Übersetzerbau VU
Übungsskriptum

Anton Ertl
Andreas Krall

2017

- Allgemeines und Beispiele
- GNU Emacs Reference Card
- AMD64-Assembler Handbuch
- make: A Program for Maintaining Programs
- lex — a Lexical Analyzer Generator
- yacc — Yet Another Compiler-Compiler
- Ox: Tutorial Introduction
- `burg`, `iburg` und `bfe`

1 Anmeldung

Melden Sie sich in unserem Anmeldesystem `https://www.complang.tuwien.ac.at/anmeldung/` für die Lehrveranstaltung an. Mit der Anmeldung wird ein Account für Sie auf unserer Übungsmaschine `g0.complang.tuwien.ac.at` eingerichtet, der Accountname ist u gefolgt von der Matrikelnummer, z.B. u9999999. Das Passwort für diesen Account geben Sie bei der Anmeldung ein.

2 Rechner

Die Übungsmaschine ist `g0.complang.tuwien.ac.at`; sollte sie längerfristig ausfallen, steht als Ersatzmaschine `g2.complang.tuwien.ac.at` zur Verfügung (Sie können sich aber vorerst nicht auf die Ersatzmaschine einloggen). Sie können sich aus dem Internet mit `ssh g0.complang.tuwien.ac.at` einloggen; falls Sie sich von einem Windows-Client aus einloggen wollen, können Sie das mit Putty tun.

Falls Sie Ihre Programme woanders entwickeln, müssen Sie Ihre Dateien für die Abgabe mit `scp` (eine ssh-Anwendung) auf unsere Rechner übertragen. Rufen Sie dann unbedingt das Test-Skript für das Beispiel *auf der Übungsmaschine* auf, damit Sie eventuelle Fehler mit katastrophalen Auswirkungen bemerken (wie z.B. das Kopieren in das falsche Verzeichnis).

Die in der Übung verwendeten Werkzeuge sind für verschiedene Plattformen auf `http://www.complang.tuwien.ac.at/ubvl/tools/` erhältlich (allerdings recht veraltet).

Wenn Sie selbst ein `.forward`-File einrichten oder ändern, testen Sie es unbedingt! Wenn es nicht funktioniert, haben wir keine Möglichkeit, Sie zu erreichen (z.B. um Ihnen die Ergebnisse der Abgabe mitzuteilen).

3 Betreuung, Information

Im WWW finden Sie unter `http://www.complang.tuwien.ac.at/ubvl/` Informationen zur Übung.

Verlautbarungen zur Übung (z.B. Klarstellungen zur Angabe) gibt es im Informatikforum (Details dazu siehe Übungshomepage).

Wenn Sie eine Frage zur Übung haben, stellen Sie sie am besten im Informatikforum (dann können auch andere antworten oder von der Antwort profitieren). Sie können auch den Leiter der Übung per Email fragen `anton@mips.complang.tuwien.ac.at`, oder in einer Sprechstunde (Termin per Email vereinbaren).

Technische Probleme wie Computerabstürze, falsche Permissions, oder vergessene Passwörter sind eine Sache für den Techniker. Wenden Sie sich di-

rekt an ihn: email an Herbert Pohlai (herbert@mips.complang.tuwien.ac.at), Tel. (+43-1) 58801/18525.

4 Beispiele

Die Beispiele finden Sie weiter hinten im Skriptum. Beachten Sie, dass die ersten Beispiele erfahrungsgemäß wesentlich leichter sind als die Beispiele „Attributierte Grammatik" bis „Gesamtbeispiel". Versuchen Sie, mit den ersten Beispielen möglichst rasch fertig zu werden, um genügend Zeit für die Schwierigeren zu haben.

5 Beurteilung

Ihre Note wird aufgrund der Qualität der von Ihnen abgegebenen Programme ermittelt. Das Hauptkriterium ist dabei die Korrektheit. Sie wird mechanisch überprüft, Sie erhalten per Email das Ergebnis der Prüfung. Wenn Sie meinen, dass sich das Prüfprogramm geirrt hat, wenden Sie sich an den Leiter der Übung.

Die Prüfprogramme sind relativ einfach, dumm und kaum fehlertolerant. Damit Sie prüfen können, ob Ihr Programm im richtigen Format ausgibt und ähnliche wichtige Kleinigkeiten, stehen Ihnen die Testprogramme und einige einfache Testeingaben und -resultate zur Verfügung; Sie können die Testprogramme auch benutzen, um Ihre Programme mit eigenen Testfällen zu prüfen (siehe http://www.complang.tuwien.ac.at/ubvl/).

Beachten Sie, dass bei der Abgabe die Überprüfung mit wesentlich komplizierteren Testfällen erfolgt als denen, die wir Ihnen vorher zur Verfügung stellen (vor allem ab dem Scanner-Beispiel). Ein erfolgreiches Absolvieren der Ihnen vorher zur Verfügung stehenden Tests heißt also noch lange nicht, dass Ihr Programm korrekt ist. Sie müssen sich selbst weitere Testfälle überlegen (wie auch im Berufsleben).

Ihre Programme werden zu den angegebenen Terminen kopiert und später überprüft. Ändern Sie zu den Abgabeterminen zwischen 14h und 15h nichts im Abgabeverzeichnis, damit es nicht zu inkonsistenten Abgaben kommt.

Ein paar Tage nach der Abgabe erhalten Sie das Ergebnis per Email. Das Ausschicken der Ergebnisse wird auch im LVA-Forum verkündet, Sie brauchen also nicht nachfragen, wenn Sie dort noch nichts gesehen haben. Eine Arbeitswoche nach der ersten Abgabe werden Ihre (eventuell von Ihnen verbesserten) Programme erneut kopiert und überprüft. Diese Version wird mit 70% der Punkte eines rechtzeitig abgegebenen Programms gewertet. Das ganze wiederholt sich zwei Arbeitswochen nach dem ersten Abgabetermin (30% der Punkte). Sie erhalten für das Beispiel das Maximum der drei Ergebnisse.

Name	online Doku	Bemerkung
emacs, vi	info emacs, man vi	Editor
gcc	info as	Assembler
gcc	info gcc	C-Compiler
make	info make	baut Programme
flex	man flex	Scanner-Generator
yacc, bison	man yacc, info bison	Parser-Generator
xvcg	man xvcg	Graphenzeichnen
ox	man ox	AG-basierter
	xdvi /usr/ftp/pub/ubvl/oxURM.dvi	Compilergenerator
burg, iburg	man iburg, man burg	Baumparser-Generator
bfe	Skriptum	Präprozessor für burg
gdb	info gdb	Debugger
objdump	info objdump	Disassembler etc.
mutt, mail	man mutt, man mail	Email
xrn	man xrn	Newsreader
lynx, mozilla firefox		WWW-Browser

Abbildung 1: Werkzeuge

Sollten Sie versuchen, durch Kopieren oder Abschreiben von Programmen eine Leistung vorzutäuschen, die Sie nicht erbracht haben, erhalten Sie keine positive Note. Die Kontrolle erfolgt in einem Gespräch am Ende des Semesters, in dem überprüft wird, ob Sie auch verstehen, was Sie abgegeben haben. Weitere Maßnahmen behalten wir uns vor.

Ihr Account ist nur für Sie lesbar. Bringen Sie andere nicht durch Ändern der Permissions in Versuchung, zu schummeln.

6 Weitere Dokumentation bzw. Werkzeuge

Abbildung 1 zeigt die zur Verfügung stehenden Werkzeuge.

Die mit „man" gekennzeichnete Dokumentation können Sie lesen, indem sie auf der Kommandozeile `man ...` eintippen. Die mit „info" gekennzeichnete Dokumentation können Sie mit dem Programm `info` lesen, oder indem sie in Emacs `C-h i` tippen. In der Dokumentation für Emacs bedeutet `C-`x Ctrl x und `M-`x Meta x (auf den Übungsgeräten also Alt x).

Alle Werkzeuge rufen Sie von der Shell-Kommandozeile aus auf, indem Sie ihren Namen tippen.

Mit flex erzeugte Scanner müssen normalerweise mit `-lfl` gelinkt werden.

Das auf den Übungsgeräten unter `yacc` aufrufbare Programm ist `bison -y`

(für den Fall, dass Sie Diskrepanzen zwischen diesem yacc und dem auf kommerziellen Unices bemerken). Mit `xvcg` können Sie sich die Ausgabe von `bison -g` anschauen.

`mail` ist ein primitives Email-Werkzeug, `mutt` ist etwas bequemer[1].

Das Ox User Reference Manual ist nicht in diesem Skriptum abgedruckt, sondern steht nur on-line zur Verfügung, da es relativ umfangreich ist und nur ein Teil der enthaltenen Information in dieser Übung nützlich ist.

7 Beispiele

Es sind insgesamt acht Beispiele abzugeben. Die ersten beiden Beispiele dienen dem Erlernen von Konzepten von Computerarchitekturen am Beispiel der AMD64-Architektur. In den weiteren Beispielen wird eine Programmiersprache vollständig implementiert. Diese Beispiele bauen aufeinander auf, d.h. Fehler, die Sie in den ersten Sprachimplementierungsbeispielen machen, sollten Sie beheben, damit sie in späteren Abgaben die Beurteilung nicht verschlechtern. Bei der Implementierung der Sprache wird mit jedem Beispiel (ausgenommen die letzten) auch ein neues Werkzeug eingeführt, das nach Einarbeitung in die Verwendungsweise des Werkzeugs die Arbeit erleichtert.

Die zu implementierende Sprache ist eingeschränkt, um den Arbeitsaufwand nicht zu groß werden zu lassen. So sind in dieser Sprache zwar grundlegende Kontrollstrukturen vorhanden und es können Variablen definiert werden, aber Speicher für Datenstrukturen kann nicht innerhalb dieser Sprache angefordert werden. Sprachkonstrukte für Speicherzugriffe sind jedoch vorhanden. Daher müssen bei den letzten Beispielen, um die Codegenerierung testen zu können, Datenstrukturen in einem C-Programm erzeugt werden und dann mit dem von Ihnen generierten Code gelinkt werden. Dadurch erlernen Sie auch, wie verschiedene Sprachen miteinander kombiniert werden können. Weiters gibt es keine direkte Möglichkeit, Daten ein- oder auszugeben; diese Funktionen werden durch eine C-Funktion übernommen, die Funktionen in der Sprache aufruft, oder durch Aufrufen von C-Funktionen aus Funktionen in unserer Sprache.

Die Kenntnisse, die Sie bei den Assembler-Beispielen erlangen, werden Sie auch wieder bei der Codegenerierung der letzten Beispiele verwenden. Die Beispiele 3-8 können alle aufeinander aufbauend implementiert werden, d.h. wenn Sie Ihr Programm von Anfang an gut entwerfen, können Sie dieses ab dem Scanner-Beispiel bis zum Gesamtbeispiel stets wiederverwenden und erweitern. Beachten Sie jedoch, dass bei jeder Abgabe stets das gesamte

[1] `mutt` und `xrn` sind so vor-eingestellt, dass der schon laufende Emacs als Editor verwendet wird. Wenn Sie mit dem Editieren des Buffers fertig sind, tippen Sie `C-x #` und `mutt/xrn` wird weitermachen.

Quellprogramm im Abgabeverzeichnis vorhanden sein muss (und zwar nicht in Form von symbolic links).

In den folgenden Abschnitten finden Sie die Angaben und Erklärungen für die Modalitäten der Beispielabgaben. Von der Sprache wird in jedem Abschnitt immer nur soviel erklärt, wie für das jeweilige Beispiel notwendig ist. Wenn Sie einen Überblick über die gesamte Sprache haben wollen, sollten Sie sich gleich am Anfang alle Angaben durchlesen.

In dieser Sprache kann man, wie in vielen Programmiersprachen, auch Programme schreiben, deren Semantik nicht definiert ist, und die Ihr Compiler trotzdem nicht als fehlerhaft erkennen muss und darf. Bei solchen Programmen ist es egal, welchen Code Ihr Compiler produziert (Code aus solchen Testeingaben wird von unseren Abgabescripts ohnehin nicht ausgeführt). Ihr Compiler sollte aber für Programme mit definierter Semantik korrekten Code produzieren.

7.1 Assembler A

7.1.1 Termin

Abgabe spätestens am 15. März 2017, 14 Uhr.

7.1.2 Angabe

Wenn man eine (vorzeichenlose) zweistellige Zahl x_1x_0 mit einer (vorzeichenlosen) einstelligen Zahl y multipliziert, kann man das Ergebnis als $(x_1b + x_0)y = x_1yb + x_0y$ darstellen, wobei b die Basis ist. Da man die Multiplikation mit b durch Anhängen einer 0 erreichen kann, braucht man dafuer nur Multiplikationen einer einstelligen Zahl mit einer einstelligen Zahl, und die Addition mehrstelliger Zahlen. Wenn wir die einzelnen Ziffern $r_2r_1r_0$ des Ergebnisses r berechnen wollen, kommen wir auf:

$$\begin{aligned} r_0 &= (x_0y)_0 \\ r_1 &= ((x_1y)_0 + (x_0y)_1)_0 \\ r_2 &= (x_1y)_1 + c \end{aligned}$$

wobei c der Überlauf der Addition aus der Addition bei der Berechnung von r_1 ist.

Wenn wir nun die Multiplikation einer 128-bit-Zahl mit einer 64-bit-Zahl auf der AMD64-Architektur implementieren wollen, die eine 64-bit×64-bit-Multiplikation bietet, entspricht das genau diesem Modell, mit $b = 2^{64}$.

Schreiben Sie eine Funktion asma in Assembler, die eine vorzeichenlose 128-bit-Zahl in `%rdi` (untere 64 bits) und `%rsi` (obere 64 bits), eine vorzeichenlose 64-bit-Zahl in `%rdx`, und eine Adresse a in `%rcx` übergeben bekommt, die beiden Zahlen multipliziert, und das 192-bit-Ergebnis in den 24 Bytes an den Adressen $a...a+23$ abspeichert, mit dem niederwertigsten Byte an der Adresse a (little-endian). Verwenden Sie in dieser Funktion die Befehle `mul` (die Variante mit einem expliziten Argument, die ein 128-bit-Resultat liefert, siehe AMD64-Assembler-Handbuch in diesem Skriptum) und `adc`.

Am einfachsten tun Sie sich dabei wahrscheinlich, wenn Sie eine einfache C-Funktion wie

```
void asma(unsigned long x0, unsigned long x1, unsigned long y,
          unsigned long *a)
{
}
```

mit z.B. `gcc -O -S` in Assembler übersetzen und sie dann verändern. Dann stimmt schon das ganze Drumherum.

7.1.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version `mul` und `adc` verwendet und korrekt ist, also bei gleicher (zulässiger) Eingabe das gleiche Resultat liefert wie das Original.

Zum Assemblieren und Linken verwendet man am besten `gcc`, der Compiler-Treiber kümmert sich dann um die richtigen Optionen für `as` und `ld`.

7.1.4 Abgabe

Zum angegebenen Termin stehen im Verzeichnis `~/abgabe/asma` die maßgeblichen Dateien. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und `make` soll eine Datei `asma.o` erzeugen. Diese Datei soll nur die Funktion `asma` enthalten, keinesfalls `main`. Diese Funktion soll den Aufrufkonventionen gehorchen und wird bei der Prüfung der abgegebenen Programme mit C-Code zusammengebunden.

7.2 Assembler B

7.2.1 Termin

Abgabe spätestens am 22. März 2017, 14 Uhr.

7.2.2 Angabe

Schreiben Sie eine Funktion mit dem C-Prototyp in Assembler (entsprechend der AMD64-System-V-ABI-Aufrufkonvention, siehe AMD64-Assembler-Handbuch):

```
void asmb(unsigned long *x, size_t n, unsigned long y,
          unsigned long *a)
```

die eine $64n$-bit-Zahl im Speicher beginnend an der Adresse x und eine 64-bit Zahl in y (beide vorzeichenlos) miteinander multipliziert und das $64(n+1)$-bittige Ergebnis in den Speicher beginnend mit der Adresse a schreibt. Sowohl die Zahl bei x als auch bei a haben ihr niederwertigstes Byte an der Adresse x bzw. a (little-endian). Verwenden Sie dafür u.a. die Befehle `mul` und `adc`.

Für besonders effiziente Lösungen (gemessen an der Anzahl der *ausgeführten* Maschinenbefehle; wird ein Befehl n mal ausgeführt, zählt er n-fach) gibt es Bonuspunkte. Dabei könnte Ihnen das Wissen helfen, dass `add` und `sub` das Carry-Flag verändern, `inc`, `dec`, `lea` und `mov` dagegen nicht.

7.2.3 Hinweis

Beachten Sie, dass Sie nur dann Punkte bekommen, wenn Ihre Version korrekt ist, also bei jeder zulässigen Eingabe das gleiche Resultat liefert wie das Original. Dadurch können Sie viel mehr verlieren als Sie durch Optimierung gewinnen können, also optimieren Sie im Zweifelsfall lieber weniger als mehr.

Die Vertrautheit mit dem Assembler müssen Sie beim Gespräch am Ende des Semesters beweisen, indem Sie Fragen zum abgegebenen Code beantworten.

7.2.4 Abgabe

Zum angegebenen Termin stehen im Verzeichnis `~/abgabe/asmb` die maßgeblichen Dateien. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und `make` soll eine Datei `asmb.o` erzeugen. Diese Datei soll nur die Funktion `asmb` enthalten, keinesfalls `main`. Diese Funktion soll den Aufrufkonventionen gehorchen und wird bei der Prüfung der abgegebenen Programme mit C-Code zusammengebunden.

7.3 Scanner

7.3.1 Termin

Abgabe spätestens am 29. März 2017, 14 Uhr.

7.3.2 Angabe

Schreiben Sie mit `flex` einen Scanner, der Identifier, Zahlen, und folgende Schlüsselwörter unterscheiden kann: `end return goto if var and not`. Weiters soll er auch noch folgende Lexeme erkennen: `; ( ) , : = != > [ ] - + *`

Identifier bestehen aus Buchstaben und Ziffern, dürfen aber nicht mit Buchstaben beginnen. Zahlen bestehen aus Ziffern und _, wobei die _ ignoriert werden. Zahlen werden als Dezimalzahlen interpretiert. Leerzeichen, Tabs und Newlines zwischen den Lexemen sind erlaubt und werden ignoriert, ebenso Kommentare, die mit `(*` anfangen und bis zum nächsten `*)` gehen; Kommentare können also nicht geschachtelt werden. Alles andere sind lexikalische Fehler. Es soll jeweils das längste mögliche Lexem erkannt werden, `if39` ist also ein Identifier (longest input match), `39_if` ist die Zahl `39_` gefolgt vom Schlüsselwort `if`.

Der Scanner soll für jedes Lexem eine Zeile ausgeben: für Schlüsselwörter und Lexeme aus Sonderzeichen soll das Lexem ausgegeben werden, für Identifier `id` gefolgt von einem Leerzeichen und dem String des Identifiers, für Zahlen `num` gefolgt von einem Leerzeichen und der Zahl in Dezimaldarstellung ohne führende Nullen und ohne _. Für Leerzeichen, Tabs, Newlines und Kommentare soll nichts ausgegeben werden (auch keine Leerzeile).

Der Scanner soll zwischen Groß- und Kleinbuchstaben unterscheiden, `End` ist also kein Schlüsselwort.

7.3.3 Abgabe

Legen Sie ein Verzeichnis `~/abgabe/scanner` an, in das Sie die maßgeblichen Dateien stellen. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können (auch den ausführbaren Scanner) und mittels `make` ein Programm namens `scanner` erzeugen, das von der Standardeingabe liest und auf die Standardausgabe ausgibt. Korrekte Eingaben sollen akzeptiert werden (Ausstieg mit Status 0, z.B. mit `exit(0)`), bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden. Bei einem lexikalischen Fehler darf der Scanner Beliebiges ausgeben (eine sinnvolle Fehlermeldung hilft bei der Fehlersuche).

7.4 Parser

7.4.1 Termin

Abgabe spätestens am 5. April 2017, 14 Uhr.

```
Program: { Funcdef ';' }
       ;

Funcdef: id '(' Pars ')' Stats end  /* Funktionsdefinition */
       ;

Pars: { id ',' } [ id ]             /* Parameterdefinition */
    ;

Stats: { { Labeldef } Stat ';' }
     ;

Labeldef: id ':'                    /* Labeldefinition */
        ;

Stat: return Expr
    | goto id
    | if Cond goto id
    | var id '=' Expr               /* Variablendefinition */
    | Lexpr '=' Expr                /* Zuweisung */
    | Term
    ;


Cond: Cterm { and Cterm }
    | not Cterm
    ;

Cterm: '(' Cond ')'
     | Expr '!=' Expr
     | Expr '>' Expr
     ;

Lexpr: id                 /* schreibender Variablenzugriff */
     | Term '[' Expr ']' /* schreibender Arrayzugriff */
     ;

Expr: Term { '+' Term }
    | Term { '*' Term }
    | { '-' } Term
    ;

Term: '(' Expr ')'
    | num
    | Term '[' Expr ']' /* lesender Arrayzugriff */
    | id                /* lesender Variablenzugriff */
    | id '(' { Expr ',' } [ Expr ] ')' /* Funktionsaufruf */
    ;
```

Abbildung 2: Grammatik; Terminale sind klein geschrieben bzw. mit Hochkomma umschlossen; Nonterminale sind groß geschrieben.

7.4.2 Angabe

Abbildung 2 zeigt die Grammatik (in `yacc/bison`-artiger EBNF) einer Programmiersprache. Schreiben Sie einen Parser für diese Sprache mit `flex` und `yacc/bison`. Die Lexeme sind die gleichen wie im Scanner-Beispiel (`id` steht für einen Identifier, `num` für eine Zahl). Das Startsymbol ist `Program`.

7.4.3 Abgabe

Zum angegebenen Termin stehen im Verzeichnis `~/abgabe/parser` die maßgeblichen Dateien. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `parser` erzeugen, das von der Standardeingabe liest. Korrekte Programme sollen akzeptiert werden (Ausstieg mit Status 0, z.B. mit `exit(0)`), bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei Syntaxfehlern der Fehlerstatus 2. Das Programm darf auch etwas ausgeben (auch bei korrekter Eingabe), z.B. damit Sie sich beim Debugging leichter tun.

7.4.4 Hinweis

Beseitigen Sie alle Konflikte in der Grammatik durch Umformen der Grammatik! `yacc` löst solche Konflikte zwar auf, aber nicht unbedingt in der von Ihnen gewünschten Art. Um solche Konflikte zu verstehen, rufen Sie `yacc/bison` mit `-v` auf und schauen Sie sich die `.output`-Datei an.

Die Verwendung von Präzedenzdeklarationen von `yacc/bison` kann leicht zu Fehlern führen, die man nicht so schnell bemerkt (bei dieser Grammatik sind sie sowieso sinnlos).

Links- oder Rechtsrekursion? Also: Soll das rekursive Vorkommen eines Nonterminals als erstes (links) oder als letztes (rechts) auf der rechten Seite der Regel stehen? Bei `yacc/bison` und anderen LR-basierten Parsergeneratoren funktioniert beides. Sie sollten sich daher in erster Linie danach richten, was leichter geht, z.B. weil es Konflikte vermeidet oder weil es einfachere Attributierungsregeln erlaubt. Z.B. kann man mittels Linksrekursion bei der Subtraktion einen Parse-Baum erzeugen, der auch dem Auswertungsbaum entspricht. Sollte es keine anderen Gründe geben, kann man der Linksrekursion den Vorzug geben, weil sie mit einer konstanten Tiefe des Parser-Stacks auskommt.

7.5 Attributierte Grammatik

7.5.1 Termin

Abgabe spätestens am 3. Mai 2017, 14 Uhr.

7.5.2 Angabe

Erweitern Sie den Parser aus dem letzten Beispiel mit Hilfe von `ox` um eine Symboltabelle und eine statische Analyse.

Die *hervorgehobenen* Begriffe beziehen sich auf Kommentare in der Grammatik.

7.5.2.1 Namen. Die folgenden Dinge haben Namen: Funktionen, Labels und Variablen.

Eine Funktion wird im *Funktionsaufruf* verwendet und in der *Funktionsdefinition* definiert. Verwendete Funktionen müssen nicht definiert werden und können nicht deklariert[2] werden. Funktionen dürfen, soweit es den Compiler betrifft, doppelt definiert werden und dürfen den gleichen Namen wie Variablen oder Labels haben; daher muss der Compiler Funktionsnamen nicht in einer Symboltabelle verwalten. Auch die Übereinstimmung der Anzahl der Argumente soll (und kann) der Compiler nicht überprüfen.

Alle Namen (`ids`), die in einer *Parameterdefinition* oder in einer *Variablendefinition* vorkommen, sind Variablennamen. Variablen, die in einer Parameterdefinition definiert wurden, sind in der ganzen Funktion sichtbar. Variablen, die einer Variablendefinition definiert wurden, sind in den folgenden `Stats` sichtbar, aber nicht in der Variablendefinition selbst, oder davor.

Bei einem *Variablenzugriff* (schreibend oder lesend) muss eine Variable oder ein Parameter mit dem Namen sichtbar sein.

Ein Label wird in der `Labeldefinition` definiert und ist in der gesamten Funktion sichtbar, auch schon vor der Definition, und nirgendwo sonst. Der Identifier hinter einem `goto` (auch bei `if...goto`) muss ein definierter und sichtbarer Label sein.

Für jede Stelle gilt: Eine dort sichtbare Variable oder ein dort sichtbarer Label darf nicht den selben Namen haben wie eine andere dort sichtbare Variable oder ein anderer dort sichtbarer Label; das heisst auch, dass ein Label nicht den gleichen Namen haben darf wie eine Variable, die in der gleichen Funktion vorkommt.

Ihre statische Analyse soll alle diese Bedingungen prüfen, also dass ein Name nur einmal in einer Funktion definiert wird (nur Funktionsnamen müssen nicht geprüft werden), dass Variablenzugriffe auf sichtbare Variablen oder Parameter erfolgen, und dass das id hinter `goto` ein sichtbarer Label ist.

[2]Im Sinne von C: Die Definition einer Funktion enthält den vollständigen Code. Die Deklaration enthält nur die Informationen, die der Compiler braucht, um eine Typüberprüfung des Aufrufs durchzuführen (in C auch bekannt als Prototyp, in anderen Sprachen oft als Signatur).

7.5.3 Hinweise

Es ist empfehlenswert, die Grammatik so umzuformen, dass sie für die AG günstig ist: Fälle, die syntaktisch gleich ausschauen, aber bei den Attributierungsregeln verschieden behandelt werden müssen, sollten auf verschiedene Regeln aufgeteilt werden; umgekehrt sollten Duplizierungen, die in dem Bemühen vorgenommen wurden, Konflikte zu vermeiden, auf ihre Sinnhaftigkeit überprüft und ggf. rückgängig gemacht werden. Testen Sie Ihre Grammatikumformungen mit den Testfällen.

Offenbar übersehen viele Leute, dass attributierte Grammatiken Information auch von rechts nach links (im Ableitungsbaum) weitergeben können. Sie denken sich dann recht komplizierte Lösungen aus. Dabei reichen die von `ox` zur Verfügung gestellten Möglichkeiten vollkommen aus, um zu einer relativ einfachen Lösung zu kommen.

