings 16th International Conference on Computer Architecture,* Jerusalem, June, pp. 195–202.

Singhal, Ashok, and Yale Patt, 1989. "Unification Parallelism: How Much Can We Exploit?" *Proceedings North American Conference on Logic Programming,* Cleveland, OH, Oct., pp. 1135–1147.

Smith, D. C. P., and J. M. Smith, 1979. "Relational Data Base Machines," *IEEE Computer,* Vol. 12, No. 3, Mar., pp. 28–39.

Smith, Sarah, 1984. *The LMI LAMBDA Technology Summary*, LMI, Inc., Los Angeles.

Sohi, Gurindar S., Edward S. Davidson, and Janak H. Patel, 1985. "An Efficient LISP-Execution Architecture with a New Representation for List Structures," *Proceedings 12th Symposium on Computer Architecture*, Boston, June 17–19, pp. 91–99.

Spring, George, and Daniel D. Friedman, 1990. *Scheme and the Art of Programming*, McGraw-Hill/MIT Press, Cambridge, MA.

Steele, Guy L., 1978. *Rabbit: A Compiler for Scheme*, AI Laboratory Report AI-TR-474, Massachusetts Institute of Technology, Cambridge, MA.

Steele, Guy L., 1984. *Common LISP—The Language*, Digital Press, Bedford, MA.

Steele, Guy L., and Sussman, Gerald J., 1978. "The Revised Report of Scheme, a Dialect of LISP," MIT AI Memo 452, Jan.

Steenkiste, Peter, and John Hennessy, 1986. "LISP on a Reduced-Instruction-Set-Processor," *Proceedings ACM Conference on LISP and Functional Programming Language*, Boston, Aug., pp. 192–201.

Steenkiste, Peter, and John Hennessy, 1987. "Tags and Type Checking in LISP: Hardware and Software Approaches," *ACM Conference on Architecture Support for Programming Languages and Operating Systems*, Palo Alto, CA, Oct., pp. 50–59.

Steinberg, S., D. Allen, L. Bagnall, and C. Scott, 1986. "The Butterfly LISP System," *Proceedings American Association of Artificial Intelligence*, Philadelphia, Aug., p. 730.

Stolfo, Salvatore J., 1984. "Five Parallel Algorithms for Production System Execution on the DADO Machine," *Proceedings American Association on Artificial Intelligence*, Austin, TX, Aug., AAAI, pp. 300–307.

Stone, Harold S., 1987. "Parallel Querying of Large Databases: A Case Study," *IEEE Computer*, Oct., pp. 11–21.

Stormon, Charles D., 1986, 1988. *An Associative Processor and Its Application to Logic Programming Considerations*, Syracuse University CASE Report 8611, Oct. 1986, revised Jan. 1988.

Stormon, C. D., M. R. Brule, J. V. Oldfield, and J. C. Ribeiro, 1988. "An Architecture Based on Content-Addressable Memory for the Rapid Execution of PROLOG," *Proceedings 5th Conference on Logic Programming*, Seattle, WA, Aug., MIT Press, Cambridge, MA, pp. 1448–1474.

Stoy, Joseph E., 1977/1981. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA.

Stoye, R. S., 1985. *The Implementation of Functional Languages Using Custom Hardware*, Ph.D. dissertation, Cambridge University, Cambridge, England.

Sussman, G. J., J. Holloway, G. Steele, and A. Bell, 1981. "Scheme-79—LISP on a Chip," *IEEE Computer*, July, pp. 10–21.

Symbolics, Inc., 1984. *Symbolics 3600 Technical Summary*, Symbolics, Inc., Cambridge, MA.

Taki, K., 1986. "The Parallel Software Research and Development Tool: Multi-PSI System, *Proceedings France-Japan AI and Computer Science Symposium*, ICOT, Tokyo, Japan, Oct., pp. 365–381.

Taki, K., K. Nakajima, H. Nakashima, and M. Ikeda, 1987. "Performance and Architectural Evaluation of the PSI Machine," *Proceedings Second International Conference on Architecture Support for Programming Language and Operating Systems*, Palo Alto, CA, Oct., pp. 128–135.

Taylor, George S., Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn, 1986. "Evaluation of the SPUR LISP Architecture," *Proceedings 13th Symposium on Computer Architecture*, Tokyo, Japan, June, pp. 444–452.

Teitelman, Warren, 1978. *INTERLISP Reference Manual*, XEROX Corp., Palo Alto, CA.

Texas Instruments, Inc., 1984. *Explorer Technical Summary*, Texas Instruments, Inc., Austin, TX.

Tick, Evan, 1983. "An Overlapped PROLOG Processor," *Technical Note 308*, SRI International, Menlo Park, CA, Oct.

Tick, Evan, 1987. *Memory Performance of PROLOG Architectures*, Kluwer Academic, Norwell, MA.

Tick, Evan, 1988. "Data Buffer Performance for Sequential PROLOG Architectures," *Proceedings 15th Symposium on Computer Architecture*, Honolulu, HI, May, pp. 434–442.

Tick, E., and D. H. D. Warren , 1984. "Towards a Pipelined PROLOG Processor," *IEEE International Symposium on Logic Programming*, Aug., pp. 29–41.

Touati, H., and A. Despain, 1987. "An Empirical Study of the Warren Abstract Machine," *Proceedings 1987 Symposium on Logic Programming*, Seattle, WA, Aug., pp. 114–124.