Verwenden Sie keine globalen Variablen oder Funktionen mit Seiteneffekten (z.B. Funktionen, die übergebene Datenstrukturen ändern) bei der Attributberechnung! `ox` macht globale Variablen einerseits unnötig, andererseits auch fast unbenutzbar, da die Ausführungsreihenfolge der Attributberechnung nicht vollständig festgelegt ist. Bei Traversals ist die Reihenfolge festgelegt, und Sie können globale Variablen verwenden; seien Sie aber trotzdem vorsichtig.

Sie brauchen angeforderten Speicher (z.B. für Symboltabellen-Einträge oder Typinformation) nicht freigeben, die Testprogramme sind nicht so groß, dass der Speicher ausgeht (zumindest wenn Sie's nicht übertreiben).

Das Werkzeug Torero (`http://www.complang.tuwien.ac.at/torero/`) ist dazu gedacht, bei der Erstellung von attributierten Grammatiken zu helfen.

7.5.4 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/ag`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `ag` erzeugen, das von der Standardeingabe liest. Korrekte Programme sollen akzeptiert werden, bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei Syntaxfehlern der Fehlerstatus 2, bei anderen Fehlern (z.B. Verwendung eines nicht sichtbaren Namens) der Fehlerstatus 3. Die Ausgabe kann beliebig sein, auch bei korrekter Eingabe.

7.6 Codeerzeugung A

7.6.1 Termin

Abgabe spätestens am 17. Mai 2017, 14 Uhr.

7.6.2 Angabe

Erweitern Sie die statische Analyse aus dem AG-Beispiel mit Hilfe von `iburg` zu einem Compiler, der folgende Untermenge der statisch korrekten Programme in AMD64-Assemblercode übersetzt: alle Programme, in denen aus `Stat` nur `return`-Anweisungen abgeleitet werden, in denen aber kein *Funktionsaufruf* abgeleitet wird. Programme, die statisch korrekt sind, aber dieser Einschränkung nicht entsprechen, werden bei diesem Beispiel nicht als Testeingaben vorkommen; statisch inkorrekte Programme müssen weiterhin als solche erkannt werden.

Ein Teil der Sprache wurde schon im Beispiel attributierte Grammatik erklärt, hier der für dieses Beispiel notwendige Zusatz:

7.6.2.1 Datendarstellung. Diese Programmiersprache kennt nur einen Datentyp: das 64-bit-Wort, das als vorzeichenbehaftete Zahl oder als Speicheradresse verwendet werden kann. Weder der Compiler noch das Laufzeitsystem soll eine Typüberprüfung vornehmen. Der Programmierer (der Anwender des Compilers) muss wissen, was er tut, der Compiler soll (und kann) das nicht überprüfen. Unsere Testprogramme führen keine Zugriffe auf ungültige Adressen aus.

7.6.2.2 Bedeutung der Operatoren. `+`, `-` und `*` haben ihre übliche Bedeutung (ein etwaiger Überlauf soll ignoriert werden).

Bei einem (schreibenden oder lesenden) *Arrayzugriff* ist der vom `Term` (auf der rechten Seite der Regel) berechnete Wert die Addresse von Element 0 des Arrays, und der von `Expr` berechnete Wert der Index des Elements, auf das zugegriffen wird. Der Zugriff erfolgt dann auf die Adresse Term+8*Expr. Der *lesende Arrayzugriff* liefert als Resultat den 64-bit-Wert an dieser Adresse, der *schreibende Arrayzugriff* überschreibt das 64-bit-Wort an dieser Stelle.

7.6.2.3 Anweisungen Die `return`-Anweisung beendet die Funktion und liefert das Resultat von `Expr` als Ergebnis des Aufrufs der Funktion.

7.6.2.4 Erzeugter Code. Ihr Compiler soll AMD64-Assemblercode ausgeben. Jede Funktion im Programm verhält sich gemäß der Aufrufkonvention. Der erzeugte Code wird nach dem Assemblieren und Linken von C-Funktionen aufgerufen. Beispiel: Die Funktion `foo(a,b) ... end;` kann von C aus mit `foo(x,y)` aufgerufen werden, wobei `a` den Wert von `x` bekommt und `b` den von `y`.

Der Name einer Funktion soll als Assembler-Label am Anfang des erzeugten Codes verwendet werden und das Symbol soll exportiert werden; andere Symbole soll Ihr Code nicht exportieren.

Folgende Einschränkungen sind dazu gedacht, Ihnen gewisse Probleme zu ersparen, die reale Compiler bei der Codeauswahl und Registerbelegung haben. Sie brauchen diese Einschränkungen nicht überprüfen, unsere Testeingaben halten sich an diese Einschränkungen (eine Überprüfung könnte Ihnen allerdings beim Debuggen Ihrer eigenen Testeingaben helfen): Funktionen haben maximal 6 Parameter. Die maximale Tiefe eines Ausdrucks[3] ist $\leq 6 - v$, wobei v die Anzahl der sichtbaren Parameter und Variablen ist. Die im Quellprogramm vorkommenden Zahlen und konstanten Ausdrücke sind $\geq -2^{31}$ und $< 2^{31}$; das gilt aber nicht für Ergebnisse von Berechnungen zur Laufzeit.

Der erzeugte Code soll korrekt sein und möglichst wenige Befehle ausführen (da es hier keine Verzweigungen gibt, ist das gleichbedeutend mit „wenige Befehle enthalten"). Dabei ist nicht an eine zusätzliche Optimierung (wie z.B. common subexpression elimination) gedacht, sondern vor allem an die Dinge, die Sie mit `iburg` tun können, also eine gute Codeauswahl (besonders bezüglich konstanter Operanden und Ausnutzung der Adressierungsarten) und, wenn Sie besonders ehrgeizig sind, einige algebraische Optimierungen (siehe z.B. `http://www.complang.tuwien.ac.at/papers/ertl00dagstuhl.ps.gz`). Für besonders effizienten erzeugten Code gibt es Sonderpunkte.

Beachten Sie, dass es leicht ist, durch eine falsche Optimierungsregel mehr Punkte zu verlieren, als Sie durch Optimierung überhaupt gewinnen können. Testen Sie daher ihre Optimierungen besonders gut (mindestens ein Testfall pro Optimierungsregel). Überlegen Sie sich, welche Optimierungen es wohl wirklich bringen (welche Fälle also tatsächlich vorkommen), und lassen Sie die anderen weg.

7.6.3 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/codea`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `codea` erzeugen, das von der Standardeingabe liest und den generierten Code auf die Standardausgabe ausgibt. Bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei einem Syntaxfehler Fehlerstatus 2, bei anderen Fehlern der Fehlerstatus 3. Im Fall eines Fehlers darf die Ausgabe beliebig sein.

[3] Tiefe eines Ausdrucks: Anzahl der Ableitungen von `Expr` zwischen einem Blatt des Syntaxbaums und dem nächsten `Statement`.

7.7 Codeerzeugung B

7.7.1 Termin

Abgabe spätestens am 31. Mai 2017, 14 Uhr.

7.7.2 Angabe

Erweitern Sie den Compiler aus dem vorigen Beispiel so, dass er folgende Untermenge der statisch korrekten Programme in AMD64-Assemblercode übersetzt: Alle Programme, in denen der Parser keinen *Funktionsaufruf* ableitet. Programme, die statisch korrekt sind, aber dieser Einschränkung nicht entsprechen, werden bei diesem Beispiel nicht als Testeingaben vorkommen.

Ein Teil der Sprache wurde schon erklärt, hier der für dieses Beispiel notwendige Zusatz:

Eine `goto`-Anweisung springt zum angegebenen Label.

Eine `if`-Anweisung wertet `Cond` nach der Kontrollflussmethode aus. Wenn die Bedingung zutrifft, springt sie zum angegebenen Label, sonst zum folgenden `Stat`.

Die Bedingung kann auch ein `and` von mehreren Bedingungen sein. In diesem Fall verursacht die erste nicht zutreffende Bedingung einen Sprung zum folgenden Statement (Kontrollflussmethode).

Die Bedingung kann auch ein `not` einer anderen Bedingung `Cterm` sein. In diesem Fall springt `Cterm` im Erfolgsfall zum folgenden Stat, und im Misserfolgsfall zum Label, die beiden Sprungziele werden also vertauscht (Kontrollflussmethode).

`and` und `not` können auch verschachtelt angewendet werden, dann muss die Kontrollflussmethode entsprechend allgemeiner angewendet werden, siehe Vorlesungsskriptum.

Eine *Variablendefinition* wertet die `Expr` aus und weist das Ergebnis der definierten Variable zu.

Eine *Zuweisung* schreibt den Wert der `Expr` in die durch `Lexpr` angegebene Variable bzw. das Arrayelement.

Eine `Term`-Anweisung wertet den Term aus und macht mit dem Ergebnis nichts (in diesem Beispiel gibt es keine Funktionsaufrufe, daher macht diese Anweisung hier gar nichts).

7.7.2.1 Erzeugter Code. Es gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel. Weiters sollte man sich überlegen, die Labels so im Assemblercode zu kodieren, dass sie nicht mit den Funktionsnamen kollidieren können, z.B. indem aus einem Label x in der Eingabesprache ein Label `L_`x auf der Assemblerebene wird.

7.7.3 Hinweis

Es bringt nichts, für `iburg` Bäume zu bauen, die mehr als eine einfache Anweisung umfassen bzw. ein `Cterm`: die Möglichkeit, durch die Baumgrammatik Knoten zusammenzufassen und so zu optimieren, kann nur auf der Ebene von Ausdrücken und einfachen Anweisungen genutzt werden, solange man in der Zwischendarstellung nahe beim abstrakten Syntaxbaum bleibt.

Auf höherer Ebene ist einfacher, für jede einfache Anweisung einen Baum zu bauen und dann in einem Traversal für jeden dieser Bäume den Labeler und den Reducer aufzurufen.

7.7.4 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/codeb`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `codeb` erzeugen, das von der Standardeingabe liest und den generierten Code auf die Standardausgabe ausgibt. Bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei einem Syntaxfehler Fehlerstatus 2, bei anderen Fehlern der Fehlerstatus 3. Im Fall eines Fehlers darf die Ausgabe beliebig sein.

7.8 Gesamtbeispiel

7.8.1 Termin

Abgabe spätestens am 14. Juni 2017, 14 Uhr. Es gibt nur einen Nachtermin.

7.8.2 Angabe

Erweitern Sie den Compiler aus dem vorigen Beispiel so, dass er alle statisch korrekten Programme in AMD64-Assemblercode übersetzt.

Ein Teil der Sprache wurde schon erklärt, hier der für dieses Beispiel notwendige Zusatz:

Der *Funktionsaufruf* wertet alle `Exprs` aus und ruft dann die Funktion `id` auf, mit den Ergebnissen der `Exprs` als Parameter. Der von der Funktion zurückgegebene Wert ist der Wert des Funktionsaufrufs.

7.8.2.1 Erzeugter Code. Der erzeugte Code ruft Funktionen entsprechend den Aufrufkonventionen auf. Ansonsten gelten die gleichen Anforderungen und Einschränkungen wie im vorigen Beispiel, wobei ein Funktionsaufruf mit n Parametern bei der Berechnung der Tiefe mit dem Wert $\max(0, n-1)$ (zuzüglich der maximalen Tiefe der Berechnungen der Parameter) eingeht.

Wichtigstes Kriterium ist wie immer die Korrektheit, für gute Codeerzeugung gibt es aber wieder Sonderpunkte. Wir empfehlen, nur Optimierungen durchzuführen, die mit den verwendeten Werkzeugen einfach möglich sind. Bei diesem Beispiel kommt es mehr auf gute Registerbelegung an als auf die Optimierung von Ausdrücken.

7.8.3 Hinweise

Bei der Registerbelegung gibt es sowohl ein großes Optimierungspotential als auch ein großes Fehlerpotential, besonders im Zusammenhang mit (verschachtelten) Funktionsaufrufen.

Eine einfache Strategie bezüglich der Parameter der aktuellen Funktion ist, sie nicht in den Argumentregistern zu lassen, sondern sie z.B. auf den Stack zu kopieren, damit man beim Berechnen der Parameter einer anderen Funktion problemlos auf sie zugreifen kann. Diese Strategie mag zwar nicht zum optimalen Code führen, aber eine gute Regel beim Programmieren lautet: "First make it work, then make it fast".

7.8.4 Abgabe

Zum angegebenen Termin stehen die maßgeblichen Dateien im Verzeichnis `~/abgabe/gesamt`. Mittels `make clean` soll man alle von Werkzeugen erzeugten Dateien löschen können und mittels `make` ein Programm namens `gesamt` erzeugen, das von der Standardeingabe liest und auf die Standardausgabe ausgibt. Bei einem lexikalischen Fehler soll der Fehlerstatus 1 erzeugt werden, bei einem Syntaxfehler Fehlerstatus 2, bei anderen Fehlern der Fehlerstatus 3. Im Fall eines Fehlers kann die Ausgabe beliebig sein. Der ausgegebene Code muss vom Assembler verarbeitet werden können.

GNU Emacs Reference Card

(for version 20)

Starting Emacs

To enter GNU Emacs 20, just type its name: `emacs`

To read in a file to edit, see Files, below.

Leaving Emacs

suspend Emacs (or iconify it under X)	`C-z`
exit Emacs permanently	`C-x C-c`

Files

read a file into Emacs	`C-x C-f`
save a file back to disk	`C-x C-s`
save all files	`C-x s`
insert contents of another file into this buffer	`C-x i`
replace this file with the file you really want	`C-x C-v`
write buffer to a specified file	`C-x C-w`
version control checkin/checkout	`C-x C-q`

Getting Help

The help system is simple. Type `C-h` (or `F1`) and follow the directions. If you are a first-time user, type `C-h t` for a tutorial.

remove help window	`C-x 1`
scroll help window	`C-M-v`
apropos: show commands matching a string	`C-h a`
show the function a key runs	`C-h c`
describe a function	`C-h f`
get mode-specific information	`C-h m`

Error Recovery

abort partially typed or executing command	`C-g`
recover a file lost by a system crash	`M-x recover-file`
undo an unwanted change	`C-x u` or `C-_`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`
reverse regular expression search	`C-M-r`
select previous search string	`M-p`
select next later search string	`M-n`
exit incremental search	`RET`
undo effect of last character	`DEL`
abort current search	`C-g`

Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not done.

Moti

entity to move over	backward	forward
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-{`	`M-}`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`
scroll current line to center of screen	`C-u C-l`

Killing and Deleting

entity to kill	backward	forward
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill region	`C-w`
copy region to kill ring	`M-w`
kill through next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* words away	`M-@`
mark paragraph	`M-h`
mark page	`C-x C-p`
mark sexp	`C-M-@`
mark function	`C-M-h`
mark entire buffer	`C-x h`

Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	`SPC`
replace this one, don't move	`,`
skip to next without replacing	`DEL`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	`RET`
enter recursive edit (`C-M-c` to exit)	`C-r`

ultiple Windows

When two commands are shown, the second is for "other frame."

delete all other windows	`C-x 1`	
split window, above and below	`C-x 2`	`C-x 5 2`
delete this window	`C-x 0`	`C-x 5 0`
split window, side by side	`C-x 3`	
scroll other window	`C-M-v`	
switch cursor to another window	`C-x o`	`C-x 5 o`
select buffer in other window	`C-x 4 b`	`C-x 5 b`
display buffer in other window	`C-x 4 C-o`	`C-x 5 C-o`
find file in other window	`C-x 4 f`	`C-x 5 f`
find file read-only in other window	`C-x 4 r`	`C-x 5 r`
run Dired in other window	`C-x 4 d`	`C-x 5 d`
find tag in other window	`C-x 4 .`	`C-x 5 .`
grow window taller	`C-x ^`	
shrink window narrower	`C-x {`	
grow window wider	`C-x }`	

Formatting

indent current line (mode-dependent)	`TAB`
indent region (mode-dependent)	`C-M-\`
indent sexp (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
join line with previous (with arg, next)	`M-^`
delete all white space around point	`M-\`
put exactly one space at point	`M-SPC`
fill paragraph	`M-q`
set fill column	`C-x f`
set prefix each line starts with	`C-x .`
set face	`M-g`

Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
fetch previous minibuffer input	`M-p`
fetch later minibuffer input or default	`M-n`
regexp search backward through history	`M-r`
regexp search forward through history	`M-s`
abort command	`C-g`

Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. Type `F10` to activate the menu bar using the minibuffer.

GNU Emacs Reference Card

Buffers

select another buffer	C-x b
list all buffers	C-x C-b
kill a buffer	C-x k

Transposing

transpose characters	C-t
transpose words	M-t
transpose lines	C-x C-t
transpose sexps	C-M-t

Spelling Check

check spelling of current word	M-$
check spelling of all words in region	M-x ispell-region
check spelling of entire buffer	M-x ispell-buffer

Tags

find a tag (a definition)	M-.
find next occurrence of tag	C-u M-.
specify a new tags file	M-x visit-tags-table
regexp search on all files in tags table	M-x tags-search
run query-replace on all the files	M-x tags-query-replace
continue last tags search or query-replace	M-,

Shells

execute a shell command	M-!
run a shell command on the region	M-|
filter region through a shell command	C-u M-|
start a shell in window *shell*	M-x shell

Rectangles

copy rectangle to register	C-x r r
kill rectangle	C-x r k
yank rectangle	C-x r y
open rectangle, shifting text right	C-x r o
blank out rectangle	C-x r c
prefix each line with a string	C-x r t

Abbrevs

add global abbrev	C-x a g
add mode-local abbrev	C-x a l
add global expansion for this abbrev	C-x a i g
add mode-local expansion for this abbrev	C-x a i l
explicitly expand abbrev	C-x a e
expand previous word dynamically	M-/

Regular Expressions

any single character except a newline	. (dot)
zero or more repeats	*
one or more repeats	+
zero or one repeat	?
quote regular expression special character *c*	\c
alternative ("or")	|
grouping	\(... \)
same text as *n*th group	\n
at word break	\b
not at word break	\B

entity	match start	match end
line	^	$
word	\<	\>
buffer	\`	\'

class of characters	match these	match others
explicit set	[...]	[^ ...]
word-syntax character	\w	\W
character with syntax *c*	\sc	\Sc

International Character Sets

specify principal language	M-x set-language-environment
show all input methods	M-x list-input-methods
enable or disable input method	C-\
set coding system for next command	C-x RET c
show all coding systems	M-x list-coding-systems
choose preferred coding system	M-x prefer-coding-system

Info

enter the Info documentation reader	C-h i
find specified function or variable in Info	C-h C-i

Moving within a node:

scroll forward	SPC
scroll reverse	DEL
beginning of node	. (dot)

Moving between nodes:

next node	n
previous node	p
move up	u
select menu item by name	m
select *n*th menu item by number (1–9)	*n*
follow cross reference (return with l)	f
return to last node you saw	l
return to directory node	d
go to any node by name	g

Other:

run Info tutorial	h
quit Info	q
search nodes for regexp	M-s

Registers

save region in register	C-x r s
insert register contents into buffer	C-x r i
save value of point in register	C-x r SPC
jump to point saved in register	C-x r j

Keyboard Macros

start defining a keyboard macro	C-x (
end keyboard macro definition	C-x)
execute last-defined keyboard macro	C-x e
append to last keyboard macro	C-u C-x (
name last keyboard macro	M-x name-last-kbd-macro
insert Lisp definition in buffer	M-x insert-kbd-macro

Commands Dealing with Emacs Lisp

eval sexp before point	C-x C-e
eval current defun	C-M-x
eval region	M-x eval-region
read and eval minibuffer	M-:
load from standard system directory	M-x load-library

Simple Customization

customize variables and faces	M-x customize

Making global key bindings in Emacs Lisp (examples):

```
(global-set-key "\C-cg" 'goto-line)
(global-set-key "\M-#" 'query-replace-regexp)
```

Writing Commands

```
(defun command-name (args)
  "documentation" (interactive "template")
  body)
```

An example:

```
(defun this-line-to-top-of-window (line)
  "Reposition line point is on to top of window.
With ARG, put point on line ARG."
  (interactive "P")
  (recenter (if (null line)
                0
              (prefix-numeric-value line))))
```

The `interactive` spec says how to read arguments interactively. Type `C-h f interactive` for more details.

AMD64-Assembler-Handbuch

Fabian Schmied, Institut für Computersprachen

Basierend auf dem Alpha-Assembler-Handbuch von Andreas Krall und dem *AMD64 Architecture Programmer's Manual*

1 Allgemeines

Die AMD64-Architektur ist eine 64-Bit-Erweiterung der weit verbreiteten Intel-x86-Architektur. Sie erweitert diese um 64-Bit-Adressierung und verbesserte Register-Ressourcen, vor allem sind alle General-Purpose-Register und der Instruktionszeiger 64 Bit breit. Insgesamt werden 16 General-Purpose-Register, 16 128-bit-Medienregister, und 8 kombinierte 80-Bit-Gleitkomma/64-bit Medienregister geboten. Die meisten Befehle existieren in mehreren Versionen, so dass sowohl mit allen 64 Bit eines Registers, als auch mit den unteren 8, 16 oder 32 Bit gearbeitet werden kann. Neben dem 64-Bit-Modus unterstützt die Architektur auch Kompatibilitäts- und Legacy-Modi, wodurch die Architektur kompatibel zur Intel-x86-Architektur ist und 32-Bit-Programme ohne Rekompilierung ausführen kann.

Für die Übersetzerbauübungen werden weder Gleitkommaprogramme, noch Programme im Kompatibilitäts- oder Legacy-Modus verwendet. Diese Anleitung geht daher nur auf den Ganzzahlanteil des 64-Bit-Modus der AMD64-Architektur ein. Weiterführende Informationen finden Sie im im Handbuch des GNU-Assemblers[1] und im Internet (z.B. das *Linux Assembly HOWTO*[2], `http://www.x86-64.org` und das *AMD64 Architecture Programmer's Manual*[3]).

2 Assemblersyntax

Diese Anleitung benutzt die Syntax des GNU-Assemblers GAS (aufrufbar mit `as`, Dokumentation über `info as`). GAS erlaubt Kommentare in C-Syntax, zusätzlich kann ein Zeilenkommentar mit # beginnen. Namen bestehen aus Buchstaben, Ziffern, '.', '$' und '_'. Das erste Zeichen eines Namens darf keine Ziffer sein, Zahlen und Zeichenketten entsprechen der C-Konvention.

Jede Zeile der Eingabedatei enthält einen oder mehrere durch ';' getrennte *Statements*, das letzte Statement einer Datei muss durch einen Zeilenumbruch abgeschlossen sein. Statements können auch leer sein, in diesem Fall werden sie ignoriert. Jedes Statement beginnt mit optionalen Labels (ein von ':' gefolgter Name, dazwischen darf kein Whitespace-Zeichen stehen), dann kommt der Befehl. Beginnt der Befehl mit '.', so handelt es sich um eine Assembler-Direktive (siehe Abschnitt 8), beginnt der Befehl mit einem Buchstaben, so ist es eine *Instruktion*, die in einen Maschinenbefehl übersetzt wird (siehe Abschnitt 6). Je nach Befehl folgen etwaige Operanden bzw. Argumente.

[1] `info as`

[2] `http://www.ibiblio.org/pub/Linux/docs/HOWTO/Assembly-HOWTO`

[3] `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf`

Die Operanden eines Befehls sind Register, Adressen, Offsets und Werte. Ein Offset ist eine ganze Zahl (siehe unten, *Offsetting*), Werte und Adressen werden durch Ausdrücke aus Zahlen und Symbolen mit Operatoren wie in C dargestellt. Folgende Operatoren werden unterstützt:

Unäre Operatoren: - (Zweierkomplementnegation), ~(bitweises NOT)

Binäre Operatoren: (Operatoren mit gleichem Vorrang werden im Code von links nach rechts ausgewertet)

Höchster Vorrang: * (Multiplikation), / (Division), % (Restbildung), << (Linksschieben), >> (Rechtsschieben)

Mittlerer Vorrang: | (bitweises inklusives OR), & (bitweises AND), ^ (bitweises exclusives OR), ! (bitweises OR NOT)

Niedriger Vorrang: + (Addition), - (Subtraktion), == (Gleichheitsprüfung), <> (Ungleichheitsprüfung), <, <=, >, >= (übrige Vergleichsoperatoren für vorzeichenbehaftete Werte – die Vergleichsoperatoren liefern -1 für erfüllte und 0 für nicht erfüllte Vergleiche)

Niedrigster Vorrang: && (Logisches AND), || (Logisches OR, hat einen leicht niedrigeren Vorrang als &&); diese Operatoren liefern 1 für einen erfüllten und 0 für einen nicht erfüllten Ausdruck

Bei der Befehlssyntax unterstützt GAS zwei Varianten: die AT&T-Syntax (default) und die Intel-Syntax. Obwohl die meisten Spezifikationen im Umfeld der Intel-Architektur (z.B. das AMD64 Architecture Programmer's Manual) die Intel-Syntax verwenden, benutzen wir in diesem Dokument die unter Linux gebräuchlichere AT&T-Syntax, die z.B. auch von *GCC* generiert wird und sich von der Intel-Syntax in einigen Punkten unterscheidet. In GAS-Programmen kann die Intel-Syntax mit der Direktive `.intel_syntax` aktiviert werden.

Operandenpräfixe: Register werden in der AT&T-Syntax mit % gekennzeichnet, Immediate-Werte mit $. Beispiele: `%eax`, `$4`. Wenn das $-Präfix weggelassen wird, wird der Wert stattdessen als Speicheradresse interpretiert.

Operandenreihenfolge: Zuerst kommt der Quelloperand, dann das Ziel. Z.B. bedeutet `mov %eax, %ebx` eine MOV-Operation von `eax` nach `ebx` (in der Intel-Syntax ist es genau umgekehrt!). Im Zweifelsfall findet sich eine gute Zusammenfassung des Intel-Befehlssatz in AT&T-Syntax unter `http://docs.sun.com/app/docs/doc/802-1948`.

Befehlssuffixe: GAS erkennt die Operandengröße am Suffix des benutzten Befehls, hier muss b (8-Bit-Byte), w (16-Bit-Word), l (32-Bit-Long) oder q (64-Bit-Quadword) stehen. Strenggenommen wäre die Syntax für eine MOV-Operation von `eax` nach `ebx` also `movl %eax, %ebx)`. Wenn GAS die Operandengröße allerdings auf Grund der Registeroperanden erkennen kann, kann das Suffix weggelassen werden.

Offsetting: Um eine Speicherstelle zu indizieren oder indirekt auf einen Wert zuzugreifen wird das Indexregister oder die Speicheradresse in Klammern hinter dem Offset angegeben. `movl 17(%ebp), %eax` kopiert also einen Wert von einer Speicherstelle nach `eax`. Die Quellspeicherstelle befindet sich dabei 17 Bytes hinter der Adresse, die in `ebp` steht. Siehe dazu auch die Erklärung von ModR/M-Adressierung in Abschnitt 4.

64 Bit	32 Bit	16 Bit	obere 8 Bit des 16-Bit-Teils	untere 8 Bit
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si	–	sil
rdi	edi	di	–	dil
rbp	ebp	bp	–	bpl
rsp	esp	sp	–	spl
r8	r8d	r8w	–	r8b
r9	r9d	r9w	–	r9b
...	...	...	–	...
r15	r15d	r15w	–	r15b

Abbildung 1: General-Purpose-Register

Relative Jumps: Bei Sprungbefehlen wird ohne besondere Kennzeichnung der Operand als Zieladresse des Sprunges angenommen. Soll ein indirekter Sprung durchgeführt werden, muss die Adresse mit einem '*' als Präfix versehen werden.

Beispielcode finden Sie später in dieser Anleitung im Abschnitt 5.3 auf Seite 9.

3 Register

3.1 General-Purpose-Register

Abbildung 1 enthält eine Übersicht über die General-Purpose-Register der AMD64-Architektur. Alle Register sind über verschiedene Namen in verschiedenen Größen ansprechbar. Die Register `ah`, `bh`, `ch` und `dh` enthalten die oberen 8 Bit des entsprechenden 16-Bit-Registers `ax`, `bx`, `cx` und `dx`.

Wird ein 8- oder 16-Bit-Teil eines Registers überschrieben, so werden die übrigen Bits des Registers nicht verändert. Wird jedoch der 32-Bit-Teil des Registers manipuliert, werden automatisch die restlichen 32 Bit des Registers auf 0 gesetzt.

3.2 Spezialregister

`Rip` ist der Instruktionszeiger, der die Adresse der nächsten auszuführenden Instruktion enthält, das Register `rsp` wird üblicherweise als Stack-Pointer und `rbp` als Frame-Pointer (Zeiger in den Activation Record) benutzt. Einige der anderen Register haben ebenfalls Bedeutungen im Rahmen der unter Linux benutzten Aufrufkonventionen; diese werden in Abschnitt 5.1 beschrieben.

Das `rflags`-Register enthält in den untersten 16 Bit Operations-Flags wie *Carry*, *Parity*, *Zero* oder *Sign*, in den nächsten 16 Bit System-Flags, die nur von Systemsoftware aus zugreifbar sind. Die oberen 32 Bit des Registers sind reserviert und liefern beim Lesen immer 0. Abbildung 2 enthält eine Übersicht über die verfügbaren Operations-Flags und ihre Bedeutung.

Flag	Name	Bedeutung
CF	Carry	Die letzte Integer-Addition, -Subtraktion, oder Compare-Operation ergab einen Übertrag (*Carry* oder *Borrow*). Inkrement- und Dekrement-Befehle beeinflussen das Flag nicht, Shift- und Rotate-Befehle schieben hinausgeschobene Bits in das Carry-Flag, logische Operationen löschen das Flag.
PF	Parity	Das letzte Resultat bestimmter Operationen hatte eine gerade Anzahl von gesetzten Bits.
AF	Auxiliary Carry	Die letzte Binary-Coded-Decimal-Operation ergab ein Carry in Bit 3. Auf BCD-Operationen wird in dieser Anleitung nicht eingegangen.
ZF	Zero	Das letzte Resultat einer arithmetischen Operation war 0. Dieses Flag wird auch von den Vergleichs-Instruktionen gesetzt und kann benutzt werden, um zwei Werte auf Gleichheit zu prüfen.
SF	Sign	Das letzte Resultat einer arithmetischen Operation war negativ. (Das SF ist auf Grund der benutzten Zweierkomplementdarstellung von Ganzzahlen immer gleich dem höchstwertigen Bit des Resultats.)
DF	Direction	Bestimmt die Verarbeitungsrichtung für String-Befehle. Auf String-Befehle wird in dieser Anleitung nicht eingegangen.
OF	Overflow	Das Resultat der letzten Signed-Integer-Operation war zu groß, um in den Datentyp zu passen. Dieses Flag ist nach einer DIV-Instruktion und nach Shifts um mehr als ein Bit undefiniert. Logische Operationen löschen das Flag.

Abbildung 2: Operations-Flags

Neben diesen beiden gibt es noch eine Reihe weiterer Spezialregister, die in dieser LVA aber keine Rolle spielen.

3.3 Medien-Register

Als Erweiterung zur klassischen Intel-x86-Architektur gibt es in der AMD64-Architektur sogenannte *Streaming SIMD*[4] *Extensions* (SSE, SSE2, und weitere, die von den aktuellen Übungsmaschinen nicht unterstützt werden). Diese enthalten unter anderem Medien-Befehle, die vor allem für Anwendungen aus dem Multimedia- und Wissenschaftsbereich gedacht sind, bei denen viele Einzelwerte aus großen Datenmengen unabhängig voneinander verarbeitet werden müssen. Die Operanden von Medien-Befehle sind 128 Bit große Vektoren, die mehrere zu bearbeitende Werte enthalten, wobei die Elemente eines Vektor-Operanden Ganzzahlen (von 8-Bit-Bytes bis zu 64-Bit-Quadwords) oder Gleitkommawerte sein können. Wir werden uns in dieser Anleitung auf die Ganzzahlverarbeitung konzentrieren.

Die meisten der arithmetischen 128-Bit-Instruktionen arbeiten mit zwei Quellregistern, die je einen Vektor aus Operanden enthalten, wobei das zweite Quellregister durch einen Vektor von Ergebniswerten überschrieben wird. Es stehen 16 derartige 128-Bit-Register zur Verfügung, die

[4]Single Instruction Multiple Data

Ausdruck	DISP	BASE	INDEX	SCALE	Bedeutung
`-4(%ebp)`	-4	`ebp`	Default	Default	`ebp-4`
`foo(,%eax,4)`	foo	Default	`eax`	4	foo + 4*`eax`
`foo(,1)`	foo	Default	Default	1	foo (siehe Text)
`-4(%ebp, %eax, 4)`	-4	`ebp`	`eax`	4	`ebp` + 4*`eax` - 4

Abbildung 3: ModR/M-Adressbeispiele

`xmm0` bis `xmm15` genannt werden. Zusätzlich gibt es ein Kontroll- und Statusregister `mxcsr`, das Flags wie *Invalid Operation Exception*, *Zero-Divide Exception* oder *Overflow Exception* enthält.

4 Speichermodell und Adressierung

Die AMD64-Architektur benutzt im 64-Bit-Modus ein flaches Segmentierungsmodell für den virtuellen Speicher. Der gesamte 64-Bit-Speicherraum wird dabei als ein einziger, flacher Adressraum betrachtet. Befehle und Daten werden darin im Little-Endian-Format[5] abgelegt und über verschiedene Adressierungsmodi angesprochen:

Bei *absoluter Adressierung* werden die Adressen direkt als Werte angegeben. Beispiel: `movl 0x1234, %eax`

Bei *RIP-relativer Adressierung* werden Adressen als Offsets zum Instruktionszeiger (`rip`) angegeben. Dies ist sinnvoll für relative Sprünge, aber auch, um positionsunabhängigen Code zu erzeugen: Wird auf Symbole RIP-relativ zugegriffen, kann der Linker den Code in einen beliebigen Speicherbereich verschieben, ohne die Adressen anpassen zu müssen. Beispiel: `movl xvar(%rip), %eax`.

ModR/M-Adressierung dient dem indirekten Zugriff auf Speicherbereiche, deren Adressen zur Laufzeit berechnet werden. Dabei wird die effektive Adresse aus einer *Basisadresse* und einem *Index*, die aus General-Purpose-Registern ausgelesen werden, sowie einem *Skalierungsfaktor* (1, 2, 4 oder 8) und einem *Displacement*, die direkt im Code angegeben sind, berechnet. Die Formel für die Adressberechnung lautet: $Base + Index * Scale + Displacement$, das Ergebnis wird wie eine absolute Adresse behandelt.

In der AT&T-Assembler-Syntax werden derartige Adressen in der Form `DISP(BASE, INDEX, SCALE)` angegeben, wobei alle vier Werte optional sind – der Default-Wert für SCALE ist 1, für alle anderen Werte ist er 0. Wenn innerhalb der Klammern nur ein Wert mit voranstehendem Komma angegeben wird, wird er als SCALE-Wert interpretiert. Abbildung 3 zeigt einige Beispiele von ModR/M-Adressen.

Daten auf dem *Stack* werden über den Stack-Pointer `rsp` adressiert, der von Befehlen wie POP, PUSH, CALL, RET und INT implizit verändert wird.

[5] Die Bytes eines Datenwerts werden von rechts nach links angeordnet, so dass das höchstwertige Byte „rechts", d.h. an der höchsten Adresse, das niederwertige Byte „links", d.h. an der niedrigsten Adresse steht. Die hexadezimalen 16-Bit-Zahl 0xABCD würde im Speicher also als Bytefolge CD AB abgelegt.

Register	Verwendungszweck	Sicherung
`rax`	Temporäres Register, erstes Rückgaberegister	Caller
`rbx`	Callee-gesichertes Register	Callee
`rcx`	Argumentregister für das vierte Ganzzahlargument	Caller
`rdx`	Argumentregister für das dritte Ganzzahlargument, zweites Rückgaberegister	Caller
`rsp`	Stack-Pointer	Callee (implizit)
`rbp`	Callee-gesichertes Register, wird als Frame-Pointer benutzt	Callee
`rsi`	Argumentregister für das zweite Ganzzahlargument	Caller
`rdi`	Argumentregister für das erste Argument	Caller
`r8`	Argumentregister für das fünfte Argument	Caller
`r9`	Argumentregister für das sechste Argument	Caller
`r10`	Temporäres Register, wird benutzt, um den Static-Chain-Pointer einer Funktion zu übergeben (in dieser Anleitung nicht weiter behandelt)	Caller
`r11`	Temporäres Register	Caller
`r12-r15`	Callee-gesicherte Register	Callee
`xmm0-xmm1`	Argument- und Rückgaberegister für SSE-Werte	Caller
`xmm2-xmm7`	Argumentregister für SSE-Werte	Caller
`xmm8-xmm15`	Temporäre Register	Caller

Abbildung 4: AMD64-ABI Aufrufkonventionen

5 Aufrufkonventionen

Um sicherzustellen, dass getrennt kompilierte Programmteile problemlos verlinkt werden können, gibt es für die AMD64-Architektur eine *System V Application Binary Interface*-Spezifikation[6], die festlegt, welche Konventionen beim Aufruf von Funktionen eingehalten werden sollten. Diese beinhaltet Einschränkungen für die Register, die in den Abschnitten 3.1 und 3.3 beschrieben wurden, und Konventionen für den Stack.

5.1 Register

Die Register `rbp`, `rbx` und `r12` bis `r15` gehören dem aufrufenden Code (*Caller*). Wenn die aufgerufene Funktion (*Callee*) die Register verändert, dann muss sie diese zuvor auf dem Stack sichern (z.B. mit dem PUSH-Befehl) und ihre Inhalte vor der Rückkehr zum Caller wiederherstellen (POP), man bezeichnet sie daher als *callee-saved Registers*. Alle anderen Register gehören dem Callee; der Caller muss also selbst für die Sicherung sorgen, wenn er ihren Inhalt über einen Aufruf hinweg benötigt (*caller-saved Registers*). Speziell gesichert wird das Register `rsp`, das üblicherweise beim Funktionsaufruf bzw. im Prolog einer Funktion verändert und nach `rbp` kopiert und im Epilog bzw. beim Rücksprung wiederhergestellt wird (siehe auch Abschnitt 5.2).

Beim Aufruf einer Funktion werden ganzzahlige Argumente der Reihe nach in `rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9` übergeben. SSE-Werte werden über die Register `xmm0` bis `xmm7` übergeben.

[6] `http://www.x86-64.org/documentation/abi.pdf`

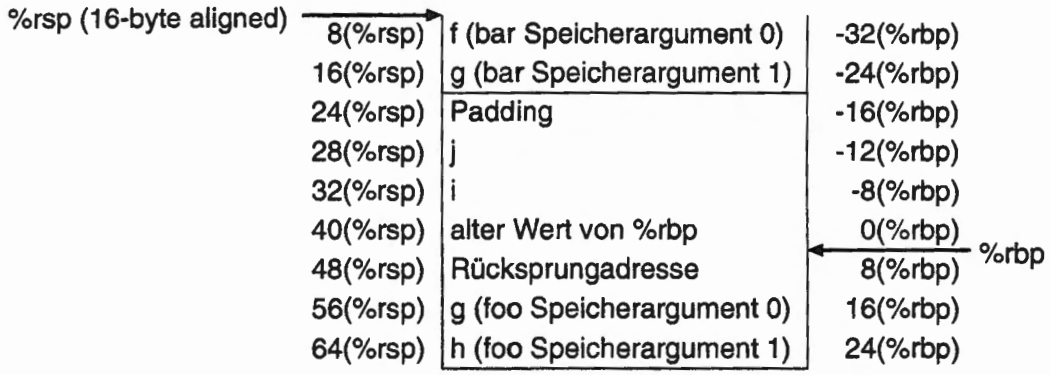


Abbildung 5: Der Stack mit einem Activation Record

Wenn mehr Argumente übergeben werden sollen als Register zur Verfügung stehen, werden die Argumente von rechts nach links (d.h. letztes Argument an die höchste Adresse[7]) auf den Stack geschrieben.

Wenn eine Funktion ganzzahlige Ergebniswerte liefert, werden diese über die Register `rax` und `rdx` an den aufrufenden Code zurückgegeben. SSE-Ergebnisse werden in die Register `xmm0` und `xmm1` geschrieben.

Abbildung 4 fasst die üblichen Verwendungen der Register zusammen.

5.2 Stack

Unmittelbar vor dem Aufruf einer Funktion C durch die Funktion B (die ihrerseits von A aufgerufen wurde) enthält der Stack normalerweise folgende Daten in einem Activation Record:

1. Argumente für B (sofern diese nicht per Register übergeben werden). Sie wurden von A auf den Stack gelegt.
2. Die Rücksprungadresse. Sie wird automatisch von der CALL-Instruktion auf den Stack gelegt.
3. Ein Feld von Activation-Record-Zeigern, die es ermöglichen, auf lokale Variablen von umgebenden Funktionen zuzugreifen (falls die aufgerufene Funktion eine statisch geschachtelte Funktion ist, darauf wird allerdings in dieser Anleitung nicht näher eingegangen).
4. Lokale Variablen der Funktion. Sie werden von der Funktion selbst initialisiert.
5. Eventuell Padding (ungenutzer Platz), das dafür sorgt, dass der Stack Pointer auf 16 Bytes ausgerichtet ist, sobald die Argumente für C auf den Stack gelegt wurden. Da das Padding vom Platz für die Argumente abhängt, kann es für jeden Aufruf anders sein.
6. Argumente für C, die im Speicher übergeben werden. Sie werden von B auf den Stack gelegt, aber man zählt sie schon zum Activation Record von C.

[7] Der Stack wächst von oben (hohe Adressen) nach unten (niedrige Adressen), die Spitze des Stacks befindet sich daher an der niedrigsten Adresse. Wenn ein Argument im Stack an einer höheren Adresse steht als ein anderes, befindet es sich also „tiefer“ im Stack.

Der Stack ist an dieser Stelle auf eine 16-Byte-Grenze ausgerichtet (aligned). Abbildung 5 zeigt den Stack der Funktion

```
int foo(char a, int b, long c, int d, int e, int f, int g, char h)
{
  char i=a+h;
  int j=b+g;
  bar(i,j,b,c,d,e,f,g);
  return i+j;
}
```

unmittelbar vor dem Aufruf von bar (also nachdem die Argumente bereitgelegt wurden), wobei `foo` einen Frame Pointer (siehe unten) verwendet.

Das `rbp`-Register wird häufig als Frame-Pointer benutzt: Es zeigt als Basisadresse auf jenen Bereich auf dem Stack, in dem der aktuelle Code seine lokalen Variablen ablegt (Activation Record). Über positive Offsets zum Frame-Pointer gelangt man an die Argumente, über negative Offsets an die lokalen Variablen. Üblicherweise enthält jede Funktion zur Initialisierung des Frame-Pointers und gleichzeitigen Sicherung von `rsp` einen Prolog, der wie folgt aussieht:

```
push %rbp       # rbp sichern
mov %rsp, %rbp  # neuen rbp setzen und rsp sichern
sub ..., %rsp   # Platz für lokale Variablen am Stack reservieren
```

(Manchmal wird stattdessen auch der ENTER-Befehl benutzt, der eine äquivalente Funktionalität zu diesem Codeblock bietet.)

Am Ende einer Funktion steht normalerweise ein Epilog, der die vom Prolog vorgenommenen Änderungen rückgängig macht und die Kontrolle an die aufrufende Funktion zurückgibt. Der LEAVE-Befehl wird benutzt, um den Stack aufzuräumen (d.h. `rsp` wieder auf den Wert des Frame-Pointers zu setzen und `rbp` vom Stack zu holen), mit RET wird dann zur Rücksprungadresse gesprungen (und die Adresse vom Stack entfernt).

```
leave           # rsp auf rbp setzen, rbp wiederherstellen
ret             # Rücksprung
```

(LEAVE ist äquivalent zu `mov %rbp, %rsp` und `pop %rbp`.)

Variante ohne Frame-Pointer Um Instruktionen zu sparen kann der Stack-Pointer auch direkt benutzt werden, wenn auf den Activation Record zugegriffen wird. Dabei wird `rbp` als normales (Callee-gesichertes) Register benutzt, und die Funktion muss manuell sicherstellen, dass `rsp` beim Rücksprung aus der Funktion wieder den ursprünglichen Wert enthält.

5.3 Beispielprogramm

Das folgende C-Beispielprogramm zeigt die Anwendung der Aufrufkonventionen:

```
long xvar;
extern long callee(long,long,long);

long caller() {
  long i = xvar;
  return i + callee(-1, -2, -3);
}
```

Hier liest eine Funktion *caller* einen globalen Wert *xvar* in eine lokale Variable *i* ein, ruft eine zweite Funktion *callee* mit drei Argumenten auf, addiert den Wert von *i* zum Ergebnis von *callee* und retourniert die Summe als ihren eigenen Rückgabewert.

Hier ein Auszug aus dem mittels `gcc -S` generierten Assembler-Code mit Kommentaren:

```
        .text                       # Programmbereich aktivieren
.globl caller                       # caller als externes Symbol definieren
        .type   caller, @function   # caller als Funktion deklarieren
caller:                             # Aufrufstelle für caller

# Funktions-Prolog
.LFB3:
        pushq   %rbp                # Frame Pointer auf dem Stack sichern
.LCFI0:
        movq    %rsp, %rbp          # neuen Frame Pointer setzen
.LCFI1:
        subq    $16, %rsp           # Platz für lokale Variablen reservieren

# Eigentliche Funktion
.LCFI2:
        movq    xvar(%rip), %rax    # Inhalt von xvar rip-relativ in rax
                                    # kopieren
        movq    %rax, -8(%rbp)      # rax als lokale Variable i speichern
        movq    $-3, %rdx           # drittes Argument (3) in edx laden
        movq    $-2, %rsi           # zweites Argument (2) in esi laden
        movq    $-1, %rdi           # erstes Argument (1) in edi laden
        call    callee              # callee aufrufen
        addq    -8(%rbp), %rax      # Inhalt von lokaler Variablen i zu rax
                                    # (Rückgabewert von callee) dazuzählen,
                                    # Resultat in rax speichern

# Funktions-Epilog
        leave                       # Activation Record der Funktion entfernen
        ret                         # Rückkehr von call
```

6 Befehlssatz

In diesem Abschnitt werden einige Befehle aus dem Befehlssatz der AMD64-Architektur vorgestellt. Der GNU-Assembler benutzt in den meisten Fällen dieselben Mnemonics für die Befehle

wie die AMD64-Architekturspezifikation. Bitte beachten Sie aber, dass die Befehle – wie in Abschnitt 2 beschrieben – eventuell durch Suffixe ergänzt werden müssen, um die Größe der Operanden zu definieren, sofern der Assembler sie nicht erkennen kann.

Alle Befehle der AMD64-Architektur finden Sie in den Architekturhandbüchern der Hersteller[8]).

6.1 Datentransferoperationen

Diese Befehle kopieren Daten zwischen Registern und dem Arbeitsspeicher.

6.1.1 Move

Move-Operationen kopieren Byte-, Word-, Long- und Quadword-Werte von Registern, Speicheradressen oder im Code angegebenen Werten zu Registern oder Speicheradressen.

```
mov source, dest
movsx source, dest    # Move mit Sign Extension
movzx source, dest    # Move mit Zero Extension
```

Source und *destination* müssen für MOV die gleiche Größe haben, MOVSX und MOVZX können kleinere Werte zu größeren Werten kopieren und füllen die restlichen Bits entweder mit Nullen auf (*Zero Extension*) oder stellen sicher, dass das Vorzeichen erhalten bleibt (*Sign Extension*). Im Gegensatz zu anderen Operationen benötigen MOVSX und MOVZX *zwei* Größenangaben, wenn der Assembler die Operandengrößen nicht selbst eruieren kann. In der Syntax lässt man dann das X weg und hängt zwei Suffixes an: das erste gibt die Größe des Quelloperanden, das zweite die des Zieloperanden an. (MOVSBW bedeutet beispielsweise eine Move-Operation mit Sign-Extension von einem 8-Bit-Operanden zu einem 16-Bit-Ziel.)

Source und *destination* können bei MOV-Operationen (was für die meisten Befehle gilt) nicht beide gleichzeitig Speicheradressen sein.

Anstelle von `mov $0, reg` wird häufig die Anweisung `xor reg, reg` benutzt, da dies in manchen Fällen effizienter sein kann.

6.1.2 Conditional Move

Conditional-Move-Operationen sind äquivalent zu normalen Move-Operationen, werden aber nur ausgeführt, wenn ein bestimmtes Bit des `rflags`-Registers gesetzt ist. In vielen Fällen ist diese Art von Move-Operation effizienter als eine äquivalente Formulierung des Programms mit Hilfe von Jumps.

```
cmovCC source, dest
```

Die entsprechenden Flags werden z.B. über Vergleichs- und Test-Instruktionen gesetzt (siehe Abschnitt 6.5). Abbildung 6 listet die möglichen Flag-Kürzel für `CC` auf.

[8] `http://www.complang.tuwien.ac.at/ubvl/#Unterlagen`

Flag-Kürzel	Flag	Bedeutung
o	OF = 1	Overflow
no	OF = 0	No overflow
b, c, nae	CF = 1	Below, carry, not above or equal (unsigned)
ae, nb, nc	CF = 0	Above or equal, not below, no carry (unsigned)
e, z	ZF = 1	Equal, zero
ne, nz	ZF = 0	Not equal, not zero
be, na	CF = 1 oder ZF = 1	Below or equal, not above (unsigned)
a, nbe	CF = 0 und ZF = 0	Above, not below or equal (unsigned)
s	SF = 1	Sign
ns	SF = 0	No sign
p, pe	PF = 1	Parity, parity even
np, po	PF = 0	No parity, parity odd
l, nge	SF <> OF	Less, not greater or equal (signed)
ge, nl	SF = OF	Greater or equal, not less (signed)
le, ng	ZF = 1 oder SF <> OF	Less or equal, not greater (signed)
g, nle	ZF = 0 und SF = OF	Greater, not less or equal (signed)

Abbildung 6: CMOV-Instruktionen

6.1.3 Stack-Operationen

Üblicherweise existiert für jede Funktion oder Prozedur ein Activation Record in einem dafür reservierten Speicherbereich – dem Stack. Die Funktion legt darin lokale Variablen ab, kann mit seiner Hilfe Registerinhalte sichern und Parameter für aufgerufene Funktionen übergeben (siehe Abschnitt 5.2). Die Stack-Befehle dienen der einfacheren Manipulation des Stacks, auf dessen aktuelle Spitze immer der Stack-Pointer `rsp` zeigt.

Die PUSH-Operation legt einen Byte-, Word-, Long- oder Quadword-Wert aus einem Register, von einer Speicheradresse oder aus dem Code auf den Stack, POP liest einen Wert vom Stack in ein Register oder einen Speicherbereich aus. ENTER erzeugt einen neuen Stack-Frame für eine Prozedur oder Funktion, LEAVE entfernt den Activation Record einer Prozedur, wie in Abschnitt 5.2 beschrieben.

```
push source
pop dest
enter size, depth
leave
```

PUSH und POP passen `rsp` entsprechend um 2, 4, oder 8 Bytes an, so dass der Stack-Pointer nach der Operation auf die neue Spitze des Stacks zeigt.

Der `depth`-Parameter (ein Wert aus dem Intervall 0-31) für ENTER gibt an, wieviele Activation-Record-Zeiger von der aufrufenden Prozedur kopiert werden sollen, um eine geschachtelte Funktion zu realisieren, der Wert `size` bestimmt, wieviele Bytes (z.B. für lokale Variablen der Prozedur) auf dem Stack allokiert werden. ENTER mit einer Schachtelungstiefe von Null entspricht der Codesequenz `push rbp; mov %rsp, %rbp; sub ..., %rsp` aus Abschnitt 5.2. LEAVE ist äquivalent zur dort angegebenen Codesequenz `mov %rbp, %rsp; pop %rbp`.

6.2 Adressladen

Die LEA-Instruktion berechnet und lädt die effektive Adresse einer Speicherstelle und legt diese in ein General-Purpose-Register.

```
lea source, destination
```

LEA ist ähnlich zum MOV-Befehl, der benutzt werden kann, um Daten von einer Speicheradresse in ein Register zu kopieren, aber anstatt den *Inhalt* des angegebenen Speicherbereichs zu laden, lädt LEA die *Adresse*.

Im einfachsten Fall kann LEA durch MOV ersetzt werden, z.B. ist der Befehl `lea (%ebx), %eax` gleichbedeutend mit `mov %ebx, %eax`. Mit LEA kann jedoch jeder beliebige Adressausdruck ausgewertet werden, z.B. `lea (%edi,%ebx,1), %eax`, was nicht durch ein MOV nachgebildet werden kann.

LEA kann auch in begrenztem Maß dazu eingesetzt werden, um Multiplikationen nachzubilden, indem ModR/M-Adressierung benutzt wird (z.B. multipliziert `lea (%ebx,%ebx,8), %eax` den Wert von `ebx` mit Neun und speichert das Ergebnis in `eax`).

6.3 Arithmetische Operationen

Arithmetische Befehle werden benutzt, um Grundrechenoperationen auf Ganzzahlen durchzuführen.

6.3.1 Addition und Subtraktion

ADD addiert zwei ganzzahlige Operanden und setzt die entsprechenden Bits des `rflags`-Registers (OF, SF, ZF, AF, CF, PF). Wenn man ein Wert zu einem Operanden mit höherer Bitbreite addiert, so wird der kleinere Wert zunächst mit Sign-Extension auf die größere Bitbreite erweitert. SUB subtrahiert zwei ganzzahlige Operanden. ADC und SBB sind äquivalent zu ADD und SUB, addieren bzw. subtrahieren aber zusätzlich Eins zum/vom Resultat, wenn das Carry-Flag gesetzt ist. NEG führt eine arithmetische Negation durch (d.h. es subtrahiert den Operanden von Null, dreht also das Vorzeichen um).

```
add source, source_dest
sub source, source_dest
adc source, source_dest   # Add mit Carry
sbb source, source_dest   # Sub mit Carry (Borrow)
neg source_dest
```

SUB und SBB subtrahieren den ersten vom zweiten Operanden und speichen das Ergebnis im zweiten Operanden (`source_dest = source_dest - source`).