Treleaven, P. C., D. R. Brownbridge, and R. P. Hopkins, 1982. "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys*, Vol. 14, No. 1, Mar., pp. 93–143.

Turk, Andrew W., 1986. "Compiler Optimization for the WAM," *Proceedings 3rd Conference on Logic Programming*, Springer-Verlag, New York, pp. 657-662.

Turner, D. A., 1979a. "Another Algorithm for Bracket Abstraction," *Journal of Symbolic Logic*, Vol. 44, No. 2, June, pp. 267–270.

Turner, D. A., 1979b. "A New Implementation Technique for Applicative Languages," *Software—Practice and Experience*, Vol. 9, pp. 31–49.

Ullman, Jeffrey, 1982. *Principles of DataBase Systems*, 2d ed., Computer Science Press, Rockville, MD.

Van Roy, P., 1984. "A PROLOG Compiler for the PLM," UCE/CSD 84/203, Computer Science Division, University of California—Berkeley, Nov.

Van Wijngaarden, et al., 1975. "Revised Report on the Algorithmic Language ALGOL68," *Acta Informatica*, Vol. 5, pp. 1–236.

Vegdahl, S. R., 1984. "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Transactions on Computers*, Vol. C-33, No. 12, Dec., pp. 1050–1071.

Veen, Arthur H., 1986. "Dataflow Machine Architecture," *ACM Computing Surveys*, Vol. 18, No. 4, Dec., pp. 365–397.

Wadler, P. L., 1976. "Analysis of an Algorithm for Real-Time Garbage Collection," *Communications of the ACM*, Vol. 19, No. 9, Sept., pp. 491–500.

Warren, David H., 1977. "Implementing PROLOG—Compiling Predicate Logic Programs," Vols. 1 and 2, Reports Nos. 39 and 40, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, May.

Warren, David H., 1983. "An Abstract PROLOG Instruction Set," *Technical Note 309*, SRI International, Menlo Park, CA, Oct.

Warren, David H. D., 1987. "The SRI Model for OR-Parallel Execution of PROLOG—Abstract Design and Implementation Issues," *Proceedings IEEE Symposium on Logic Programming*, San Francisco, CA, Aug., pp. 92–102.

Warren, David H. D., and F. C. N. Periera, 1981. "An Efficient Easily Adaptable System for Interpreting Natural Language Queries," *DAI Research Paper No. 155*, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

Warren, David S., 1989. "The XWAM: A Machine That Integrates PROLOG and Deductive Database Query Evaluation," invited talk at the North American Conference on Logic Programming, Cleveland, OH, Oct. Also see SUNY Stony Brook, Computer Science Department, Technical Report 89/25, Stony Brook, NY, Oct. 23, 1989.

Watson, Ian, and John Gurd, 1979. "A Prototype Data Flow Computer with Token Labelling," *Proceedings AFIPS National Computer Conference*, New York, June, pp. 623–628.

Watson, Ian, and John Gurd, 1982. "A Practical Data Flow Computer," *IEEE Computer*, Vol. 15, No. 2, Feb., pp. 51–57.

Wegner, P., 1972. "The Vienna Definition Language," *Computing Surveys*, Vol. 4, No. 1, Mar., pp. 5–63.

Weissman, Clark, 1967. *LISP 1.5 Primer*, Dickenson Press, Belmont, CA.

Westphal, H., and P. Robert, 1987. "The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism," *Proceedings IEEE Symposium on Logic Programming*, San Francisco, Aug., pp. 436–448.

Wilensky, Robert, 1984. *LISPcraft,* W. W. Norton, New York.

Winston, Patrick, and Berthold Horn, 1984. *LISP,* Addison-Wesley, Reading, MA.

Wirth, Niklaus, 1977. "What Can We Do about the Unnecessary Diversity of Notation for Syntatic Definitions?" *Communications of the ACM,* Vol. 20, No. 11, Nov., pp. 822–823.

Wolfe, Alexander, 1987. "TI Puts Its LISP Chip into a System for Military AI," *Electronics,* Mar., pp. 95–96.

Wong, Kam-Fai, and M. Howard Williams, 1989. "A Type Driven Hardware Engine for PROLOG Clause Retrieval over a Large Knowledge Base," *Proceedings 16th International Symposium on Computer Architecture,* Jerusalem, May, pp. 211–222.

Woo, N. S., 1985. "A Hardware Unification Unit: Design and Analysis," *12th International Symposium on Computer Architecture,* Boston, June, pp. 198–207.

Wos, Larry, R. Overbeek, E. Lusk, and J. Boyle, 1984. *Automated Reasoning: Introduction and Applications,* Prentice-Hall, Englewood Cliffs, NJ.

Yuhara, Masanobu, Aikara Hattori, Mitsuhiro Kishimoto, and Hiromu Hayashi, 1986. "Evaluation of the FACOM ALPHA LISP Machine," *Proceedings 13th Symposium on Computer Architecture,* Tokyo, Japan, June, pp. 184–190.

Yung, Chao-Chih, 1986. *Relational Databases,* Prentice-Hall, Englewood Cliffs, NJ.

Zloof, M., 1977. "Query-by-Example: A Data Base Language," *IBM Systems Journal,* Vol. 16, No. 4, pp. 324–343.

INDEX