6.3.2 Multiplikation und Division

MUL und DIV führen vorzeichenlose, IMUL und IDIV vorzeichenbehaftete Multiplikation und Division von Ganzzahlen durch. Bei einer Multiplikation kann die Bitbreite des Operationsergebnisses doppelt so groß sein wie die der Quelloperanden.

```
mul factor
imul factor
imul factor, multiplicand_product
imul factor, multiplicand, product
div divisor
idiv divisor
```

MUL erwartet einen Operanden (je nach Größe) in `al`, `ax`, `eax` oder `rax` und legt das Ergebnis in `ax`, `dx:ax`, `edx:eax` oder `rdx:rax` ab.

IMUL verhält sich in der einargumentigen Form wie MUL, kann jedoch auch zwei oder drei Argumente nehmen. (Siehe dazu auch *::Machine-Dependencies::i386-Dependent::i386-Notes* in der GAS-Dokumentation.)

DIV und IDIV erwarten den Dividenden in `ah:al`, `dx:ax`, `edx:eax` oder `rdx:rax`. Der Quotient wird in `al`, `ax`, `eax` oder `rax` gespeichert, der Rest steht in `ah` oder im entsprechenden dx-Register. Division ist die langsamste aller Ganzzahloperationen.

6.3.3 Inkrement und Dekrement

INC und DEC erhöhen bzw. verringern den Inhalt eines Registers oder einer Speicherstelle um 1.

```
inc source_dest
dec source_dest
```

INC und DEC verhalten sich genau wie ADD und SUB mit Eins als erstem Argument, verändern aber das Carry-Flag nicht.

6.4 Rotate und Shift

Diese Befehle führen zyklische Rotation oder nichtzyklische Schiebeoperationen um eine bestimmte Anzahl (Count) von Bits durch. Der Count kann ein 8-bit-Wert oder der Inhalt des `cl`-Registers sein. (Ein Rotate oder Verschieben um N Bits entspricht einem N-maligen Rotieren oder Verschieben um ein Bit, die Werte von CF und OF sind danach jedoch nicht definiert.)

```
rcl count, source_dest   # Rotate through Carry left
rcr count, source_dest   # Rotate through Carry right
rol count, source_dest   # Rotate left
ror count, source_dest   # Rotate right
```

```
sal count, source_dest   # Shift arithmetic left (signed)
sar count, source_dest   # Shift arithmetic right (signed)
shl count, source_dest   # Shift left (unsigned)
shr count, source_dest   # Shift right (unsigned)
shld count, source, source_dest   # Shift left double (unsigned)
shrd count, source, source_dest   # Shift right double (unsigned)
```

RCL und RCR rotieren den Operanden durch das Carry, d.h. ein hinausrotiertes Bit wird ins Carry geschrieben und der Wert des Carry-Flags wird am anderen Ende des Werts hineinrotiert. Auch ROL und ROR verändern das Carry-Flag auf das zuletzt hinausrotierten Bits, rotieren das Bit aber sofort wieder hinein, d.h. der vorherige Wert des Carry-Flags geht verloren. Das Overflow-Flag zeigt nach ROL- und ROR-Operationen um ein Bit an, ob sich durch die Rotation das Vorzeichenbit des Operanden verändert hat.

SHL und SAL können als effizienter Ersatz für eine Multiplikationsoperation mit Zwei (bzw. Potenzen davon), SHR und SAR statt einer Division durch Zwei benutzt werden. (Das Ergebnis von SAR unterscheidet sich bei negativem Operanden allerdings in der Rundung vom Ergebnis von IDIV.) SAR konserviert das Vorzeichen des Operanden bei der Schiebeoperation, d.h. es werden bei negativen Werten Einsen und bei positiven Werten Nullen von links hereingeschoben, SHR schiebt immer Nullen herein. Alle Shift-Operationen schieben das hinausgeschobene Bit ins Carry-Flag.

SHLD und SHRD führen Schiebeoperationen mit doppelter Länge durch, d.h. `count` Bits des `source`-Werts werden (statt Nullen) in den `source_dest`-Wert hineingeschoben.

6.5 Vergleichen und Testen

CMP subtrahiert analog zu SUB den Wert des ersten Operanden von dem des zweiten, überschreibt diesen jedoch nicht. Der einzige Effekt der Operation ist daher das Setzen der Flags (CF, SF, ZF, AF, CF, PF). TEST führt eine logische AND-Operation mit den beiden Operanden aus (analog zur AND-Instruktion), überschreibt jedoch ebenfalls keinen der beiden Operanden, sondern setzt nur die entsprechenden Flags (SF, ZF und PF; OF und CF werden gelöscht). TEST wird üblicherweise benutzt, um bestimmte oder alle Bits eines Operanden auf 0 zu überprüfen.

BT kopiert das im ersten Operanden (Wert oder Register) angegebene Bit des zweiten Operanden (Register oder Speicheradresse) in das Carry-Flag. BTC invertiert danach das gelesene Bit im Operanden, BTS setzt es auf Eins, BTR auf Null.

Die SET-Operationen setzen den Wert ihres Byte-Operanden (Register oder Speicheradresse) nach Prüfen eines Flags auf Null oder Eins.

```
cmp source, source
test source, source
bt index, source
btc index, source_dest
bts index, source_dest
btr index, source_dest
setCC dest
```

Die für `CC` einsetzbaren Flag-Kürzel sind dieselben wie bei den CMOV-Instruktionen (siehe Abbildung 6).

6.6 Logische Operationen

AND, OR und XOR führen die bekannten logischen Operationen bitweise auf ihren Operanden (Werte, Register oder Speicheradresse) aus. Sie setzen dabei die entsprechenden Flags (ZF, SF und PF, CF und OF werden gelöscht) und überschreiben ihren zweiten Operanden (Register oder Speicheradresse). NOT invertiert den Wert des Operanden und überschreibt diesen, ohne Flags zu verändern.

```
and source, source_dest
or source, source_dest
xor source, source_dest
not source_dest
```

AND und OR können benutzt werden, um auf Null, Vorzeichen und Parität zu prüfen, indem beide Operanden dasselbe Register enthalten. Wenn XOR dasselbe Register in beiden Operanden erhält, wird das Register auf Null gesetzt.

6.7 Kontrolltransfer

6.7.1 Jumps und Loops

JMP führt einen unbedingten Sprung zur angegebenen Adresse durch. Die Adresse kann relativ zum Instruktionszeiger `rip` oder indirekt in einem Register oder per Speicheradresse angegeben werden. Für relative Sprünge werden im Assemblercode normalerweise Labels als Sprungziele notiert, woraus bei der Assemblierung automatisch Offsets zu `rip` berechnet werden.

Die J-Instruktionen führen bedingte Sprünge auf Basis von Flag-Werten aus und sind immer relativ. JCXZ, JECXZ und JRCXZ sind bedingte Sprungbefehle, die aber nicht ein Flag prüfen, sondern zu einem Sprung führen, wenn das Register `cx`, `ecx` bzw. `rcx` den Wert 0 enthält.

Die LOOP-Befehle dekrementieren den Inhalt des `cx`-, `ecx`- oder `rcx`-Registers ohne ein Flag zu verändern und führen dann einen bedingten Sprung aus, wenn ihre Schleifenbedingung erfüllt ist. Die Bedingung für LOOP selbst ist erfüllt, wenn das Register nach dem Dekrementieren einen anderen Wert als Null enthält. Für LOOPE und LOOPZ ist die Bedingung erfüllt, wenn außerdem noch das Zero-Flag gesetzt ist, für LOOPNE und LOOPNZ muss das Zero-Flag (zusätzlich zur Bedingung von LOOP) gelöscht sein.

```
jmp label; jmp *address
jCC label
jcxz label; jecxz label; jrcxz label
loop label
loope label; loopne label
loopnz label; loopz label
```

Die für CC einsetzbaren Flag-Kürzel sind dieselben wie für die CMOV-Instruktionen (siehe Abbildung 6).

6.7.2 Prozeduraufrufe

Der CALL-Befehl dient zum Aufruf einer Prozedur. Er ist äquivalent zum JMP-Befehl, legt aber vor dem Sprung die Adresse des nächsten Befehls als Rücksprungadresse auf den Stack. RET holt diese Adresse später wieder vom Stack und führt den Rücksprung aus. Beide Befehle verändern dabei natürlich den Stack-Pointer `rsp`. RET kann als optionales Argument eine Bytezahl übernehmen, das angibt, wieviel nach dem Entfernen der Rücksprungadresse von `rsp` abgezogen werden soll (z.B. um übergebene Parameter vom Stack zu entfernen).

```
call label
call *address
ret
ret size
```

6.8 Flags

Zur Verwaltung des `rflags`-Registers gibt es eine ganze Reihe an Assembler-Befehlen: PUSHF, PUSHFD und PUSHFQ sowie POPF, POPFD und POPFQ dienen der Sicherung der Flags (`flags`, `eflags` bzw. `rflags`) auf dem Stack und der Wiederherstellung früher gesicherter Flags. Dabei wird natürlich der Stack-Pointer angepasst. Flags die nicht verändert werden können oder dürfen werden dabei nicht manipuliert.

CLC, CMC und STC dienen dem Löschen, Invertieren und Setzen des Carry-Flags. Dies ist z.B. nötig, bevor eine Shift- oder Rotate-Operation gestartet wird, die einen bestimmten Carry-Wert benötigt.

LAHF lädt das unterste Byte des `rflags`-Registers (dieses enthält CF, PF, AF, ZF und SF) in das `ah`-Register, SAHF setzt dieses Byte auf den Wert in `ah`.

STI und CLI setzen und löschen das *Interrupt-Flag*. Wenn dieses gelöscht ist, werden keine externen Interrupts behandelt. Darauf wird in dieser Anleitung nicht weiter eingegangen, es ist vor allem für die Systemprogrammierung relevant.

```
pushf; pushfd; pushfq
popf; popfd; popfq
clc   # clear carry
cnc   # complement carry
stc   # set carry
lahf  # load status flags into ah
sahf  # store ah into flags
sti   # set interrupt flag
cli   # clear interrupt flag
```

6.9 No Operation

Der NOP-Befehl tut nichts (außer Platz zu verbrauchen und den Instruktionszeiger zu erhöhen).

```
nop
```

6.10 Befehlssatzübersicht

Die folgende Abbildung enthält eine Übersicht über die wichtigsten Ganzzahlbefehle der AMD64-Architektur, ihre Semantik und Argumente. Abkürzungen: R... Register, I... Immediate-Wert, M... Memory, D... Displacement-Wert bzw. Sprungziel. X:Y steht für einen doppeltlangen Wert, wobei X die höherwertigen und Y die niederwertigen bits stellt.

Abkürzung	Argumente	Bedeutung	
`adc` *src, src_dest*	R, R/M R/M, R I, R/M	add with carry	*src_dest = src_dest + src + CF*
`add` *src, src_dest*	R, R/M R/M, R I, R/M	arithmetic add	*src_dest = src_dest + src*
`and` *src, src_dest*	R, R/M R/M, R I, R/M	bitwise and	*src_dest = src_dest and src*
`bt` *index, src*	R, R/M I, R/M	bit test	*CF = src[index]*
`btc` *index, src*	R, R/M I, R/M	bit test and complement	*CF = src[index]* *src[index] = not(src[index])*
`btr` *index, src*	R, R/M I, R/M	bit test and reset	*CF = src[index]* *src[index] = 0*
`bts` *index, src*	R, R/M I, R/M	bit test and set	*CF = src[index]* *src[index] = 1*
`clc`		clear carry	*CF = 0*
`cli`		clear interrupt flag	
`cmovCC` *src, dest*	R/M, R	conditional move	***if CC then** dest = src*
`cmp` *src1, src2*	R, R/M R/M, R I, R/M	compare	*src2 - src1 (setzt nut Flags)*
`cnc`		complement carry	*CF = not(CF)*
`dec` *src_dest*	R/M	decrement	*src_dest = src_dest - 1*
`div` *divisor*	R/M	divide	*rax = rdx:rax/divisor, rdx = Rest* *(und Varianten)*
`enter` *size, depth*	I, I	enter function, erzeugt Activation Record	
`idiv` *divisor*	R/M	signed divide	*rax = rdx:rax/divisor, rdx = Rest* *(und Varianten)*
`imul` *factor*	R/M	signed multiply	*rdx:rax = rax * factor* *(und Varianten)*
`imul` *factor, src_dest*	R/M, R	signed multiply	*src_dest = src_dest * factor*
`imul` *factor, src, dest*	I, R/M, R	signed multiply	*dest = src * factor*
`inc` *src_dest*	R/M	increment	*src_dest = src_dest + 1*
`jCC` *label*	D	conditional jump	***if CC then*** *jump to label*
`jcxz` *label*	D	jump if cx zero	***if cx == 0 then*** *jump to label*
`jecxz` *label*	D	jump if ecx zero	***if ecx == 0 then*** *jump to label*
`jmp` *label*	D	jump relative	*jump to label*
`jmp*` *address*	R/M	jump indirect	*jump to address*
`jrcxz` *label*	D	jump if rcx zero	***if rcx == 0 then*** *jump to label*
`lahf`		load status flags into ah	*ah = low_byte_of(flags)*
`lea` *src, dest*	R/M, R	load effective address	*dest = address_of(src)*
`leave`		leave function, entfernt einen Activation Record	

Abkürzung	Argumente	Bedeutung	
`loop` *label*	D	loop if cx (und Varianten)	*cx = cx - 1* *if cx != 0 then* *jump to label* *(und Varianten)*
`loope` *label*	D	loop if cx and equal (und Varianten)	*cx = cx - 1* *if cx != 0 and ZF then* *jump to label* *(und Varianten)*
`loopne` *label*	D	loop if cx and not equal (und Varianten)	*cx = cx - 1* *if cx != 0 and not(ZF) then* *jump to label* *(und Varianten)*
`loopnz` *label*	D	loop if cx and not zero (und Varianten)	*cx = cx - 1* *if cx != 0 and not(ZF) then* *jump to label* *(und Varianten)*
`loopz` *label*	D	loop if cx and zero (und Varianten)	*cx = cx - 1* *if cx != 0 and ZF then* *jump to label* *(und Varianten)*
`mov` *src, dest*	R, R/M R/M, R I, R/M	move	*dest = src*
`movsx` *src, dest*	R/M, R	move with sign extension	*dest = sign_extend(src)*
`movzx` *src, dest*	R/M, R	move with zero extension	*dest = zero_extend(src)*
`mul` *factor*	R/M	multiply	*rdx:rax = rax * factor* *(und Varianten)*
`nop`		no operation	
`not` *src_dest*	R/M	bitwise not	*src_dest = not(src_dest)*
`or` *src, src_dest*	R, R/M R/M, R I, R/M	bitwise or	*src_dest = src_dest or src*
`pop` *dest*	R/M	pop from stack	
`popf`		pop flags from stack	*pop value of `flags` register*
`popfd`		pop flags from stack	*pop value of `eflags` register*
`popfq`		pop flags from stack	*pop value of `rflags` register*
`push` *src*	R/M	push onto stack	
`pushf`		push flags onto stack	*push value of `flags` register*
`pushfd`		push flags onto stack	*push value of `eflags` register*
`pushfq`		push flags onto stack	*push value of `rflags` register*
`rcl` *count, src_dest*	I, R/M `%cl`, R/M	rotate through carry left	*src_dest = rol(src_dest, count, **CF**)*
`rcr` *count, src_dest*	I, R/M `%cl`, R/M	rotate through carry right	*src_dest = ror(src_dest, count, **CF**)*
`rol` *count, src_dest*	I, R/M `%cl`, R/M	rotate left	*src_dest = rol(src_dest, count)*
`ror` *count, src_dest*	I, R/M `%cl`, R/M	rotate right	*src_dest = ror(src_dest, count)*
`sahf`		store ah into flags	*low_byte_of(`flags`) = `ah`*
`sal` *count, src_dest*	I, R/M `%cl`, R/M	shift arithmetic left	*src_dest = src_dest << count*
`sar` *count, src_dest*	I, R/M `%cl`, R/M	shift arithmetic right (signed)	*src_dest = src_dest >> count* *(signed)*
`sbb` *src, src_dest*	R, R/M R/M, R I, R/M	sub with borrow	*src_dest = src_dest - src - **CF***
`setCC` *dest*	R/M	set to flag value	*dest = CC*
`shl` *count, src_dest*	I, R/M `%cl`, R/M	shift left	*src_dest = src_dest << count*

Abkürzung	Argumente	Bedeutung	
`shld` *count, src, src_dest*	I, R, R/M %cl, R, R/M	shift left double	*src_dest = src_dest:src << count*
`shr` *count, src_dest*	I, R/M %cl, R/M	shift right (unsigned)	*src_dest = src_dest >> count (unsigned)*
`shrd` *count, src, src_dest*	I, R, R/M %cl, R, R/M	shift right double (unsigned)	*src_dest = src:src_dest >> count (unsigned)*
`stc`		set carry	*CF = 1*
`sti`		set interrupt flag	
`sub` *src, src_dest*	R, R/M R/M, R I, R/M	arithmetic subtract	*src_dest = src_dest - src*
`test` *src1, src2*	R, R/M R/M, R I, R/M	test	*src2 and src1 (setzt nur Flags)*
`xor` *src, src_dest*	R, R/M R/M, R I, R/M	bitwise xor	*src_dest = src_dest xor src*

Tabelle 1: Wichtige Ganzzahlinstruktionen der AMD64-Architektur.

7 128-Bit-Medienbefehle

Als Zusatz zur ursprünglichen Intel-x86-Architektur wurden schon bei früheren Erweiterungen SIMD-Befehle (*Single Instruction Multiple Data*) hinzugefügt, die auch in der AMD64-Architektur vorhanden sind. Diese dienen zur gleichzeitigen Bearbeitung eines ganzen Vektors von Operanden mit einem einzigen Befehl, wie es zum Beispiel für wissenschaftliche und Multimedia-Anwendungen sinnvoll sein kann. Diese Anleitung stellt einige Ganzzahlbefehle aus diesen Befehlssatzerweiterungen kurz vor, die auf den in Abschnitt 3.3 vorgestellten Registern operieren.

7.1 Datentransferoperationen

MOV-Operationen transferieren Daten zwischen `xmm`-Registern, General-Purpose-Registern und Speicherbereichen. Abbildung 7 beschreibt die verschiedenen Move-Befehle, darin stehen *G* für General-Purpose-Register, *X* für `xmm`-Register und *M* für eine Speicheradresse. *ZE* steht für *Zero-Extension*, d.h. die nicht vom Transfer betroffenen Bits des Zieloperanden werden mit Nullen aufgefüllt. (Die Argumentreihenfolge ist identisch mit der von Ganzzahl-MOV-Operationen, d.h. Zieloperand zuletzt).

7.2 Auspackoperationen

Eine häufige Anwendung von Medieninstruktionen ist das Extrahieren von gepackten Daten. Zum Beispiel ist es häufig nötig, aus Vektoren zusammengesetzter RGB-Farbwerte einzeln alle R-, alle G- und alle B-Werte zu extrahieren.

Instruktion	Operandengröße	Operanden	Anmerkung
`MOVD`	32 oder 64 Bit	$G, M \rightarrow X(ZE)$ oder $X \rightarrow G, M$	Bearbeitet die unteren 32 oder 64 Bits des `xmm`-Registers.
`MOVQ`	64 Bit	$M \rightarrow X(ZE)$ oder $X \rightarrow M$ oder $X \rightarrow X$	Bearbeitet die unteren 64 Bits des `xmm`-Registers.
`MOVDQA`	128 Bit	$M \rightarrow X$ oder $X \rightarrow M$ oder $X \rightarrow X$	Speicheradressen müssen 128-bit-aligned sein.
`MOVDQU`	128 Bit	$M \rightarrow X$ oder $X \rightarrow M$ oder $X \rightarrow X$	Wie MOVDQA, aber Speicheradressen können *unaligned* sein.

Abbildung 7: 128-Bit-Move-Befehle

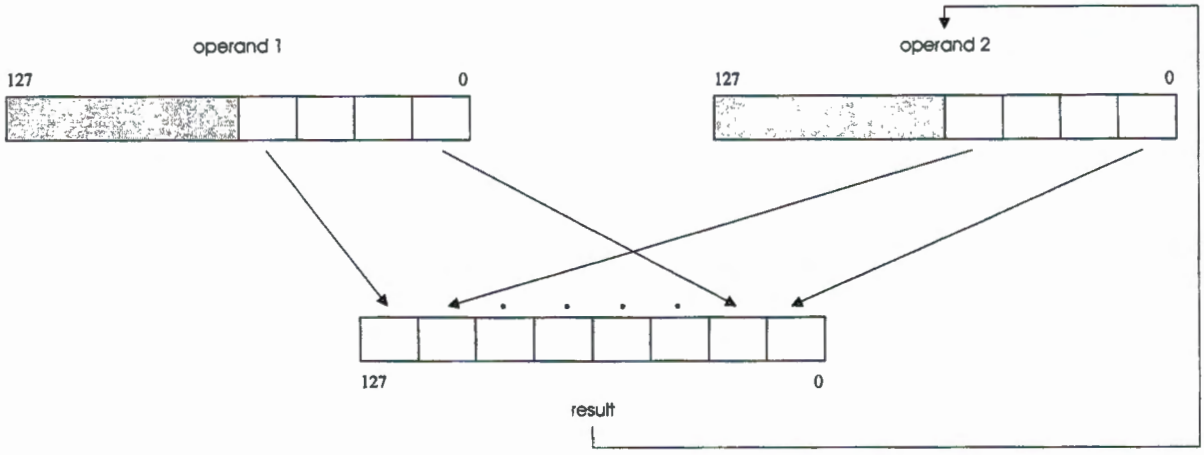


Abbildung 8: Illustration der Auspackoperation PUNPCKLWD (nach: *AMD64 Architecture Programmer's Manual*)

Für solche Auspackoperationen bietet die AMD64-Architektur die 128-Bit-PUNPCK-Befehle. Jede dieser Operationen nimmt zwei Operanden: ein `xmm`-Register oder eine Speicheradresse als ersten Quellvektor und ein `xmm`-Register als zweiten Quell- und Zielvektor. Die Instruktionen gehen die beiden Vektoren Element für Element durch, und schreiben die Elemente abwechselnd (*interleaved*, d.h. jeweils zuerst ein Element aus dem zweiten, dann eines aus dem ersten Vektor) in den Zielvektor. Damit alle Elemente in den Zielvektor passen, betrachten die Befehle jeweils nur die obere oder untere Hälfte (höherwertiges oder niederwertiges Quadword) der Quellvektoren.

Abbildung 8 illustiert dies anhand der PUNPCKLWD-Instruktion, Abbildung 9 zeigt eine Liste der verschiedenen Auspackmöglichkeiten.

7.3 Rechen- und Vergleichs-Befehle

Für die Abwicklung der Grundrechenarten und Schiebeoperationen gibt es 128-Bit-Vektorbefehle, deren Funktionalität im Wesentlichen jener der in Abschnitt 6 beschriebenen arithmetischen und Shift-Instruktionen entspricht, nur dass die Vektorbefehle auf den einzelnen Elementen zweier Vektoren arbeiten (und zwar wird immer ein Element des einen Vektors mit dem entsprechen-

Instruktion	Elementgröße	Entpackte Elemente
PUNPCKHBW	8 Bit	Höherwertiges Quadword
PUNPCKHWD	16 Bit	Höherwertiges Quadword
PUNPCKHDQ	32 Bit	Höherwertiges Quadword
PUNPCKHQDQ	64 Bit	Höherwertiges Quadword
PUNPCKLBW	8 Bit	Niederwertiges Quadword
PUNPCKLWD	16 Bit	Niederwertiges Quadword
PUNPCKLDQ	32 Bit	Niederwertiges Quadword
PUNPCKLQDQ	64 Bit	Niederwertiges Quadword

Abbildung 9: 128-Bit-Auspack-Befehle

den Element des anderen Vektors kombiniert) und die Ergebnisse wieder in die entsprechenden Elemente in einem Vektor speichern. Der Quell-Operand kann dabei im Speicher liegen, wobei die Adresse des Vektors auf 16 Bytes (128 bits) ausgerichtet sein muss. Der andere Operand ist immer ein `xmm`-Register.

Die Vektorbefehle beeinflussen die in Abschnitt 3.1 vorgestellten Flags nicht. Die Vergleichsbefehle liefern als Resultat wieder einen Vektor, wobei das entsprechende Element 0 ist, wenn das Ergebnis falsch ist, und alle Bits gesetzt hat, wenn das Ergebnis wahr ist. Mit `PMOVMSKB` *`xmm`*, *`reg32`* kann man die 16 höchstwertigen Bits der einzelnen Bytes eines XMM-Registers in ein General-Purpose-Register übertragen (als mit 0en aufgefüllter 16-Bit-Wert), sodass man in weiterer Folge z.B. Schleifen steuern kann.

Die Funktionalität des Befehls PMADDW ist etwas anders: PMADDW multipliziert die 16-Bit-Elemente des einen Vektors mit jenen des zweiten Vektors und addiert die Resultate (siehe Abbildung 10). Durch eine Kombination mit PADDD kann diese Instruktion benutzt werden, um effizient Skalarprodukte zu berechnen. Abbildung 11 gibt eine Übersicht über die arithmetischen und Shift-Vektorbefehle.

8 Assemblerdirektiven

Der GNU-Assembler stellt eine Reihe von Assembleranweisungen zur Verügung, von denen einige hier beschrieben werden. Einige der Anweisungen sind nur deshalb in der Liste, weil Sie vom *GCC*-Compiler erzeugt werden. Genauere Informationen und eine vollständige Liste finden Sie in der Assembler-Dokumentation unter *::Pseudo Ops.*

`.align` *Zahl*

Sorgt dafür, dass die folgenden Befehle und Daten so angeordnet werden, dass ihre Adressen an bestimmten Speichergrenzen ausgerichtet (aligned) werden. Das Verhalten dieser Direktive ist unterschiedlich, je nachdem welches Binärformat generiert werden soll: Beim *ELF*-Format gibt das Argument das Alignment in Bytes an. `.align 8` bedeutet beispielsweise, dass die Adressen Vielfache von Acht sein und alle dazwischenliegenden unbenutzen Bytes auf Null (oder NOP-Instruktionen, je nach System und Speicherbereich) aufgefüllt werden sollen. Beim *a.out*-Format gibt die Zahl hingegen die Anzahl der unteren Adressbits an, die Null sein

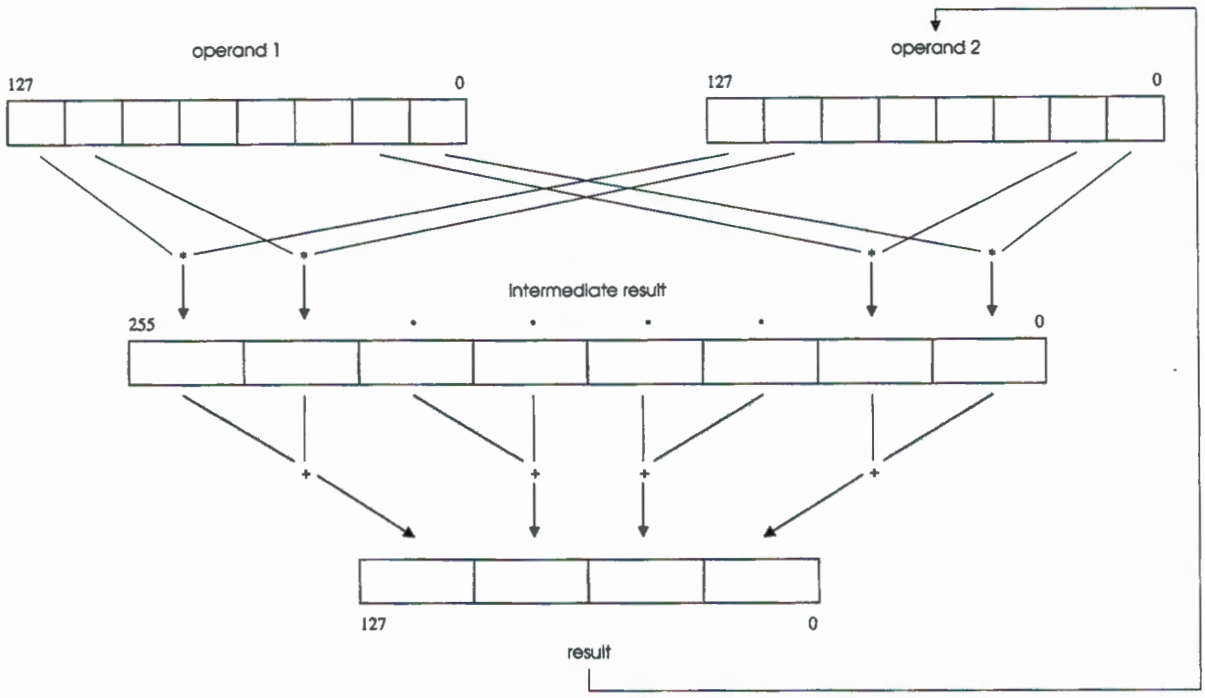


Abbildung 10: Illustration der Operation PMADDW (nach: *AMD64 Architecture Programmer's Manual*)

müssen. Damit wäre `.align 3` die der oberen Direktive entsprechende Anweisung.

`.ascii` *Text* Speichert einen in doppelte Hochkommata eingeschlossenen Text ab.

`.asciiz` *Text* Speichert einen in doppelte Hochkommata eingeschlossenen Text ab und schließt ihn mit Null ab.

`.byte` *expr [, expr]** Speichert die auf 8 Bit abgeschnittenen Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Hinter dem Ausdruck kann noch ein durch einen Doppelpunkt getrennter Wiederholungsfaktor stehen.

`.comm` *name, expr* Reserviert einen Speicherbereich mit mindestens *expr* Bytes unter dem Namen *name*. Der Linker legt alle Common-Blöcke mit dem selben Namen übereinander.

`.data` Alle nachfolgenden Daten werden in der `.data`-Sektion angelegt. Die `.text`- und `.data`-Sektionen werden häufig auch als „Segmente" bezeichnet, wobei diese nichts mit der von x86- und AMD64-Architekturen unterstützten Speichersegmentierung zu tun haben.

`.double` *expr [, expr]** Speichert die nachfolgenden Ausdrücke als 64-Bit-Gleitkommazahlen aufeinanderfolgend ab. Siehe `.byte`.

`.endr` Das Ende eines Repeat-Blockes. Siehe `.rpt`.

`.err` Wird vom Übersetzer verwendet. Beendet den Assembler.

`.extern` *name [size]* Definiert ein globales, externes Symbol mit dem Namen *name*. Der optionale Parameter *size* gibt die Größe in Bytes an.

`.file` *number string* Wird vom Compiler verwendet. Ordnet eine Dateinummer dem Dateinamen zu.

`.float` *expr [, expr]** Speichert die nachfolgenden Ausdrücke als 32-Bit-Gleitkommazahlen auf-

Instruktion	Elementgröße	Operation
PADDB, PADDW, PADDD, PADDQ	8/16/32/64 Bit	Einfache Addition (Overflows werden ignoriert)
PADDSB, PADDSW, PADDUSB, PADDUSW	8/16/32/64 Bit	Addition mit Sättigung (Overflows führen zum größten oder kleinsten darstellbaren Wert)
PSUBB, PSUBW, PSUBD, PSUBQ	8/16/32/64 Bit	Einfache Subtraktion
PSUBSB, PSUBSW	8/16 Bit	Subtraktion mit Sättigung
PSUBUSB, PSUBUSW	8/16 Bit	Subtraktion unsigned mit Sättigung
PMULHW	16 Bit	Multipliziert signed 16-Bit-Elemente miteinander und schreibt die oberen 16 Bit des 32-Bit-Ergebnisses in das Zielelement
PMULLW	16 Bit	Wie PMULHW, schreibt aber die unteren 16 Bit
PMULHUW	16 Bit	Wie PMULHW, aber mit vorzeichenlosen (unsigned) Werten
PMADDWD	16 Bit	Multiplikation mit Addition (siehe Text)
PSLLW, PSLLD, PSLLQ	16/32/64 Bit	Logisches Linksschieben
PSRLW, PSRLD, PSRLQ	16/32/64 Bit	Logisches Rechtsschieben
PSLLDQ, PSRLDQ	128 Bit	Logisches Links-/Rechtsschieben, jedoch ist das erste Argument (count) die Anzahl der zu schiebenden **Bytes**, nicht Bits
PSRAW, PSRAD	16/32 Bit	Arithmetisches Rechtsschieben, d.h. Rechtsschieben, wobei das Sign-Bit erhalten bleibt
PMAXSW, PMINSW	16 Bit	Maximum/Minimum (signed)
PMAXUB, PMINUB	8 Bit	Maximum/Minimum (unsigned)
PCMPEQB, PCMPEQW, PCMPEQD	8/16/32 Bit	Gleichheit
PCMPGTB, PCMPGTW, PCMPGTD	8/16/32 Bit	Größer als (signed)

Abbildung 11: 128-Bit-Arithmetik- und Schiebebefehle

einanderfolgend ab. Siehe `.byte`.

`.globl` *name* — Deklariert *name* als externes Symbol. Falls *name* anderswo im Programm definiert wird, wird das Symbol vom Linker exportiert, sonst wird es importiert.

`.ident` — GAS ignoriert diese Direktive.

`.lcomm` *name, size* — Für das Symbol *name* werden in der `bss`-Sektion *size* Bytes Speicherplatz reserviert.

`.long` *expr [, expr]** — Speichert die auf 32 Bit abgeschnittenen Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Siehe `.byte`.

`.quad` *expr [, expr]** — Speichert 64 Bit große Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Siehe `.byte`.

`.rept` *expr* — Wiederholt alle Befehle die zwischen `.rept` und `.endr` stehen *expr* mal. Es dürfen keine Label in diesen Befehlen vorkommen. `.rept`-Anweisungen dürfen nicht geschachtelt werden.

`.section` *name* — Der folgende Code wird in die angegebene Sektion assembliert. Sektionen

werden in dieser Anleitung nicht genauer beschrieben.

`.size` *name, expr* Legt die Größe des Symbols *name* in Bytes fest.

`.space` *expr* Füllt *expr* aufeinanderfolgende Bytes mit Null.

`.string` *string* Definiert einen String.

`.struct` *expr* Ermöglicht die Definition von Datenstrukturen. In nachfolgenden Anweisungen wie `.word` oder `.byte` vorkommende Label erhalten einen Wert relativ zur `.struct`-Anweisung plus *expr*.

`.text` Alle nachfolgenden Daten werden in der `.text` Sektion angelegt. Die `.text`- und `.data`-Sektionen werden häufig auch als „Segmente“ bezeichnet, wobei diese nichts mit der von x86- und AMD64-Architekturen unterstützten Speichersegmentierung zu tun haben.

`.type` *name, typedescr* Spezifiziert den Typen eines Symbols als Funktion oder Objekt.

`.uleb128` *expr [, expr]** Speichert Werte im kompakten *Unsigned Little Endian Base 128*-Format ab.

`.version` *string* Definiert Versionsinformation.

`.word` *expr [, expr]** Speichert die auf 16 Bit abgeschnittenen Werte von beliebig vielen Ausdrücken aufeinanderfolgend ab. Siehe `.byte`.

name = *expr* Weist dem Symbol *name* den Wert des Ausdrucks *expr* zu.

name = *register* Das Register *register* erhält den Namen *name*.

Chapter 9

make: A Program for Maintaining Programs

9.1 Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator such as yacc or lex. The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules.

Unfortunately, it is very easy to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, you may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can render obsolete a dozen other files. Forgetting to compile a routine that you've changed or one that uses changed declarations result in a program that does not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

make is a program that mechanizes many of the activities of program development and maintenance. If the information on inter-file dependencies and command sequences is stored in a file, the simple command

```
$ make
```

is frequently sufficient to update the relevant files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the make command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — make — test . . .

make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. This chapter is a guide for users of make.

> NOTE: The Domain Software Engineering Environment (DSEE) is an optional product that provides users with an integrated programming environment. Some of the features of make are similar to those of DSEE, although DSEE provides additional features as well. For more information about DSEE, see *Getting Started with the Domain Software Engineering Environment* (008788).

9.2 Basic Features

The basic operation of make is to update a target file by ensuring that all of the files on which it depends exist and are current, then creating the target if it has not been modified since its dependents were. make does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example. A program named prog is made by compiling and loading three C language files x.c, y.c, and z.c with the IS library. By convention, the output of the C compilations are found in files named x.o, y.o, and z.o. Assume that the files x.c and y.x share some declarations in a file named defs, but that z.c does not. That is, x.c and y.c have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
        cc x.o y.o z.o -lS -o prog

x.o y.o : defs
```

If this information is stored in a file named makefile, the command

```
$ make
```


performs the operations needed to recreate prog after any changes had been made to any of the four source files x.c, y.c, z.c, or defs.

make operates by using three sources of information: a user-supplied description file (as above), filenames and "last-modified" times from the file system, and built-in rules to bridge some of the gaps.

In our example, the first line says that prog depends on three .o files. Once these object files are current, the second line describes how to load them to create prog. The third line says that x.o and y.o depend on the file defs.

From the file system, make discovers that there are three .c files corresponding to the needed .o files. It then uses built-in information about how to generate an object from a source file (i.e., issue a cc command with the -c option).

If make did not have the ability to determine automatically what needs to be done, this longer description file would be necessary:

```
prog : x.o y.o z.o
        cc x.o y.o z.o -lS -o prog

x.o : x.c defs
        cc -c x.c

y.o : y.c defs
        cc -c y.c

z.o : z.c
        cc -c z.c
```

If none of the source or object files had changed since the last time prog was made, all of the files would be current, and the command

```
$ make
```

simply announces this fact and stops. If, however, the defs file had been edited, x.c and y.c (but not z.c) are recompiled, and then prog is created from the new .o files. If only the file y.c had changed, only it is recompiled, but prog must still be reloaded.

If no target name is given on the make command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

```
$ make x.o
```

recompiles x.o if x.c or defs had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise, the current time is used. It is often useful for programs to in-

clude rules with mnemonic names and commands that don't actually produce a file with that name. These entries can take advantage of make's ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files.

You can also maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be enclosed in parentheses. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical.

> NOTE: To get a dollar sign, escape it with another dollar sign. The sequence $$ is escaped to $.

All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command:

- $*
- $@
- $?
- $<

They are discussed later. The following fragment shows the use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
```


The command

```
$ make
```

loads the three object files with the lS library. The command

```
$ make "LIBES= -ll -lS"
```

loads them with both the lex (-ll) and the standard (-lS) libraries, because macro definitions on the command line override definitions in the description. (The shell requires that you quote arguments that include embedded blanks.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

9.3 Description Files and Substitutions

A description file contains three types of information:

- Macro definitions
- Dependency information
- Executable commands

A comment convention is also supplied: all characters after a pound sign (#) are ignored, as is the pound sign itself. Blank lines and lines beginning with this character are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is an identifier followed by an equal sign (=); the identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the make command line.

The general form of an entry in a description file is:

target1 [*target2* . . .] :[:] [*dependent1* . . .] [; *commands*] [# . . .]
[(tab) *commands*] [# . . .]

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters * and ? are expanded when the line is evaluated.) A command is any string of characters not including a pound sign (except when the pound sign is in quotes) or newline. Commands may appear either

- After a semicolon on a dependency line
- On lines beginning with a tab immediately following a dependency line

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

For the more common single-colon case, a command sequence may be associated with at most one of these dependency lines. If the target is out-of-date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default creation rule may be invoked.

In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double-colon form is particularly useful in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). make normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the -i option is specified on the make command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some Domain commands return meaningless status). Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., cd and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set:

- $@ is set to the name of the file to be "made"
- $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below)

- $< is the name of the related file that caused the action
- $* is the prefix shared by the current and the dependent filenames.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If no such name exists, make prints a message and stops.

9.4 Using make

The make command takes four kinds of arguments: macro definitions, flags, description filenames, and target filenames. The prototypical make command line is:

$ make [*flags*] [*macro definitions*] [*targets*]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments are made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are as follows:

-i	Ignore error codes returned by invoked commands. This mode is entered if the fake target name ".IGNORE" appears in the description file.
-s	Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ".SILENT" appears in the description file.
-r	Do not use the built-in rules.
-n	No execute mode. Print commands, but do not execute them. Even lines beginning with an at-sign (@) are printed.
-t	Touch the target files (causing them to be up-to-date) rather than issue the usual commands.
-q	Question. The make command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.
-p	Print out the complete set of macro definitions and target descriptions.
-d	Debug mode. Print out detailed information on files and times examined.
-f	Description filename. The next argument is assumed to be the name of a description file. A filename of "-" denotes the standard input. If no -f arguments appear, the file named makefile in the current directory is read.

NOTE: The contents of the description files override any built-in rules present.

Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is "made".

9.4.1 Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	FORTRAN source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.

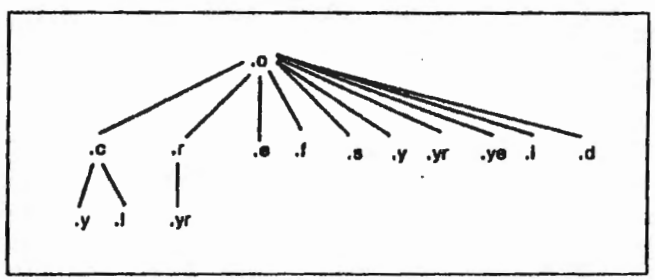


Figure 9-1. make Dependency Tree

If the file x.o is needed and there is an x.c in the description or directory, the x.o is compiled. If there is also an x.l, that grammar is run through lex before compiling the result. However, if there is no x.c but there is an x.l, then make discards the intermediate C-language file and uses the direct link shown in Figure 9-1.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
$ make CC=newcc
```

causes the newcc command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
$ make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

9.4.2 An Example

To illustrate the use of make, here's the description file used to maintain the make command itself. The code for make is spread over many C source files and a yacc grammar. The description file contains:

```
# Description file for the make command

P = und -3 | opr -r2 # send to GCOS to be printed
FILES = makefile version.c defs main.o doname.c misc.c files.c
dosys.cgram.y lex.c gcos.o
OBJECTS = version.o main.o doname.o misc.c files.o dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make: $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du
```

```
install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print: $(FILES)      # print recently changed files
        pr $?.| $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c version.c gram.o
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

make usually prints each command before issuing it. Typing make with no arguments in a directory containing only the source and description file outputs the following:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars are mentioned by name in the description file, make found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the size command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file print is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since print was

touched. The printed output can be sent to a different printer or to a file by changing the definition of the P macro:

```
$ make print "P = opr -sp"
```

or

```
$ make print "P= cat >zap"
```

9.5 Suggestions and Warnings

The most common difficulties arise from make's specific understanding of what constitutes a dependency. If file x.c has a `#include "defs"` line, then the object file x.o depends on defs; the source file x.c does not. (If defs is changed, it is not necessary to do anything to the file x.c, while it is necessary to recreate x.o.)

To discover what make would do, the -n option is very useful. The command

```
$ make -n
```

orders make to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the -t (touch) option can save a lot of time, instead of issuing a large number of superfluous recompilations, make updates the modification times on the affected file. Thus, the command

```
$ make -ts
```

("touch silently") causes the relevant files to appear up-to-date. Be careful, though, because this mode of operation subverts the intention of make and destroys all memory of the previous relationships.

The -d (debugging) option causes make to print a very detailed description of its activities, including file times. The output is verbose, and recommended only as a last resort.

9.6 Summary of Suffixes and Rules

The make program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the -r option is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; make looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, make acts as described earlier. The transformation rule names are the concatenation of the two suffixes.

The name of the rule to transform a .r file to a .o file is thus r.o. If the rule is present and no explicit command sequence has been given in your description files, the command sequence for the rule r.o is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, you can just add an entry for ".SUFFIXES" in its own description file; the dependents are added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed.)

The following is an excerpt from the default rules file:

```
.SUFFIXES: .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=

LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=

.c.o:
        $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o:
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o:
        $(AS) -o $@ $<
```



```
.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.y.c:
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

9.7 Extensions to make

NOTE: The following sections describe extensions to make that were added after the preceding documentation was written.

While make is an excellent program administration tool, it had a number of limitations that hindered its use for large-scale software development:

- Handling of libraries was tedious.
- Handling of the Source Code Control System (SCCS) filename format was difficult or impossible.
- Environment variables were completely ignored.
- Ability to maintain files in a remote directory was inadequate.

The augmented version of make eliminates these problems. The additional features are within the original syntactic framework of make and few, if any, new syntactical entities are introduced. A notable exception is the include file capability.

9.8 Environment Variables

Environment variables are read and added to the macro definitions each time make executes. Precedence is a prime consideration in doing this properly. The following describes make's interaction with the environment. A new macro, MAKEFLAGS, is maintained by make and defined as the collection of all input flag arguments into a string (without minus signs). The new macro is exported and thus accessible to further invocations of make. Command line flags and assignments in the makefile update MAKEFLAGS. Thus, to describe how the environment interacts with make, consider the MAKEFLAGS macro (environment variable).

9

lex — a Lexical Analyzer Generator

lex is a program generator designed for lexical processing of character input streams. lex accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions. The regular expressions are specified by the programmer in the source specifications given to lex. The lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the programmer are executed. The lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by lex, the corresponding fragment is executed.

The programmer supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general-purpose programming language employed for the programmer's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the programmer's freedom to write actions is unimpaired. This avoids forcing the programmer who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by lex. The program fragments written by the programmer are executed in the order in which the corresponding regular expressions occur in the input stream.

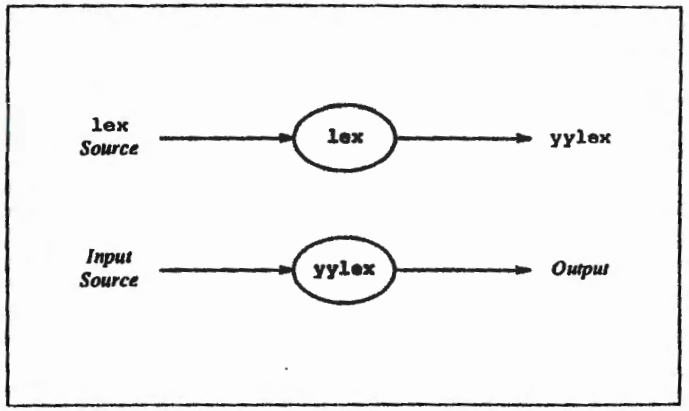
The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream is then backed up to the end of the current partition, so that the programmer has general freedom to manipulate it.

lex is designed to simplify interfacing with yacc, which is described in the next chapter.

`lex` is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called 'host languages.' Just as general-purpose languages can produce code to run on different computer hardware, `lex` can write code in different host languages. The host language is used for the output code generated by `lex` and also for the program fragments added by the programmer. Compatible run-time libraries for the different host languages are also provided. This makes `lex` adaptable to different environments and different programmer. Each application may be directed to the combination of hardware and host language appropriate to the task, the programmer's background, and the properties of local implementations.

`lex` turns the programmer's expressions and actions (called `source` in this document) into the host general-purpose language; the generated program is named `yylex`. The `yylex` program recognizes expressions in a stream (called `input` in this document) and performs the specified actions for each expression as it is detected — see Figure 9-1 below.

Figure 9-1 *An overview of* `lex`



For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines (followed by newlines):

```
%%
[ \t]+$ ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates 'one or more ...'; and the $ indicates 'end-of-line'. No action is specified, so the program generated by `lex` (`yylex`) ignores these characters. Everything else is

* followed by newline

copied to the output stream. To change any remaining string of blanks or tabs to a single blank, add another rule:

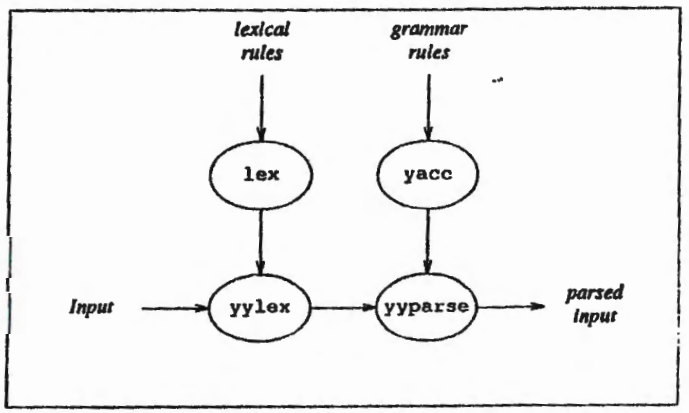
```
%%
[ \t]+$ ;
[ \t]+  printf(" ");
```

The finite automaton generated for this source scans for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the ends of lines, and the second rule all remaining strings of blanks or tabs.

`lex` can also be used with a parser generator to perform the lexical analysis phase.

`lex` can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. `lex` can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface `lex` and `yacc`. `lex` programs recognize only regular expressions; `yacc` writes parsers that accept a large class of context-free grammars, but require a lower-level analyzer to recognize input tokens. Thus, a combination of `lex` and `yacc` is often appropriate. When used as a preprocessor for a later parser generator, `lex` is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 9-2. Additional programs, written by other generators or by hand, can be added easily to programs written by `lex`.

Figure 9-2 `lex` *with* `yacc`



`yacc` programmers will realize that the name `yylex` is what `yacc` expects its lexical analyzer to be named, so that the use of this name by `lex` simplifies interfacing.

`lex` generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a `lex` program to recognize and partition an input stream is proportional to the length of the input. The number of `lex` rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by `lex`.

In the program written by `lex`, the programmer's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the programmer to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

`lex` is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for `ab` and another for `abcdefg`, and the input stream is `abcdefh`, `lex` recognizes `ab` and leave the input pointer just before "`cd...`" Such backup is more costly than processing simpler languages.

9.1. `lex` Source

The general format of `lex` source is:

```
{definitions}
%%
{rules}
%%
{programmer subroutines}
```

where the definitions and the programmer subroutines are often omitted. The second `%%` is optional, but the first is required to mark the beginning of the rules. The absolute minimum `lex` program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of `lex` programs shown above, the *rules* represent the programmer's control decisions; they are a table, in which the left column contains *regular expressions* (see section 9.2) and the right column contains *actions*, program fragments to be executed when the expressions

```
integer printf("found keyword INT");
```

to look for the string `integer` in the input stream and print the message 'found keyword INT' whenever it appears. In this example the host procedural language is C and the C library function `printf()` is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is

merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. `lex` rules such as

```
colour  printf("color");
mechanise       printf("mechanize");
petrol  printf("gas");
```

would be a start. These rules are not quite enough, since the word `petroleum` would become `gaseum`; a way of dealing with this is described later.

9.2. `lex` Regular Expressions

The definitions of regular expressions are very similar to those in the editors *ex*(1) and *vi*(1). A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string `integer` wherever it appears and the expression

```
a57D
```

looks for the string `a57D`.

Operators

The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape must be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz"++"
```

matches the string `xyz++` when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the programmer can avoid remembering the list above of current operator characters, and is safe should further extensions to `lex` lengthen the list.

An operator character may also be turned into a text character by preceding it with `\` as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as

explained above, blanks or tabs end a rule. Any blank character not contained within [ ] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character Classes

Classes of characters can be specified using the operator pair [ ]. The construction `[abc]` matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: \, -, and ^. The - character indicates ranges. For example,

```
[a-z0-9<>_]
```

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation-dependent and generates a warning message. For example, `[0-z]` in ASCII is many more characters than it is in EBCDIC. If it is desired to include the character - in a character class, it should be first or last, thus:

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the system's character set. Thus

```
[^abc]
```

matches all characters except a, b, or c, including all special or control characters; and

```
[^a-zA-Z]
```

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary Character

To match almost any character, the operator character

```
.
```

(period) is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional Expressions

The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either `ac` or `abc`.

Repeated Expressions

Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive a characters, including zero; while

```
a+
```

is one or more instances of a. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping

The operator | indicates alternation:

```
(ab|cd)
```

matches either `ab` or `cd`. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
(ab|cd+)?(ef)*
```

matches such strings as `abefef`, `efefef`, `cdef`, or `cddd`; but not `abc`, `abcd`, or `abcdef`.

Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression is only be matched at the beginning of a line This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [ ] operators. If the very last character is $, the expression is only be matched at the end of a line (when immediately followed by newline).

The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
ab/cd
```

matches the string ab, but only if it is followed by cd. Thus

```
ab$
```

is the same as

```
ab/\n.
```

Left context is handled in lex by *start conditions* as explained in section 9.9 — *Left Context-Sensitivity*. If a rule is only to be executed when the lex automaton interpreter is in start condition x, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered 'being at the beginning of a line' to be start condition ONE, then the ^ operator would be equivalent to

```
<ONE>.
```

Start conditions are explained more fully below.

Repetitions and Definitions

The operators { } specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named digit and inserts it at that point in the expression. The definitions are given in the first part of the lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of a.

Finally, initial % is special, being the separator for lex source segments.

9.3. lex Actions

When an expression written as above is matched, lex executes the corresponding action. This section describes some features of lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the lex programmer who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When lex is being used with yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action does this. A frequent rule is

```
[ \t\n] ;
```

which ignores the three spacing characters (blank, tab, and newline).

Another easy way to avoid writing actions is the action character |, which indicates that the action to be used for this rule is the action given for the next rule. The previous example could also have been written

```
" "             |
"\t"            |
"\n"            ;
```

with the same result. The quotes around \n and \t are not required.

Actual Text that Matched

In more complex actions, the programmer often wants to know the actual text that matched some expression like [a-z]+. lex leaves this text in an external character array named yytext.
Thus, to print the name found, a rule like

```
[a-z]+ printf("%s", yytext);
```

prints the string in yytext. The C function printf accepts a format argument and data to be printed; in this case, the format is 'print string' (% indicating data conversion, and s indicating string type), and the data are the characters in yytext. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+ ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches read() it normally matches the instances of read contained in bread or readjust; to avoid this, a rule of the form [a-z]+ is needed. This is explained further below.

Length of Matched Text

Sometimes it is more convenient to know the end of what has been found; hence lex also provides a count yyleng of the number of characters matched. To count both the number of words and the number of characters in words in the input, the programmer might write

```
[a-zA-Z]+       {words++; chars += yyleng;}
```

which accumulates in chars the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yyleng-1].
```

yymore *and* yyless

Occasionally, a `lex` action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, `yymore()` can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in `yytext`. Second, `yyless(n)` may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument n indicates the number of characters to be retained in `yytext`. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write:

```
\"[^"]+ {
        if (yytext[yyleng-1] == '\\')
            yymore();
        else
            ... normal programmer processing
        }
```

which, when faced with a string such as `"abc\"def "` first matches the five characters `"abc\`; then the call to `yymore()` tacks the next part of the string, `"def`, onto the end. Note that the final quote terminating the string should be picked up in the code labeled 'normal processing'.

The function `yyless()` might be used to reprocess text in various circumstances. Consider the problem of resolving (in old-style C) the ambiguity of '=-a'. Suppose it is desired to treat this as '=- a' but print a message. A rule might be

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as '=-'. Alternatively it might be desired to treat this as '= -a'. To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]      {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```


performs the other interpretation. Note that the expressions for the two cases might more easily be written:

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of '=-3', however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, `lex` also permits access to the I/O routines it uses. They are:

1. `input()` which returns the next input character;
2. `output(c)` which writes the character c on the output; and
3. `unput(c)` pushes the character c back onto the input stream to be read later by `input()`.

By default these routines are provided as macro definitions, but the programmer can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to transmit input or output to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by `input` must mean end of file; and the relationship between `unput` and `input` must be retained or the `lex` lookahead will not work. `lex` does not look ahead at all if it does not have to, but every rule ending in `+ * ?` or `$` or containing `/` implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See section 9.10 for a discussion of the character set used by `lex`. The standard `lex` library imposes a 100-character limit on backup.

Another `lex` library routine that the programmer will sometimes want to redefine is `yywrap()` which is called whenever `lex` reaches an end-of-file. If `yywrap` returns a 1, `lex` continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the programmer should provide a `yywrap` which arranges for new input and returns 0. This instructs `lex` to continue processing. The default `yywrap` always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through `yywrap`.
In fact, unless a private version of `input()` is supplied a file containing nulls cannot be handled, since a value of 0 returned by `input` is taken to be end-of-file.

9.4. Ambiguous Source Rules

lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ... ;
[a-z]+  identifier action ... ;
```

to be given in that order. If the input is integers, it is taken as an identifier, because [a-z]+ matches 8 characters, while integer matches only 7. If the input is integer, both rules match 7 characters, and the keyword rule is selected because it was given *first*. Anything shorter (for example, int) will not match the expression integer, and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression matches

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^'\n]*'
```

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the . operator does not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like [.\n]+ or equivalents; the lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both she and he in an input text. Some lex rules to do this might be

```
she     s++;
he      h++;
\n      |
.       ;
```


where the last two rules ignore everything besides he and she. Remember that '.' does not include newline. Since she includes he, lex will normally not recognize the instances of he included in she, since once it has passed a she those characters are gone.

Sometimes the programmer would like to override this choice. The action REJECT means 'go do the next alternative.' It executes whatever rule was second choice after the current rule. The position of the input pointer is adjusted accordingly. Suppose the programmer really wants to count the included instances of he:

```
she     {s++; REJECT;}
he      {h++; REJECT;}
\n      |
.       ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression is then counted. In this example, of course, the programmer could note that she includes he but not vice versa, and omit the REJECT action on he; in other cases, however, it would not be possible *a priori* to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+  { ... ; REJECT;}
a[cd]+  { ... ; REJECT;}
```

If the input is ab, only the first rule matches, and on ad only the second matches. The input string accb matches the first rule for four characters and then the second rule for three characters. In contrast, the input accd agrees with the second rule for four characters and the first rule for three.

In general, REJECT is useful whenever the purpose of lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word the is considered to contain both th and he. Assuming a two-dimensional array named digram to be incremented, the appropriate source is shown below.

```
%%
[a-z][a-z]   {digram[yytext[0]][yytext[1]]++; REJECT;}
.            ;
\n           ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

9.5. lex Source Definitions

Remember the format of the lex source:

```
{definitions}
%%
{rules}
%%
{programmer routines}
```

So far only the rules have been described. The programmer needs additional options, though, to define variables for use in his program and for use by lex. These can go either in the definitions section or in the rules section.

Remember that lex is turning the rules into a program. Any source not intercepted by lex is copied into the generated program. There are three classes of such things.

1. Any line which is not part of a lex rule or action which begins with a blank or tab is copied into the lex-generated program. Such source input prior to the first %% delimiter is external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by lex which contains the actions. This material must look like program fragments, and should precede the first lex rule.

   As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only the delimiters %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the lex output.

Definitions intended for lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define lex substitution strings. The format of such lines is

```
name translation
```

and it associates the string given as a translation with the name. The name and

translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be invoked by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D            [0-9]
E            [DEde][-+]?{D}+
%%
{D}+         printf("integer");
{D}+"."{D}*({E})?        |
{D}*"."{D}+({E})?        |
{D}+{E}              printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as 35.EQ.I, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within lex itself for larger source programs. These possibilities are discussed below under section 9.11 — *Summary of Source Format.*

9.6. Using lex

There are two steps in compiling a lex source program. First, the lex source must be turned into a generated program in the host general-purpose language. Then this program must be compiled and loaded, usually with a library of lex subroutines. The generated program is on a file named lex.yy.c. The I/O library is defined in terms of the C standard library in section 3 of the *SunOS Reference Manual.*

The lex library is accessed by the loader flag -ll.
So an appropriate set of commands is:

```
tutorial% lex source
tutorial% cc lex.yy.c -ll
```

The resulting program is placed on the usual file a.out for later execution. To use lex with yacc see below. Although the default lex I/O routines use the C standard library, the lex automata themselves do not do so; if private versions of input, output, and unput are given, the library can be avoided. lex has several options which are described in the lex(1) manual page.

9.7. lex and yacc

If you want to use `lex` with `yacc`, note that what `lex` writes is a program named `yylex()`, the name required by `yacc` for its analyzer. Normally, the default main program in the `lex` library calls this routine, but if `yacc` is loaded, and its main program is used, `yacc` calls `yylex()`.

In this case each `lex` rule should end with

```
return(token);
```

to return the appropriate token value.

An easy way to get access to `yacc`'s names for tokens is to compile the `lex` output file as part of the `yacc` output file by placing the line

```
# include "lex.yy.c"
```

in the last section of `yacc` input. Supposing the grammar to be named 'good' and the lexical rules to be named 'better' the command sequence can just be:

```
tutorial% yacc good
tutorial% lex better
tutorial% cc y.tab.c -ll
tutorial%
```

The `lex` and `yacc` programs can be generated in either order.

9.8. Examples

As a trivial problem, consider copying an input file while adding 3 to every non-negative number divisible by 7. Here is a suitable `lex` source program

```
%%
        int k;
[0-9]+  {
        k = atoi(yytext);
        if (k%7 == 0)
            printf("%d", k+3);
        else
            printf("%d",k);
        }
```

to do just that. The rule `[0-9]+` recognizes strings of digits; `atoi()` converts the digits to binary and stores the result in `k`.
The operator `%` (remainder) is used to check whether `k` is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as `49.63` or `X7`. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as shown below.


```
%%
        int k;
-?[0-9]+{
        k = atoi(yytext);
        printf("%d", k%7 == 0 ? k+3 : k);
        }
-?[0-9.]+       ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a '.' or preceded by a letter are picked up by one of the last two rules, and not changed. The `if-else` has been replaced by a C conditional expression to save space; the form `a?b:c` means 'if a then b else c'.

For an example of statistics gathering, here is a program which constructs a histogram of the lengths of words, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
l s.
yywrap()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement `return(1);` indicates that `lex` is to perform wrapup. If `yywrap` returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a `yywrap` that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double-precision FORTRAN to single-precision FORTRAN. Because FORTRAN does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
...
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes `double precision` to `real`, or DOUBLE PRECISION to REAL.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n}  {
     printf(yytext[0]=='d'? "real" : "REAL");
     }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
"$"     "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as 'beginning of line, then five blanks, then anything but blank or zero.' Note the two different meanings of `^`. There follow some rules to change double-precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      |
{0-9}+{W}"."{W}{d}{W}[+-]?{W}[0-9]+      |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+      {
     /* convert constants */
     for(p=yytext; *p != 0; p++)
          {
          if (*p == 'd' || *p == 'D')
               *p=+ 'e'- 'd';
          ECHO;
          }
```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter d or D. The program then adds 'e'–'d', which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial d. By using the array `yytext` the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}     |
{d}{c}{o}{s}     |
{d}{s}{q}{r}{t}  |
{d}{a}{t}{a}{n}  |
...
{d}{f}{l}{o}{a}{t}       printf("%s",yytext+1);
```


Another list of names must have initial d changed to initial a:

```
{d}{l}{o}{g}     |
{d}{l}{o}{g}10   |
{d}{m}{i}{n}1    |
{d}{m}{a}{x}1    {
          yytext[0] =+ 'a' - 'd';
          ECHO;
          }
```

And one routine must have initial d changed to initial r:

```
{d}l{m}{a}{c}{h}          {yytext[0] =+ 'r'  - 'd';
                    ECHO;
                    }
```

To avoid such names as `dsinx` being detected as instances of `dsin`, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*      |
[0-9]+    |
\n        |
.         ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

9.9. Left Context-Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The `^` operator, for example, is a prior context operator, recognizing immediately preceding left context just as `$` recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the programmer's action code; such a flag is the simplest way of dealing with the problem, since `lex` is not involved at all. It may be more convenient, however, to have `lex` remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It is only be recognized when `lex` is in that start condition. The current start condition may be changed at any time. Finally,

if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word `magic` to `first` on every line which begins with the letter a, changing `magic` to `second` on every line which begins with the letter b, and changing `magic` to `third` on every line which begins with the letter c. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
        int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to `lex` in the definitions section with a line reading

```
%Start  name1 name2 ...
```

where the conditions may be named in any order. The word `Start` may be abbreviated to `s` or `S`. The conditions may be referenced at the head of a rule with the <> brackets:

```
<name1>expression
```

is a rule which is only recognized when `lex` is in the start condition `name1`. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to `name1`. To resume the normal state,

```
BEGIN 0;
```

which resets to the initial condition of the `lex` automaton interpreter. A rule may be active in several start conditions:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic       printf("second");
<CC>magic       printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but `lex` does the work rather than the programmer's code.

9.10. Character Set

The programs generated by `lex` handle character I/O only through the routines `input`, `output`, and `unput`. Thus the character representation provided in these routines is accepted by `lex` and employed to return values in `yytext`. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter a is represented in the same form as the character constant 'a'.

If this interpretation is changed, by providing I/O routines which translate the characters, `lex` must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by two lines containing only '%T'. The table contains lines of the form

```
{integer} {character string}
```

which indicate the value associated with each character. Thus the next example

Figure 9-3 *Sample character table.*

```
%T
 1      Aa
 2      Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear

either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

9.11. Summary of Source Format

The general form of a `lex` source file is:

```
{definitions}
%%
{rules}
%%
{programmer subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form 'name space translation'.
2. Included code, in the form 'space code'.
3. Included code, in the form

```
%{
code
%}
```

4. Start condition declarations, given in the form

```
%S name1 name2 ...
```

5. Character set tables, in the form

```
%T
number space character-string
...
%T
```


6. Changes to internal array sizes, in the form

```
%x nnn
```

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Table 9-1 *Changing Internal Array Sizes in lex*

Letter	*Parameter*
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form 'expression action' where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in `lex` use the following operators:

Table 9-2 *Regular Expression Operators in lex*

Operator	*Meaning*
x	the character "x"
"x"	an "x", even if x is an operator
\x	an "x", even if x is an operator
[xy]	the character x or y
[x-z]	the characters x, y or z
[^x]	any character but x
.	any character but newline
^x	an x at the beginning of a line
<y>x	an x when lex is in start condition y
x$	an x at the end of a line followed by newline
x?	an optional x
x*	0,1,2, ... instances of x
x+	1,2,3, ... instances of x
x|y	an x or a y
(x)	an x
x/y	an x but only if followed by y
{xx}	the translation of xx from the definitions section
x{m,n}	*m* through *n* occurrences of x

9.12. Caveats and Bugs

There are pathological expressions which produce exponential growth of the tables when converted to deterministic automata; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT is executed, the programmer must not have used unput to change the characters forthcoming from the input stream. This is the only restriction on the programmer's ability to manipulate the not-yet-processed input.

10

yacc — Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an 'input language' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

yacc provides a general tool for describing the input to a computer program. The yacc programmer specifies the structure of the input, together with code to be invoked as each item is recognized. yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the programmer's application handled by this subroutine.

The input subroutine produced by yacc calls a programmer-supplied routine to return the next basic input item. Thus, the programmer can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The programmer-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

The class of specifications that yacc accepts is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, FORTRAN, APL, Pascal, Ratfor, etc., yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

yacc provides a general tool for imposing structure on the input to a computer program. The yacc programmer prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. yacc then generates a function to control the input process. This function, called a *parser*, calls the programmer-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then programmer code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

yacc generates its actions and output subroutines in C. Moreover, many of the syntactic conventions of yacc follow C.

The heart of the yacc input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma ',' is enclosed in single quotes — implying that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This routine reads the input stream, recognizing the lower-level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols are referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
...
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters such as ',' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be 'slipped in' to a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid yacc specifications for their input revealed errors of conception or design early in the program development.

The next several sections describe the basic process of preparing a yacc specification; Section 10.1 describes the preparation of grammar rules, Section 10.2 the preparation of the programmer-supplied actions associated with these rules, and Section 10.3 the preparation of lexical analyzers. Section 10.4 describes the operation of the parser. Section 10.5 discusses various reasons why yacc may be unable to produce a parser from a specification, and what to do about it. Section 10.6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 10.7 discusses error detection and recovery. Section 10.8 discusses the operating environment and special features of the parsers yacc produces. Section 10.9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10.10 discusses some advanced topics. Section 10.11 has a brief example, and section 10.12 gives a summary of the yacc input syntax. Section 10.13 gives an example using some of the more advanced features of yacc, and, finally, section 10.14 describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of yacc.

10.1. Basic Specifications

Names refer to either tokens or nonterminal symbols. yacc requires token names to be declared as such. In addition, for reasons discussed in Section 10.3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar) rules*, and *programs*. The sections are separated by double percent %% marks. The percent % is generally used in yacc specifications as an escape character.

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal yacc specification is

```
%%
rules
```

Spaces (also called blanks), tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal — they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot '.', underscore '_', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ''. As in C the backslash '\' is an escape character within literals, and all the C escapes are recognized:

```
'\n'    newline
'\r'    return
'\''    single quote '
'\\'    backslash '\'
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  'xxx' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar '|' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A       :       B  C  D  ;
A       :       E  F  ;
A       :       G  ;
```

can be given to yacc as

```
A       :       B  C  D
        |       E  F
        |       G
        ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :  ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1  name2 . . .
```

in the declarations section. See Sections 3, 5, and 6 for much more discussion. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this

symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the programmer-supplied lexical analyzer to return the endmarker when appropriate — see Section 10.3, below. Usually the endmarker represents some reasonably obvious I/O status, such as 'end-of-file' or 'end-of-record'.

10.2. Actions

With each grammar rule, the programmer may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces '{' and '}'. For example,

```
A       :       '('  B  ')'
                {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                {       printf("a message\n");
                        flag = 25;  }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign symbol '$' is used as a signal to yacc in this context.

To return a value, the action normally sets the pseudo-variable '$$' to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A       :       B  C  D  ;
```


for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :       '('  expr  ')'  ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :       '('  expr  ')'          {  $$ = $2 ;  }
```

By default, the value of a rule is the value of $1 (the first element in it). Thus, grammar rules of the form

```
A       :       B  ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual $ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A       :       B
                {  $$ = 1;  }
                C
                {  x = $2;   y = $3;  }
        ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. yacc actually treats the above example as if it had been written:

```
$ACT    :       /* empty */
                {  $$ = 1;  }
        ;
A       :       B  $ACT  C
                {  x = $2;   y = $3;  }
        ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to

construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. The parse tree can be built by supplying actions such as:

```
expr    :       expr  '+'  expr
                      {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The programmer may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks '%{' and '%}'. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making `variable` accessible to all of the actions. The `yacc` parser uses only names beginning in 'yy'; the programmer should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.10.

10.3. Lexical Analysis

The programmer must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called `yylex()`. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable `yylval()`.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by `yacc`, or chosen by the programmer. In either case, the '# define' mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name `DIGIT` has been defined in the declarations section of the `yacc` specification file. The relevant portion of the lexical analyzer might look like:


```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
        ...
        case '0':
        case '1':
        ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return the token number of `DIGIT`, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier `DIGIT` will be defined as the token number associated with the token `DIGIT`.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of `if` or `while` as token names will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name `error` is reserved for error handling, and should not be used naively (see Section 10.7).

As mentioned above, the token numbers may be chosen by `yacc` or by the programmer. In the default situation, the numbers are chosen by `yacc`. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the programmer; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *lex* program developed by Mike Lesk[8] and described in the previous chapter on `lex`. These lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications use regular expressions instead of grammar rules. `lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and

whose lexical analyzers must be crafted by hand.

10.4. How the Parser Works

yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by yacc consists of a finite-state machine with a stack. The parser can read and remember the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite-state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls `yylex()` to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

shift Action

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

reduce Action

The *reduce* action keeps the stack from growing without bound. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a '.') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.  reduce 18
```

refers to *grammar rule* 18, while the action


```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       : x  y  z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

which pushes state 20 onto the stack, and becomes the current state.

In effect, the reduce action 'turns back the clock' in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of programmer-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable `yylval()` is copied onto the value stack. After the return from the programmer's code, the reduction is carried out. When the *goto* action is done, the external variable `yyval()` is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

accept and error Actions

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The

parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 10.7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When yacc is invoked with the -v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:


```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :  rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :  sound_place

        DELL  shift 5
        .  error

        place  goto 4

state 3
        sound  :  DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme  :  sound  place_    (1)

        .  reduce  1

state 5
        place  :  DELL_    (3)

        .  reduce  3

state 6
        sound  :  DING  DONG_    (2)

        .  reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read, becoming the lookahead token. The action in state 0 on DING is 'shift 3', so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, DONG, is read, becoming the lookahead token. The action in state 3 on the token DONG is 'shift 6', so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3,

and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, DELL, must be read. The action is 'shift 5', so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by '$end' in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, and so on. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

10.5. Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr : expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not unambiguously specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

The first is called *left association*, the second *right association*.

yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an

input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left-associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any 'shift/shift' conflicts.

When there are shift/reduce or reduce/reduce conflicts, yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the programmer rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than yacc can construct. The use of actions within rules can also cause conflicts, if the action must

be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an 'if-then-else' construction:

```
stat    :    IF '(' cond ')' stat
        |    IF '(' cond ')' stat ELSE stat
        ;
```

In these rules, IF and ELSE are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if rule*, and the second the *if-else rule*.

These two rules form an ambiguous construction, since input of the form:

```
IF ( condition-1 ) IF ( condition-2 ) statement-1 ELSE statement-2
```

can be structured according to these rules in two ways:

```
IF ( condition-1 ) (
        IF ( condition-2 ) statement-1
)
ELSE statement-2
```

or

```
IF ( condition-1 ) (
        IF ( condition-2 ) statement-1
        ELSE statement-2
)
```

The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding 'un-*ELSE*'d' IF. In this example, consider the situation where the parser has seen

```
IF ( condition-1 ) IF ( condition-2 ) statement-1
```

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get


```
IF ( condition-1 ) stat
```

and then read the remaining input,

```
ELSE statement-2
```

and reduce

```
IF ( condition-1 ) stat ELSE statement-2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, *statement*-2 read, and then the right hand portion of

```
IF ( condition-1 ) IF ( condition-2 ) statement-1 ELSE statement-2
```

can be reduced by the if-else rule to get

```
IF ( condition-1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

```
IF ( condition-1 ) IF ( condition-2 ) statement-1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (–v) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

        stat : IF ( cond ) stat_          (18)
        stat : IF ( cond ) stat_ELSE stat

        ELSE    shift 45
        .       reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the ELSE will have been shifted in this state. Back in state 23, the alternative action, described by '.', is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not ELSE, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following 'shift' commands refer to other states, while the numbers following 'reduce' commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most states, there will be at most one reduce action possible in the state, and this will be the default command. Programmers who encounter unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the programmer might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references cited in Chapter 1 might be consulted.

10.6. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the programmer specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a yacc keyword: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,


```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left-associative, and have lower precedence than star and slash, which are also left-associative. The keyword `%right` is used to describe right-associative operators, and the keyword `%nonassoc` is used to describe operators, like the `.LT.` operator in FORTRAN, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword `%nonassoc` in yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '=' expr
        |       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword `%prec` changes the precedence level associated with a particular grammar rule. `%prec` appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It changes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left  '+'  '-'
%left  '*'  '/'

%%

expr    :       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       '-'  expr       %prec  '*'
        |       NAME
        ;
```

A token declared by `%left`, `%right`, and `%nonassoc` need not be, but may be, declared by `%token` as well.

The precedences and associativities are used by `yacc` to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left-associative implies reduce, right-associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by `yacc`. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially 'cookbook' fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

10.7. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser 'restarted' after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the programmer some control over this process, `yacc` provides a simple, but reasonably general, feature. The token name 'error' is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token 'error' is legal. It then behaves as if the token 'error' were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :       error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :       error   ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any 'cleanup' action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :       error   '\n'  {  printf( "Reenter last line: " );  }  input
                        {       $$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :       error   '\n'
                        {       yyerrok;
                                printf( "Reenter last line: " );    }
                input
                        {       $$  =  $4;  }
        ;
```

As mentioned above, the token seen immediately after the 'error' symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the programmer, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by `yylex()` would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                        {       resynch();
                                yyerrok ;
                                yyclearin ;     }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the programmer can get control to deal with the error actions required by other portions of the program.

10.8. The yacc Environment

When the programmer inputs a specification to `yacc`, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the name may differ from installation to installation). `yacc` produces an integer-valued function called `yyparse()`. When `yyparse()` is called, it in turn repeatedly calls `yylex()` — the lexical analyzer supplied by the programmer (see Section 10.3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) `yyparse()` returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, `yyparse()` returns the value 0.

The programmer must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called `main` must be defined, that eventually calls `yyparse()`. In addition, a routine called `yyerror()` prints a message when a syntax error is detected.

The programmer must supply these two routines in one form or another. They can be as simple as the following example, or they can be as complex as needed.

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to `yyerror()` is a string containing an error message, usually the string 'syntax error'. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable `yychar` contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics.

The external integer variable `yydebug` is normally set to 0. If it is set to a nonzero value, the parser generates a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

10.9. Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

1. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of 'knowing who to blame when things go wrong.'
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be added easily.
5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in section 10.11 is written following this style, as are the examples in the text of this paper (where space permits). The programmer must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the yacc parser encourages so called 'left-recursive' grammar rules: rules of the form

```
name    :       name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list  ','  item
        ;
```

and

```
seq     :       item
        |       seq  item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right-recursive rules, such as

```
seq     :       item
        |       item  seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the programmer should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
        other declarations
%%
prog    :       decls  stats
        ;
decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;
stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;
        other rules
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except* for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single-token exception does not affect the lexical scan.

This kind of 'backdoor' approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the programmer to use words like 'if', which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of `yacc`; it is difficult to pass information to the lexical analyzer telling it 'this instance of `if` is a keyword, and that instance is a variable'. The programmer can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

10.10. Advanced Topics

This section discusses a number of advanced features of `yacc`.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT makes `yyparse` return the value 0; YYERROR makes the parser behave as if the current input symbol results in a syntax error; `yyerror()` is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :   adj  noun  verb  adj  noun
            {  look at the sentence . . .  }
        ;

adj :   THE     {   $$ = THE;  }
    |   YOUNG   {   $$ = YOUNG;  }
    . . .
    ;

noun    :   DOG
            {   $$ = DOG;  }
    |   CRONE
            {   if( $0 == YOUNG ){
                    printf( "what?\n" );
                    }
                $$ = CRONE;
                }
    ;
    . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. `yacc` can also support values of other types, including structures. In addition, `yacc` keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The `yacc` value stack (see Section 10.4) is declared to be a `union` of the various types of values desired. The programmer declares the union, and associates a union member name to each token and nonterminal symbol having a value. When the value is referenced through a `$$` or `$n` construction, `yacc` automatically inserts the appropriate union name, so that no unwanted conversions will take place. In addition, type-checking commands such as `lint(1)` will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the programmer since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where `yacc` cannot easily determine the type.

To declare the union, the programmer includes in the declaration section:

```
%union  {
        body of union ...
        }
```

This declares the `yacc` value stack, and the external variables `yylval` and `yyval`, to have type equal to this union. If `yacc` was invoked with the `-d` option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable `YYSTYPE` to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of `%{` and `%}`.

Once `YYSTYPE` is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords `%token`, `%left`, `%right`, and `%nonassoc`, the union member name is associated with the tokens listed. Thus, saying

```
%left  <optype>  '+'  '-'
```

will tag any reference to values returned by these two tokens with the union member name *optype*. Another keyword, `%type`, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left-context values (such as $0 — see the previous subsection) leaves yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :   aaa  {  $<intval>$  =  3;  } bbb
            {   fun( $<intval>2, $<other>0 );  }
    ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in 10.13. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the yacc value stack is used to hold int's, as was true historically. This paper is reprinted in this manual.

10.11. A Simple Example

This example gives the complete yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled 'a' through 'z', and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line-by-line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>
int  regs[26];
int  base;
%}
%start  list
%token  DIGIT  LETTER
%left  '|'
%left  '&'
```



```
%left  '+'  '-'
%left  '*'  '/'  '%'
%left UMINUS   /* supplies precedence for unary minus */
%%   /* beginning of rules section */
list    :       /* empty */
        |       list  stat  '\n'
        |       list  error  '\n'
                        {       yyerrok;  }
        ;
stat    :       expr
                        {       printf( "%d\n", $1 );  }
        |       LETTER  '='  expr
                        {       regs[$1]  =  $3;  }
        ;
expr    :       '('  expr  ')'
                        {       $$  =  $2;  }
        |       expr  '+'  expr
                        {       $$  =  $1  +  $3;  }
        |       expr  '-'  expr
                        {       $$  =  $1  -  $3;  }
        |       expr  '*'  expr
                        {       $$  =  $1  *  $3;  }
        |       expr  '/'  expr
                        {       $$  =  $1  /  $3;  }
        |       expr  '%'  expr
                        {       $$  =  $1  %  $3;  }
        |       expr  '&'  expr
                        {       $$  =  $1  &  $3;  }
        |       expr  '|'  expr
                        {       $$  =  $1  |  $3;  }
        |       '-'  expr       %prec  UMINUS
                        {       $$  =  -  $2;  }
        |       LETTER
                        {       $$  =  regs[$1];  }
        |       number
        ;
number  :       DIGIT
                        {       $$ = $1;    base  =  ($1==0)  ?  8  :  10;  }
        |       number  DIGIT
                        {       $$  =  base * $1  +  $2;  }
        ;
%%   /* start of programs */
yylex()         /* lexical analysis routine */
{
/* returns LETTER for lower case letter, yylval=0 thru 25 */
/* return DIGIT for digit, yylval=0 thru 9 */
/* all other characters are returned immediately */
        int     c;
        while((c = getchar()) == ' ') { /* skip blanks */ }
```

```
                                        /* c is now nonblank */
        if(islower(c))  {
                yylval = c - 'a';
                return(LETTER);
        }
        if(isdigit(c))  {
                yylval = c - '0';
                return(DIGIT);
        }
        return(c);
}
```

10.12. yacc Input Syntax

This section describes the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to yacc */

        /* basic entities */
%token  IDENTIFIER      /* includes identifiers and literals */
%token  C_IDENTIFIER    /* identifier (not literal) followed by : */
%token  NUMBER          /* [0-9]+ */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT  RIGHT  NONASSOC  TOKEN  PREC  TYPE  START  UNION

%token  MARK    /* the %% mark */
%token  LCURL   /* the %{ mark */
%token  RCURL   /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs  MARK  rules  tail
        ;

tail    :       MARK    {       In this action, eat up the rest of the file      }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs  def
        ;
```



```
def     :       START  IDENTIFIER
        |       UNION  (  Copy union definition to output  )
        |       LCURL  (  Copy C code to output file  )  RCURL
        |       defs  rword  tag  nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<'  IDENTIFIER  '>'
        ;

nlist   :       nmno
        |       nlist  nmno
        |       nlist  ','  nmno
        ;

nmno    :       IDENTIFIER      /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER   /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER  rbody  prec
        |       rules  rule
        ;

rule    :       C_IDENTIFIER  rbody  prec
        |       '|'  rbody  prec
        ;

rbody   :       /* empty */
        |       rbody  IDENTIFIER
        |       rbody  act
        ;

act     :       '{'  (  Copy action, translate $$, etc.  )  '}'
        ;

prec    :       /* empty */
        |       PREC  IDENTIFIER
        |       PREC  IDENTIFIER  act
        |       prec  ';'
        ;
```

10.13. An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in Section 10.10. The desk calculator example in section 10.11 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, 'a' through 'z'. Moreover, it also understands *intervals*, written

```
( x , y )
```

where *x* is less than or equal to *y*. There are 26 interval-valued variables 'A' through 'Z' that may also be used. The usage is similar to that in section 10.11 — assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of `yacc` and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, `INTERVAL`, by using `typedef`.
The `yacc` value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of `YYERROR` to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of `yacc` is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval-value. This causes a large number of conflicts when the grammar is run through `yacc`: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + ( 3.5 - 4. )
```

and

```
2.5 + ( 3.5 , 4. )
```

Notice that the 2.5 is to be used in an interval-valued expression in the second example, but this fact is not known until the ',' is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval-valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar-valued expressions scalar-valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more

normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{
#  include  <stdio.h>
#  include  <ctype.h>
typedef  struct  interval  {
         double  lo,  hi;
         }  INTERVAL;
INTERVAL  vmul(),  vdiv();
double  atof();
double  dreg[ 26 ];
INTERVAL  vreg[ 26 ];
%}
%start    lines
%union    {
         int  ival;
         double  dval;
         INTERVAL  vval;
         }
%token <ival> DREG VREG    /* indices into dreg, vreg arrays */
%token <dval> CONST                /* floating point constant */
%type <dval> dexp          /* expression */
%type <vval> vexp          /* interval expression */
         /* precedence information about the operators */
%left    '+'  '-'
%left    '*'  '/'
%left    UMINUS    /* precedence for unary minus */
%%
lines    :         /* empty */
         |         lines  line
         ;
line     :         dexp  '\n'
                              {         printf(  "%15.8f\n",  $1  );  }
         |         vexp  '\n'
                              {         printf(  "(%15.8f  ,  %15.8f  )\n",  $1.lo,  $1.hi  );  }
         |         DREG  '='  dexp  '\n'
                              {         dreg[$1]  =  $3;  }
         |         VREG  '='  vexp  '\n'
                              {         vreg[$1]  =  $3;  }
         |         error  '\n'
                              {         yyerrok;  }
```

```
        ;
dexp    :       CONST
        |       DREG
                        {       $$ = dreg[$1];  }
        |       dexp '+' dexp
                        {       $$ = $1 + $3;   }
        |       dexp '-' dexp
                        {       $$ = $1 - $3;   }
        |       dexp '*' dexp
                        {       $$ = $1 * $3;   }
        |       dexp '/' dexp
                        {       $$ = $1 / $3;   }
        |       '-' dexp        %prec UMINUS
                        {       $$ = -$2;       }
        |       '(' dexp ')'
                        {       $$ = $2;        }
        ;
vexp    :       dexp
                        {       $$.hi = $$.lo = $1;     }
        |       '(' dexp ',' dexp ')'
                        {
                        $$.lo = $2;
                        $$.hi = $4;
                        if( $$.lo > $$.hi ){
                                printf( "interval out of order\n" );
                                YYERROR;
                                }
                        }
        |       VREG
                        {       $$ = vreg[$1];          }
        |       vexp '+' vexp
                        {       $$.hi = $1.hi + $3.hi;
                                $$.lo = $1.lo + $3.lo;          }
        |       dexp '+' vexp
                        {       $$.hi = $1 + $3.hi;
                                $$.lo = $1 + $3.lo;             }
        |       vexp '-' vexp
                        {       $$.hi = $1.hi - $3.lo;
                                $$.lo = $1.lo - $3.hi;          }
        |       dexp '-' vexp
                        {       $$.hi = $1 - $3.lo;
                                $$.lo = $1 - $3.hi;             }
        |       vexp '*' vexp
                        {       $$ = vmul( $1.lo, $1.hi, $3 );  }
        |       dexp '*' vexp
                        {       $$ = vmul( $1, $1, $3 );        }
        |       vexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1.lo, $1.hi, $3 );  }
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 );        }
        |       '-' vexp        %prec UMINUS
                        {       $$.hi = -$2.lo;    $$.lo = -$2.hi;      }
        |       '(' vexp ')'
                        {       $$ = $2;        }
        ;
```



```
%%
# define BSZ 50  /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register  c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if(  isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if(  islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if(  isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char  buf[BSZ+1], *cp = buf;
                int  dot = 0,  exp = 0;

                for(  ;  (cp-buf)<BSZ  ;  ++cp,c=getchar()  ){

                        *cp = c;
                        if(  isdigit( c ) )  continue;
                        if(  c == '.' ){
                                if( dot++ || exp ) return( '.' );
                                        /* will cause syntax error */
                                continue;
                                }

                        if(  c == 'e' ){
                                if( exp++ ) return( 'e' );
                                        /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if(  (cp-buf) >= BSZ  )
                        printf( "constant too long: truncated\n" );
                else ungetc( c, stdin );  /* push back last char read */
                yylval.dval = atof( buf );
                return( CONST );
                }
        return( c );
        }
INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;    v.lo = b; }
        else { v.hi = b;    v.lo = a; }

        if( c>d ) {
                if( c>v.hi ) v.hi = c;
```

```
                if( d<v.lo ) v.lo = d;
                }
        else {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

10.14. Old Features Supported but not Encouraged

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes '"'.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with `yacc`, since it suggests that `yacc` is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash '\' may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

```
%< is the same as %left
%> is the same as %right
%binary and %2 are the same as %nonassoc
%0 and %term are the same as %token
%= is the same as %prec
```

5. Actions may also have the form

   `={ . . . }`

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Ox:
An Attribute-Grammar Compiling System
based on Yacc, Lex, and C:

Tutorial Introduction

November 5, 1993

bischoff@cs.iastate.edu

1 Introduction

Ox is an attribute-grammar compiling system based on Yacc, Lex, and C. Ox [1] generalizes the function of Yacc in the way that attribute grammars generalize context-free grammars. Ordinary Yacc and Lex specifications can be augmented with definitions of synthesized and inherited attributes written in C/C++ syntax. From these specifications, Ox generates a program that builds and decorates attributed parse trees. Ox accepts a most general class of attribute grammars. The user can specify parse-tree traversals for easy ordering of side effects such as code generation. Ox handles the tedious and error-prone details of writing code for parse-tree management, so its use eases problems of security and maintainability associated with that aspect of translator development.

Ox is a Yacc/Lex/C/C++ preprocessor, and is designed to bring attribute grammars to the mainstream of Unix-based language development. Ox inherits all of the familiar syntax and semantics of Yacc, Lex, and C/C++. This makes Ox easily accessible to language designers, developers, and experimenters who use those tools. It also provides a ready "escape hatch" in case it is desired to return to an ordinary Yacc implementation.

This paper gives an overview of Ox by emphasizing examples. It quickly familiarizes you with the Ox features that are most immediately useful. A more complete reference, the *Ox User Reference Manual*, accompanies the Ox electronic distribution, which can be obtained free by writing to `ox-request@cs.iastate.edu`.

[1] The name "Ox" comes from an attempt to pronounce an acronym for "An Attribute-Grammar Compiling System"

Familiarity with the use of Yacc, Lex, C, and Make is sufficient to understand this tutorial and to begin using Ox. Some prior exposure to attribute grammars is helpful. Readers with an urge for details and hands-on experience should use the index of the reference manual and should have access to a system on which Ox is installed. The examples herein (in machine-readable form) are included with the Ox distribution.

2 Converting a Yacc/Lex program for use with Ox

Probably the easiest way to get started with Ox is to convert an existing Yacc/Lex [2] parser or translator. This can usually be done without changing the Yacc and Lex code. Ox can also be used with Yacc-only translators, i.e., those with lexical analyzers hand-coded in C (see section 9.3).

2.1 A parser of arithmetic expressions

As a running example, we start with a Yacc/Lex parser for integer arithmetic expressions.

The Lex file is named `scan.l`, and specifies the tokens of the language as digit strings, parentheses, and four binary operators:

```
%{
#include "y.tab.h"
%}

%%
[ \n\t\f]+        ;
[0-9]+            return(ICONST);
[()*/+\-]         return(yytext[0]);
%%
```

The Yacc file (named `gram.y`) specifies the syntax. The grammar is disambiguated by use of the `%left` reserved word:

[2] Ox is designed to work also with Yacc and Lex workalikes and C++. Throughout this paper, "Yacc", "Lex", and "C" can generally be taken to mean "Yacc or Bison", "Lex or Flex", and "C or C++", respectively.

```
%token ICONST
%left '+' '-'
%left '*' '/'

%%
expr    :       expr    '*'     expr
        |       expr    '/'     expr
        |       expr    '+'     expr
        |       expr    '-'     expr
        |       '('     expr    ')'
        |       ICONST
        ;
%%
main()
  {return(yyparse());
  }
```

The following Make file is used to build and maintain the parser, which is named gc:

```
gc: y.tab.o lex.yy.o
        cc -o gc y.tab.o lex.yy.o -ly -ll

y.tab.c y.tab.h: gram.y
        yacc -d gram.y

lex.yy.c: scan.l
        lex scan.l

y.tab.o: y.tab.c
        cc -c y.tab.c

lex.yy.o: lex.yy.c y.tab.h
        cc -c lex.yy.c
```

2.2 A parser that builds a parse tree

The above parser does no semantic analysis. To get ready for Ox implementation of semantics, we need merely replace the following lines in the Make file:


```
y.tab.c y.tab.h: gram.y
        yacc -d gram.y

lex.yy.c: scan.l
        lex scan.l
```

with these:

```
oxout.y oxout.l: gram.y scan.l
        ox gram.y scan.l

y.tab.c y.tab.h: oxout.y
        yacc -d oxout.y

lex.yy.c: oxout.l
        lex oxout.l
```

The command:

```
ox gram.y scan.l
```

transforms `gram.y` (called the *Y-file*) into `oxout.y`, and transforms `scan.l` (called the *L-file*) into `oxout.l`. These Ox outputs replace `gram.y` and `scan.l` in the remaining steps of parser construction.

The user-observed behaviors of the original program and the one preprocessed by Ox are the same. The difference is that the version made using Ox and the new Make file builds a dummy (attribute-less) parse tree, while the original builds no parse tree. The original code in the example lacks Yacc actions, but had it contained such actions, their effects would have been undisturbed by the Ox preprocessing.

Having modified our Make file, we are ready to augment the Y-file and L-file with Ox constructs.

3 Adding Ox-generated semantics

This section introduces the form and meaning of Ox-specific constructs, by way of converting our parse-tree-building parser into a calculator.

Each parse tree has leaves labeled by the `ICONST` token. Let us endow this token with an attribute `string`: a character pointer that for each `ICONST` node is to point to a copy of the lexeme corresponding to the node. This is done by placing the *attribute declaration*:

```
@attributes {char *string;} ICONST
```

before the first %% mark in the Y-file. The above-mentioned storage location created for each ICONST node is called an *attribute instance* (concisely: an *instance*). It is an instance of the string attribute of ICONST.

We supply a C macro (named lexeme) that constructs a copy of the lexeme. For brevity of the example, lexeme unsafely neglects to check for return of NULL by malloc. Here is the modified L-file:

```
%{
#include "y.tab.h"
#include <string.h>

#define lexeme strcpy((char *)malloc(yyleng+1),yytext)
%}

%%
[ \n\t\f]+        ;
[0-9]+            return(ICONST); @{ @ICONST.string@ = lexeme; @}
"("               return('('); .
")"               return(')');
"*"               return('*');
"/"               return('/');
"+"               return('+');
"-"               return('-');
%%
```

To the right of the lexical rule for ICONST, there is between @{ and @} an *attribute definition* that causes the string attribute instance in each ICONST node to get a pointer to a copy of the constant's lexeme.

Notice that we have replaced the single lexical rule:

```
[()*/+\-]          return(yytext[0]);
```

with six rules that are together equivalent to that single rule. Ox would have been unable to determine from the object of the single return statement (namely yytext[0]) the specific token that would be returned. By replacing the rule, we make the returned tokens explicit, and avoid a warning from Ox.

Each parse-tree node labeled by expr is the root of a subtree corresponding to a subexpression. Placing the attribute declaration:


```
@attributes {long val;} expr
```

in the Y-file causes the Ox-generated translator to allocate space (an attribute instance) for a long named val each time it creates a node labeled by expr.

The *body* (the part between curly braces) of an attribute declaration resembles that of a C structure declaration, except that curly braces cannot be nested.[3]

The definitions for the val attribute of expr are seen in the *attribute reference sections* (code fragments delimited by @{ and @}) in the modified Y-file. Each of the attribute definitions starts with the *implicit-mode annunciator* @i, whose meaning is explained in section 4.1. In this example, each attribute reference section contains exactly one attribute definition.

[3]Attributes can be of any C fundamental or derived type. The Ox code in section 8 uses an attribute that is a C structure.

```
%token ICONST
%left '+' '-'
%left '*' '/'

@attributes {char *string; } ICONST
@attributes {long val;}      expr

%%
expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;  @}

        |       expr    '/'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ / @expr.2.val@;  @}

        |       expr    '+'     expr
                @{ @i @expr.val@ = @expr.1.val@ + @expr.2.val@;    @}

        |       expr    '-'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ - @expr.2.val@;  @}

        |       '('     expr    ')'
                @{ @i @expr.0.val@ = @expr.1.val@;                 @}

        |       ICONST
                @{ @i @expr.val@ = atoi(@ICONST.string@);          @}
        ;
%%
main()
  {return(yyparse());
  }
```

The grammar symbol expr has three *grammar-symbol occurrences* (namely `expr.0`, `expr.1`, and `expr.2`) in the grammar rule:

```
expr    :       expr    '*'     expr
```

An *attribute occurrence* (concisely: an *occurrence*) is a grammar-symbol occurrence together with an attribute of the symbol. An *attribute reference* takes the form:

@grammarsymbol.[*integer.*]*attributename@*

where *attributename* appears as an identifier in the body of the attribute

declaration for *grammarsymbol*. If integer is *n*, the reference is to the *n*th occurrence of *grammarsymbol* counting from the left of the rule (the leftmost occurrence being the 0th). The square brackets above denote that *integer* and the second . are optional (the default value for *integer* being 0).

Attribute definitions are basically C code fragments containing attribute references. In general, an attribute definition section contains zero to many attribute definitions. Each attribute definition is announced by a mode annunciator, and terminated by @} or by the next mode annunciator.

4 Order of Attribute-Instance Evaluation

Attribute grammars specify semantics in a *declarative* or *functional* (rather than sequential or imperative) style. When a parse tree is created, the tree's attribute instances are evaluated in an order constrained (but not fully determined) by the attribute grammar. It is clear that in the example of section 3, all `val` instances in the leaf nodes must be evaluated before the `val` instance of the root node.

Ox (rather than the compiler designer) generates code that causes instances to be evaluated in a correct order.

4.1 Dependency relations in the Y-file

There is a constraint for each grammar rule in the Y-file: a *dependency relation* on the attribute occurrences in that rule. For the rule:

```
expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;
                @}
```

there is the constraint that instances corresponding to `expr.1.val` and `expr.2.val` in sibling parse-tree nodes must be evaluated before the one corresponding to `expr.0.val` in their parent node.

Each rule's dependency relation is determined by its individual attribute definitions. There are several modes for communicating dependency information to Ox. The *implicit mode* is, for most Ox translators, the only such mode needed. The *explicit mode* is described briefly in section 9.7.

The implicit-mode annunciator @i (see the example in section 3) signals to Ox the beginning of an attribute definition. Further, it informs Ox that

an instance corresponding to the definition's *leftmost* attribute reference is to be evaluated *after* those corresponding to other attribute references in the definition. This is to say that the occurrence corresponding to the leftmost reference *depends on* the occurrences corresponding to the other references in the definition.

4.2 Dependency relations in the L-file

Note that the mode annunciator `@i` does not appear in the L-file of the example in section 3. Mode annunciators are not used in L-files. An attribute reference section in an L-file is executed as a whole whenever the corresponding lexical rule is matched. In the example, this is done whenever the Lex-generated scanner matches a digit string. Executing an attribute reference section may involve the evaluation of several attribute instances. An attribute reference section in the L-file must contain exactly one attribute reference for each attribute occurrence defined there (in the previous example, that for `ICONST.string`).

5 Using global variables

Attribute reference sections can contain any C code, including references to global variables.

In our running example, we haven't yet shown how to print the main result of the semantic analysis (i.e., the value of the expression). The approach is to copy the `val` attribute instance of the root node into a global variable, then print it after termination of `yyparse()`. We introduce a unique start production for this purpose. The L-file need not be changed. Here is shown the new Y-file, with changed or added lines marked by empty C comments:


```
%token ICONST
%left '+' '-'
%left '*' '/'

%{                                                              /* */
long globVal;                                                   /* */
%}                                                              /* */

@attributes {char *string;} ICONST
@attributes {long val;}     s expr                              /* */
%%
s       :       expr                                            /* */
                @{ @i globVal = @s.val@ = @expr.val@;       @}  /* */
        ;                                                       /* */

expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;  @}

        |       expr    '/'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ / @expr.2.val@;  @}

        |       expr    '+'     expr
                @{ @i @expr.val@ = @expr.1.val@ + @expr.2.val@;    @}

        |       expr    '-'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ - @expr.2.val@;  @}

        |       '('     expr    ')'
                @{ @i @expr.0.val@ = @expr.1.val@;                 @}

        |       ICONST
                @{ @i @expr.val@ = atoi(@ICONST.string@);          @}
        ;
%%
main()
  {yyparse();                                                   /* */
   printf("%d\n",globVal);                                      /* */
  }
```

Upon completion of the call to `yyparse`, the tree's attribute instances have all been evaluated. The evaluation of `@s.val@` entails an assignment to `globVal`. The printing of `globVal` is the last thing done by the calculator.

6 Parse-tree traversals

A parse tree is much more useful if it can be traversed, and if its attribute instances can be accessed during traversals. Such traversals are particularly useful for code generation. Ox can be instructed to generate a translator that performs various kinds of traversals after evaluation of all of the tree's attribute instances.

6.1 Application: translation to prefix

The following Y-file specifies an expression parser that translates its (infix) input to prefix form. The L-file is the same as that of the previous example.


```
%token ICONST
%left '+' '-'
%left '*' '/'

@traversal @preorder yourTrav
@traversal @preorder yoursToo

@attributes {char *string;} ICONST
@attributes {long val;}     s expr
%%
s       :       expr
                @{ @i @s.val@ = @expr.val@;
                   @yoursToo printf("\n%d\n",@s.val@);
                @}
        ;
expr    :       expr    '*'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ * @expr.2.val@;
                   @yourTrav printf(" * ");
                @}
        |       expr    '/'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ / @expr.2.val@;
                   @yourTrav printf(" / ");
                @}
        |       expr    '+'     expr
                @{ @i @expr.val@ = @expr.1.val@ + @expr.2.val@;
                   @yourTrav printf(" + ");
                @}
        |       expr    '-'     expr
                @{ @i @expr.0.val@ = @expr.1.val@ - @expr.2.val@;
                   @yourTrav printf(" - ");
                @}
        |       '('     expr    ')'
                @{ @i @expr.0.val@ = @expr.1.val@;
                @}
        |       ICONST
                @{ @i @expr.val@ = atoi(@ICONST.string@);
                   @yourTrav printf(" %s ",@ICONST.string@);
                @}
        ;
%%
main()
  {return(yyparse());
  }
```

The line

```
@traversal @preorder yourTrav
```

declares a left-to-right preorder traversal named `yourTrav`. Suppose that in our example, the `yourTrav` traversal has reached a node at which a grammar rule R is applied. If the attribute reference section of R contains the *traversal-mode annunciator* `@yourTrav` (which was given meaning by its `@traversal` declaration), then the `printf` statment following `@yourTrav` is executed, and the traversal is continued for the subtree rooted at the node in question. Using `@postorder` instead of `@preorder` would cause a traversal that executes the `printf` *after* completing the traversal of that subtree, resulting in a postfix translation.

A traversal that accesses the `val` instance in the root node is an alternative to using the global variable `globVal` of section 5. Placing the line:

```
@traversal @preorder yoursToo
```

in the declarations section, and the line:

```
@yoursToo printf("%d\n",@$.val@);
```

in the attribute reference section for the start production accomplishes the same thing as the use of `globVal`.

One traversal is done for each traversal declaration, the traversals being done one after another, in the order in which the declarations appear. In the example, the declaration of `yoursToo` appears after that of `yourTrav`, so the value of the expression is printed after the preorder translation is printed.

7 Inherited vs. Synthesized Attributes

It is useful to think of the lexical rules (i.e., the rules in the L-file) as virtual grammar rules (productions) whose right-hand sides are the empty string and whose left-hand sides, while actual Yacc tokens, are virtual nonterminals. This generic concept of *rule* is consistent with usual concepts of *attribute grammar*, and leads to the following definitions:

An attribute occurrence o in a rule R is ***synthesized*** if and only if

- o is on the LHS of R and the attribute reference section of R contains a definition of o, or
- o is on the RHS of R and the attribute reference section of R contains no definition of o.

An attribute occurrence o in a rule R is ***inherited*** if and only if

- o is on the left-hand side (LHS) of R and the attribute reference section of R contains no definition of o, or
- o is on the right-hand side (RHS) of R and the attribute reference section of R contains a definition of o.

Ox issues an error message if it finds an attribute that has both synthesized and inherited occurrences in the grammar. An attribute is *synthesized* if and only if it has at least one occurrence, and its every occurrence is synthesized. An attribute is *inherited* if and only if it has at least one occurrence, and its every occurrence is inherited. It follows from the above that the grammar's start symbol can have only synthesized attributes. Referring to returned tokens as rules emphasizes the equal status of tokens and nonterminals, inasmuch as each kind of symbol (except the start symbol) can have both synthesized and inherited attributes. Each symbol has a distinct name space, so same-named attributes of different symbols are distinct attributes, and can differ as to whether they are inherited or synthesized.

For each parse-tree node except the root node, two rules of the Ox input specification are of particular interest. The *home rule* is the rule applied at the node, i.e., the rule whose LHS is the label of the given node, and whose RHS symbols are the labels of the children of the node. The *parent rule* is the rule applied at the node's parent. The attribute definition of a synthesized attribute instance of a given node is associated with the node's home rule (i.e., it appears in the attribute reference section for that rule), and definitions of inherited attribute instances are similarly associated with the parent rule.

In a legal input specification, each attribute of a symbol appearing in a rule is either synthesized or inherited, but not both, so the definitions of all attributes "fit together" completely and without contradiction.

8 Using inherited attributes

This section gives an example indicating the use of inherited attributes for semantic analysis involving right context. The example also gives a better idea of how Ox code is used together with C code.

In many languages, for instance Pascal, each variable declaration is essentially a list of identifiers followed by a type specifier. Here we show a simple language whose every sentence consists of such a variable declaration. Our translator parses the input, recording in a symbol object the identifier and type of each variable declared. Then the symbol objects are printed during a postorder traversal.

Here is the L-file:

```
%{
#include "y.tab.h"
#include <string.h>

#define lexeme strcpy((char *)malloc(yyleng+1),yytext)
%}

%%
[ \n\t\f]+          ;
real                return(REAL);
integer             return(INT);
boolean             return(BOOL);
[a-zA-Z]+           return(IDENT);  @{ @IDENT.string@ = lexeme; @}
","                 return(',');
";"                 return(';');
":"                 return(':');
.                   {fprintf(stderr,"illegal character\n"); exit(-1);}
%%
```

The definitions in section 7 together with the following Y-file imply that:

- string is a synthesized attribute of IDENT.
- sym is an inherited attribute of IDENT.
- tMark is an inherited attribute of varList.
- varDecl has no attributes.


```
%token REAL INT BOOL IDENT

%{
#include <stdlib.h>
struct sym {char *str,*typeMark;};

struct sym *allocSym(cp,t)
  char *cp,*t;
  {struct sym *pSym;
   pSym = (struct sym *) malloc(sizeof (struct sym));
   pSym->str = cp; pSym->typeMark = t;
   return pSym;
  }
%}

@attributes {char *string; struct sym *sym; } IDENT
@attributes {char *tMark;                   } varList
@traversal @postorder myT /* my Traversal */

%%
varDecl   :       varList ':' REAL ';'
                  @{ @i @varList.tMark@ = "real";    @}

          |       varList ':' INT ';'
                  @{ @i @varList.tMark@ = "integer"; @}

          |       varList ':' BOOL ';'
                  @{ @i @varList.tMark@ = "boolean"; @}
          ;

varList   :       IDENT
                  @{ @i @IDENT.sym@ =
                         allocSym(@IDENT.string@,@varList.tMark@);
                     @myT printf("%s: %s;\n",@IDENT.sym@->typeMark,
                                              @IDENT.sym@->str);
                  @}
          |       varList ',' IDENT
                  @{ @i @varList.1.tMark@ = @varList.tMark@;
                     @i @IDENT.sym@ =
                          allocSym(@IDENT.string@,@varList.tMark@);
                     @myT printf("%s: %s;\n",@IDENT.sym@->typeMark,
                                              @IDENT.sym@->str);
                  @}
          ;
```

```
%%
main()
  {return(yyparse()); }
```

9 Overview of other features

This section briefly describes some Ox features that are provided for convenience or for advanced or specialized use. Detailed descriptions of these features appear in the *Ox User Reference Manual*.

9.1 Macro facility

Ox's input specification may be such that the same or similar text appears in more than one place in attribute reference sections. Ox has a macro substitution feature that can be used to decrease verbosity in such cases.

9.2 Automatic generation of copy rules

Often a Y-file has attribute definitions that function only to copy an instance belonging to one node to a like-named instance belonging to the node's parent or child. Large attribute grammars tend to have many such definitions, which are sometimes called *copy rules*. The situation is conspicuous when contextual information is moved leafward via inherited attributes. Ox syntax provides ways of specifying that a copy rule is global to the attribute grammar, obviating repetition of attribute definitions in many grammar rules.

9.3 Using Ox with scanners not based on Lex

By default, Ox provides preprocessing for Lex files augmented with Ox constructs. By using a command line option, Ox can be informed that the L-file contains Ox-augmented C code rather than the usual Ox-augmented Lex code.

9.4 Use of multiple scanners

Some translators contain several scanners. Such a translator is designed so that at any moment, it is using one scanner or another, and switches to a

different one when there is a change in context. An Ox translator that uses more than one scanner can be constructed by submitting to Ox more than one L-file.

9.5 Stripping Ox constructs

Occasionally, the Ox user may desire copies of the Y-file and L-file(s) stripped of Ox-specific constructs. By a command-line option, the Ox user can filter all Ox-specific constructs from the inputs, to obtain files acceptable to Yacc and Lex. The original copies of the Y-file and L-file(s) are unchanged, but Ox's outputs on `oxout*.*` contain neither Ox constructs nor the usual Ox-generated parse-tree-management code.

9.6 Accessing Yacc pseudovariables

Attribute definitions that refer to the Yacc pseudovariables `$$`, `$1`, `$2`, etc. are permitted in various forms, including:

```
@i @grammarsymbol.[integer.]attributename@ = $n;
```

where `$n` denotes a Yacc pseudovariable. It is also possible to copy attribute instances into pseudovariables.

9.7 Expressing dependencies explicitly

Suppose that you have a C function `fun` in a library, and that you want to use it to define an attribute occurrence, say `sym.attrb`, in terms of some other occurrence `othersym.otherAttrb`. Further suppose that the first formal parameter of `fun` is of the same type as `othersym.otherAttrb`, and that `fun`'s second formal parameter is a pointer to something of the same type as `sym.attrb`. A call to `fun` changes the contents of the location indicated by its second argument.

It wouldn't work to write:

```
@i fun(@otherSym.otherAttrb@, &@sym.attrb@);
```

since the mode annuciator `@i` (see section 4.1) implies that the occurrence appearing first (`otherSym.otherAttrb`) is the occurrence being defined, and that it depends on `sym.attrb`. Actually you intend the opposite.

One solution would be to modify the definition of `fun` (reversing the order of its formal parameter list). If you don't want to disturb the library, however, it would be best to use Ox's *explicit mode annunciator* `@e` as follows:

```
@e sym.attrb : otherSym.otherAttrb ;
   fun(@otherSym.otherAttrb@, &@sym.attrb@);
```

In the first line above, Ox is explicitly given dependency information using a Make-like syntax: it is declared that `sym.attrb` depends on `otherSym.otherAttrb`. Use of the explicit mode makes the order of the occurrences in the second line's call to `fun` irrelevant to Ox's understanding of the dependencies.

9.8 Generating ANSI/ISO/C++ output

By default, the C code generated by Ox follows traditional C syntax. There is, however, a command-line option to produce code compatible with ANSI/ISO/C++ syntax. Thus it is easy to make Ox's code generation conform to the expectations of practically any C or C++ compiler.

10 Acknowledgements

This is to thank Terry Dineen, Carolyn Giberson, Markus Klingspor, John Levine, Carla Marceau, and Michael Seager for their helpful reviews of early versions of this paper.

burg, iburg und bfe

1 Einleitung

burg [FHP92] und iburg [FHP93] sind Programme die aus Baumgrammatiken Baumparser erzeugen. Die erzeugten Baumparser finden immer einen Parse mit minimalen Kosten und sind daher insbesondere für die Befehlsauswahl geeignet. Die von iburg erzeugten Parser verwenden dabei die im Vorlesungsskriptum beschriebene Methode, während die von burg erzeugten Parser als Baumautomaten implementiert und daher etwas schneller sind. Beide Baumparsergeneratoren akzeptieren die gleiche Eingabe und die erzeugten Parser finden gleich gute Ableitungen. Im Prinzip sind sie so entworfen, daß sie kompatipel miteinander sind. Leider ist diese Kompatibilität unter heutigen 64-bit Betriebsystemen nicht gegeben. Daher ist im Rest dieses Handbuchs praktisch ausschließlich von iburg die Rede, und Sie sollten auch iburg verwenden.

Der erzeugte Baumparser macht zwei Durchgänge durch den Baum: Im ersten Durchgang (Labeller) wird für jeden Knoten ein Zustand (bei iburg ähnlich dem Zustand eines endlichen Automaten) berechnet, anfangend mit den Blättern; im zweiten Durchgang (Reducer) wird ein optimaler Ableitungsbaum durchwandert, wobei gegebenenfalls entsprechende Aktionen durchgeführt werden können.

Die Eingabe von iburg ist auf maximale Flexibilität ausgelegt, und daher etwas unbequem. Sie ist eher dafür gedacht, als Zwischendarstellung für einen Codegeneratorgenerator zu dienen als direkt von Hand geschrieben zu werden. bfe ist ein front-end für iburg, das auf die in der Übung vorkommende Aufgabenstellung zugeschnitten ist und den Benutzern einige der Arbeiten abnimmt, die sie bei direkter Verwendung von iburg machen müßten.

```
%{
 enum { Assign=1, Constant, Fetch, Four, Mul, Plus };

 typedef struct tree {
         int op;
         struct tree *kids[2];
         STATEPTR_TYPE state;
 } *NODEPTR_TYPE;

 #define OP_LABEL(p)    ((p)->op)
 #define LEFT_CHILD(p)  ((p)->kids[0])
 #define RIGHT_CHILD(p) ((p)->kids[1])
 #define STATE_LABEL(p) ((p)->state)
 #define PANIC printf
 %}
 %start reg
 %term  Assign=1 Constant=2 Fetch=3 Four=4 Mul=5 Plus=6
 %%
 con:    Constant                = 1 (0);
 con:    Four                    = 2 (0);
 addr:   con                     = 3 (0);
 addr:   Plus(con,reg)           = 4 (0);
 addr:   Plus(con,Mul(Four,reg)) = 5 (0);
 reg:    Fetch(addr)             = 6 (1);
 reg:    Assign(addr,reg)        = 7 (1);
 %%
```

Abbildung 1: Eine Baumgrammatik

2 Der Aufbau der iburg-Eingabe

Burg erstellt aus einer Baumgrammatik als Eingabe einen Baum-Parser in C, der BURM genannt wird. Die Grammatik von iburg gleicht strukturell der von YACC. Kommentare werden wie in C geschrieben.

Das folgende Programmstück zeigt eine Beispielgrammatik, mit der eine sehr einfache Befehlsauswahl realisiert werden kann.

Abb. 2 zeigt eine EBNF-Gramatik für iburg-Eingaben, insbesondere für die Baumgrammatik. Kommentare, der Text zwischen "%{" und "%}" und der Text nach dem zweiten optionalen "%%" (User-Code) wird rein lexika-

```
grammar: {dcl} '%%' {rule} [ '%%' ]

dcl: '%start' Nonterminal
dcl: '%term' { Operator '=' ESN_Integer }

rule: Nonterminal ':' tree '=' ERN_Integer [ cost ] ';'

cost: /* empty */
cost: '(' Integer { ',' Integer } ')'

tree: Operator '(' tree ',' tree ')'
tree: Operator '(' tree ')'
tree: Operator
tree: Nonterminal
```

Abbildung 2: EBNF-Grammatik für iburg-Baumgrammatiken

lisch behandelt und daher in der EBNF-Grammatik nicht berücksichtigt. Nonterminale und Operatoren sind Identifier, wobei Operatoren vorher mit "`%term`" deklariert wurden. ERN steht für externe Regelnummer und ESN für externe Symbolnummer (s.u.).

Es folgt eine genauere Beschreibung der Einzelteile der iburg-Eingabe:

2.1 Die Configuration Section

Text zwischen "`%{`" und "`%}`" (sogenannte Configuration Sections) wird direkt in den erzeugten Baum-Parser kopiert. Da das der Scanner von iburg übernimmt, scheinen Configuration Sections nicht in Abb. 2 auf.

Üblicherweise enthalten Configuration Sections Deklarationen, Makros und/oder "`#include`"s, die der Baum-Parser braucht:

Der Typ `NODEPTR_TYPE` dekariert einen Zeiger auf einen Baumknoten. Der Baum-Parser greift auf den Operator eines Baumknoten `p` mit `OP_LABEL(p)` (ganzzahlig), und auf seine Kinder mit `LEFT_CHILD(p)` (`NODEPTR_TYPE`) und `RIGHT_CHILD(p)` (`NODEPTR_TYPE`) zu.

Der Baum-Parser berechnet und speichert einen Zustand in jedem Knoten des geparsten Baums. Er greift auf diesen Zustand über das Makro `STATE_LABEL(p)` (l-value vom Typ STATEPTR_TYPE) zu.

Bei Auftreten eines Fehlers wird `PANIC` als Routine im printf-Format aufgerufen.

2.2 Die Baumgrammatik

Nach der Configuration Section folgt die Baumgrammatik.

Eine Baum-Grammatik ähnelt einer kontextfreien Grammatik: sie besteht aus Regeln, Operatoren (Terminalen), Non-Terminalen und einem speziellen Start-Nonterminal. Im Unterschied zu normalen Grammatiken leiten Baumgrammatiken Bäume ab und nicht Strings (daher nennen wir die Terminale auch Operatoren). Die rechte Seite einer Regel, Baummuster genannt, ist ein Baum, der Nonterminale enthalten kann. Baummuster werden in geklammerter Prefix-Notation dargestellt. Regeln, deren Baummuster nur aus einem Nonterminal besteht, nennt man Kettenregeln.

2.2.1 Die Declaration Section

In der Declarations Section werden das Start-Nonterminal und die Operatoren (Terminale) der Bäume deklariert.

Alle Operatoren müssen deklariert werden, eine solche Deklarationszeile beginnt mit `%term`. Jeder Operator hat eine festgelgte Anzahl von Kindern (maximal zwei), die die Benutzer festlegen, indem sie den Operator entsprechend in Baummustern verwenden. Jedes Terminal hat eine eindeutigen externe Symbolnummer (ESN), einer positiven ganzen Zahl, definiert. `OP_LABEL(p)` muß diese externe Symbolnummer für den Knoten, auf den p zeigt, liefern.

Non-Terminale werden nicht deklariert, mit Ausnahme des Startsymboles, welches optional in einer Zeile, die mit "`%start`" beginnt, deklariert werden kann. Gibt es keine "`%start`"-Deklaration, verwendet `iburg` das Nonterminal der ersten Regel als Startsymbol.

2.2.2 Die Rules Section

Nach dem ersten "`%%`" beginnt die Rules Section, in der die Regeln der Baum-Grammatik definiert werden.

Eine Regel besteht aus einem Nonterminal auf der linken Seite, gefolgt von einem Doppelpunkt und einem Baummuster auf der rechten Seite, der externen Regelnummer (ERN) und einem Kostenvektor.

Eingebettete semantische Aktionen wie in YACC oder Attributierungen wie in Ox sind in `iburg` nicht erlaubt. Stattdessen wird jeder Regel eine sog. externe Regelnummer (ERN) zugeordnet. Im Reducer-Durchgang des Baumparsers können dann aufgrund dieser Regelnummer entsprechende Aktionen durchgeführt werden.

Am Ende einer Regel steht der Kostenvektor, ein Vektor von positiven, ganzzahligen Kosten, in Klammern und durch Kommas getrennt. Wird der Kostenvektor weggelassen, werden die Kosten als 0 angenommen. `iburg` berücksichtigt nur die ersten vier Elemente des Kostenvektors. Die Kosten einer Ableitung ergeben sich aus der Summe der Kosten der Regel selbst, und den Kosten aller Subregeln. Im Standardfall berücksichtigt `iburg` nur das erste Element des Kostenvektors, die sog. *Principal Costs*. Alternativen dazu können durch Aufrufoptionen von `iburg` ausgewählt werden.

Wenn kein Startsymbol deklariert wurde, verwendet `iburg` das Non-Terminal der ersten Regel als Startsymbol.

2.3 Anwendungscode

Der nach dem zweiten "%%" stehende Text wird direkt in den erzeugten Baumparser kopiert. Das macht wieder der Scanner von `iburg` und ist daher in Abb. 2 nicht zu sehen.

In diesem Abschnitt können z.B. Routinen wie der Reducer stehen.

3 Der Parser

Während im vorigen Kapitel mit der Baumgrammatik die statische Komponente eines `iburg`-Programmes beschrieben wurde, werden in diesem Kapitel die Funktionsweise und Realisierung des erzeugten Baumparsers erläutert.

Von den beiden Durchgängen des Labellers wird der erste (**Labeller**) auf Wunsch automatisch erzeugt. Die Benutzer müssen sich nur noch um den Aufruf kümmern.

Der zweite Durchlauf (**Reducer**) durchläuft den Ableitungsbaum, von der Wurzel ausgehend. Da das genaue Vorgehen hier von der Anwendung abhängt, stellt `iburg` nur einige Grundroutinen und Datenstrukturen zur Verfügung und überläßt dem Benutzer, einen Reducer zu schreiben. `bfe` (siehe Abschnitt 5) erzeugt Reducer, die für die Aufgabenstellungen in dieser Übung passen sollten.

Der von `iburg` generierte Code stellt Grundroutinen für Labeller und Reducer zur Verfügung, sodaß die Benutzer auf Wunsch z.B. selbst eine Labelling-Routine schreiben können. Außerdem generiert `iburg` einige Hilfsroutinen, die gebräuchliche Kombinationen der Grundroutinen beinhalten bzw. nützlich zum Debugging sind.

3.1 Der Labeller

Der von `iburg` generierte Labeller heißt `burm_label` und hat folgenden Typ:

```
extern STATEPTR_TYPE burm_label(NODEPTR_TYPE p);
```

Diese Routine führt das Labeln des gesamten Baumes, auf den `p` zeigt, durch, und liefert den Zustand (`STATE_LABEL(p)`) von `p` zurück. Bäume, die nicht abgeleitet werden können, erhalten den State 0.

Den Rest dieses Abschnitts brauchen sie nur lesen, wenn Sie selbst einen Labeller schreiben wollen.

Die Grundroutine `burm_state` beinhaltet zum Unterschied von `burm_label` keinen Code zum Durchlaufen des Baumes und zum Beschreiben der Felder. Sie kann zum Labeln verwendet werden, wenn der Programmierer eigenen Code zum Durchlaufen des Baumes ausführen will.

```
extern STATEPTR_TYPE burm_state(int op,
       STATEPTR_TYPE leftstate, STATEPTR_TYPE rightstate);
```

Die Parameter sind eine externe Symbolnummer für einen Knoten und die Labels für das linke und rechte Kind des Knotens. Sie liefert das State Label, das dem jeweiligen Knoten zuzuweisen ist. Bei unären Operatoren wird `rightstate` ignoriert, bei Blättern werden `left-` und `rightstate` ignoriert.

3.2 Der Reducer

Sie brauchen diesen Abschnitt nur zu lesen, wenn Sie selbst einen Reducer schreiben wollen, statt den von `bfe` erzeugten zu verwenden.

Die wichtigste Routine für den Reducer ist `burm_rule`. Sie hat folgende Form:

```
extern int burm_rule(STATEPTR_TYPE state, int goalnt);
```

Die Parameter sind `state`, der Zustand (`STATE_LABEL(p)`) des Baumes und `goalnt`, eine Zahl, die das Zielnonterminal identifiziert, aus dem der Baum abgeleitet werden soll. Die Zahl des Startnonterminals ist 1. `burm_rule` liefert die externe Regelnummer der Wurzelregel des Ableitungsbaums (also der ersten Regel der Ableitung). Wenn sich aus dem Zielnonterminal der durch `state` beschriebenen Baum nicht ableiten läßt (das ist normalerweise ein Bug), liefert `burm_rule` 0.

Wie kommt man nun zu den Zahlen für andere Nonterminale als das Start-Nonterminal? Dafür stellt der erzeugte Baumparser folgendes Array zur Verfügung:

```
extern short *burm_nts[] = { ... };
```

Dieses Array enthält für jede externe Regel einen Vektor mit den Zahlen für die Nonterminale im Baummuster der Regel. Die Nonterminale einer Regel werden dabei von links nach rechts durchnummeriert. Beispiel:

```
addr:   Plus(con,Mul(Four,reg)) = 5 (0);
```

Dann findet man die Zahl für `con` in `burm_nts[5][0]` und für `reg` in `burm_nts[5][1]`; 5 ist dabei die externe Regelnummer.

Das würde schon reichen, um einen Reducer zu schreiben. Etwas bequemer wird es durch:

```
extern NODEPTR_TYPE *burm_kids(NODEPTR_TYPE p, int eruleno,
                               NODEPTR_TYPE kids[]);
```

Die Parameter sind die Adresse `p` eines Baumes, eine externe Regelnummer, und ein leerer Vektor von Zeigern auf Bäume. Die Prozedur nimmt an, daß p mit der gegebenen Regel abgeleitet werden kann und füllt den Vektor mit den Unterbäumen von `p`, die den Nonterminalen im Baummuster der Regel entsprechen. `burm_kids` liefert `kids`. Für den Baum

```
Plus(Four,Mul(Four,Fetch(Constant)))
```

und die obige Regel würde `burm_kids` in `kids[0]` `Four` ablegen und in `kids[1]` `Fetch(Constant)` (in der Form eines Zeigers auf den Fetch-Knoten).

Der Beispielparser in Abb.3 zeigt wie Labeller und Reducer als User-Code implementiert werden können. Der Reducer gibt den Maschinencode für den Subject Tree aus. (Die Routine `burm_string` wird in Kap.3.3 erklärt.)

```
parse(NODEPTR_TYPE p)
{
  burm_label(p);  /* label the tree */
  reduce(p, 1);   /* and reduce it  */
}

reduce(NODEPTR_TYPE p, int goalnt)
{
  int eruleno;
  short *nts;
  NODEPTR_TYPE kids[100];
  int i;

  eruleno=burm_rule(STATE_LABEL(p), goalnt); /* matching rule number */
  nts=burm_nts[eruleno];                     /* subtree goal non-terminals */

  if (eruleno==0) {
    fprintf(stderr, "tree cannot be derived from start symbol");
    exit(1);
  }

  burm_kids (p, eruleno, kids); /* initialize subtree pointers     */
  for (i=0; nts[i]; i++)        /* traverse subtrees left-to-right */
    reduce(kids[i], nts[i]);    /* reduce kids                     */

#if DEBUG
  printf ("%s\n", burm_string[eruleno]);          /* display rule */
#endif

  switch (eruleno){
    case 1:
      printf(...);  /* machine-code for rule 1 */
      break;
    case 2:
      printf(...);  /* machine-code for rule 2 */
      break;
    case 3:
    .
    .
    .
  }
}
```

Abbildung 3: Ein Beispiel-Parser

3.3 Weitere Routinen im erzeugten Baumparser

Dieser Abschnitt ist nur für besonders Interessierte.

Mit den bisher behandelten Routinen können bereits vollwertige Parser implementiert werden. Für Diagnose-Zwecke und fortgeschrittene Programmierung ür stellt `iburg` dem Programmierer jedoch noch eine Reihe weiterer externer Funktionen und Makros zur Verfügung.

Für jedes Nonterminal **X** definiert `iburg` eine Preprozessor-Direktive in der Form `burm_X_NT`, die der Numerierung der Nonterminale entspricht. Außerdem wird ein Makro `burm_X_rule(a)` definiert, das dem Funktionsaufruf `burm_rule(a, X)` entspricht. Für die Grammatik in Abb. 1 definiert `iburg`

```
#define burm_reg_NT 1
#define burm_con_NT 2
#define burm_addr_NT 3
#define burm_reg_rule(a) ...
#define burm_con_rule(a) ...
#define burm_addr_rule(a) ...
```

Diese Definitionen sind nur im Code nach dem zweiten "%%" sichtbar.

Die `iburg`-Option -I erzeugt Funktionen zur Beschreibung der Eingabe, welche für Diagnostik-Zwecke nützlich sind: Die Vektoren

```
extern char *burm_opname[];
extern short burm_arity[];
```

beinhalten den Namen und die Anzahl der Kinder jedes Terminales. Der Index ist wieder die externe Symbolnummer des Terminales.

Die Vektoren

```
extern char *burm_string[];
extern short burm_cost[][4];
```

beinhalten den Text und den Kostenvektor für jede Regel. Der Index ist die externe Regelnummer der Regel.

Der 0-terminierte Vektor

```
extern char *burm_ntname[];
```

hat den Index von burm_X_NT und beinhaltet den Namen des Non-Terminals X.

Die Routinen

```
extern int burm_op_label(NODEPTR_TYPE p);
extern STATEPTR_TYPE burm_state_label(NODEPTR_TYPE p);
extern NODEPTR_TYPE burm_child(NODEPTR_TYPE p, int index);
```

sind die Funktionsversionen der Konfigurationsmakros.

`burm_child(p, 0)` entspricht `LEFT_CHILD(p)`,

`burm_child(p, 1)` entspricht `RIGHT_CHILD(p)`.

Ein Beispiel für deren Anwendung ist der Grammatik-unabhängige Ausdruck

`burm_opname[burm_op_label(p)]`,

der den textuellen Namen des Operators im Knoten, auf den p zeigt, liefert.

4 Aufrufparameter von iburg

iburg liest eine Baumgrammatik und generiert einen Baumparser in C. Der erzeugte Baumparser kann für sich selbst compiliert werden oder in ein anderes File inkludiert werden. Das Kommando

`iburg [Optionen] [inputfile [outputfile]]`

startet **iburg**. Wenn unter **inputfile** eine Datei angegeben wird, erwartet **iburg** seine Input-Grammatik von dort, andernfalls liest es vom Standard Input.

Optionen (Auszug)

`-p prefix`	exportierte Namen mit prefix beginnen (Default ist burm)
`-I`	Erzeugt die Funktionen burm_arity, burm_child, burm_cost, burm_ntname, burm_op_label, burm_opname, burm_state_label und burm_string
`-T`	Tracing (siehe man-Page)

5 BFE (iburg Front End)

Dieses Werkzeug erzeugt automatisch einen Reducer (ähnlich dem in Abb. 3). Die Eingabe ist eine abgewandelte **iburg**-Spezifikation: der Teil vor dem ersten "%%" wird 1:1 übernommen; der Regelteil ist jedoch hier zeilenorientiert, wobei die Zeilen folgende Form haben:

```
rule [ '#' Integer { ',' Integer } [ '#' Aktionen ] ]
```

Regeln werden wie in **iburg** geschrieben, die Integers stellen den Kostenvektor dar. Die Definition der externen Regelnummer fällt weg. Die Aktionen bestehen aus C-Code, den der erzeugte Reducer beim Matching mit der jeweiligen Regel ausführen soll; dabei werden die Aktionen der Blätter vor denen der Wurzel ausgeführt.[1]

Der Reducer durchläuft den Ableitungsbaum, nicht (direkt) den Eingabebaum. Allerdings wird dabei doch auf Knoten des Eingabebaums zugegriffen (insbesondere auf `STATE_LABEL(p)`), und zwar für jeden Knoten des Ableitungsbaumes auf den Wurzelknoten des zugehörigen Eingabebaums; d.h., Knoten des Eingabebaums können mehrmals vorkommen (im Fall von Kettenregeln), aber auch gar nicht.

In den Aktionen kann auf den entsprechenden Knoten des Eingabebaumes über `bnode` (`NODEPTR_TYPE`) zugegriffen werden; auf die Bäume, die den in der rechten Seite der Regel vorkommenden Nonterminalen entsprechen, kann mittels `kids[0]` (`NODEPTR_TYPE`), etc. zugegriffen werden. (Die Nonterminale sind von links nach rechts durchnumeriert.)

Beispiel:

```
reg: NOT(reg)     #1# freereg(kids[0]->reg); bnode->reg=newreg(); printf("not $%d,$%d", kids[0]->reg, bnode->reg);
reg: ADD(reg,val) #1# freereg(kids[0]->reg); bnode->reg=newreg(); printf("addq $%d,%d,$%d", kids[0]->reg, kids[1]->val, bnode->reg);
```

Die Aktionen erzeugen Alpha-Assemblercode. Dabei bedienen sie sich einer einfachen Registerbelegung (`newreg()`, `freereg()`). Das Register wird im Feld `reg` des Eingabeknoten abgelegt.

Beachten Sie, dass ein Eingabeknoten mehrmals im Reducer vorkommen kann (wenn es Kettenregeln gibt); um dabei Kollisionen zu vermeiden, sollten die Aktionen von Regeln mit verschiedenen linken Seiten auf verschiedene Felder schreiben.

Der erzeugte Reducer hat folgenden Prototype:

[1] Leider müssen Sie hier auf den Luxus einer attributierten Grammatik verzichten und die Ausführungsreihenfolge beachten.

```
void burm_reduce(NODEPTR_TYPE bnode, int goalnt);
```

wobei `goalnt` für das Startnonterminal 1 ist. Für ein vollständiges Parsen eines Baumes `p` samt Aktionen braucht man also nur folgende Aufrufe zu tätigen:

```
burm_label(p);        /* label the tree */
burm_reduce(p, 1);    /* and reduce it  */
```

Die Ausgabe von `bfe` besteht aus der um den Reducer erweiterten `iburg`-Eingabe und erfolgt auf der Standardausgabe. Der Aufruf erfolgt in der Form

```
bfe [file]
```

Die Ausgabe kann gleich an `iburg` weitergeleitet werden:

```
bfe file.bfe | iburg
```

6 Erstellen von Baumgrammatiken

Stellen Sie zunächst sicher, daß Ihre Grammatik alle möglichen Eingabebäume ableiten kann. In den meisten Fällen wird es ein spezielles Nonterminal (z.B. `reg`) geben, auf das sich alle Bäume und Unterbäume reduzieren lassen. Dann brauchen Sie zunächst nur für jeden Operator eine Regel der Form

```
reg: OP2(reg,reg) = 1 (1); /* bei binaeren Operatoren */
reg: OP1(reg)     = 2 (1); /* bei unaeren Operatoren */
reg: OP0          = 3 (1); /* bei nullaeren Operatoren */
```

Die Kosten können Sie dabei entsprechend den Kosten des erzeugten Codes anders festlegen.

Eine solche Baumgrammatik ist dann auf jeden Fall vollständig (kann also alle Bäume ableiten). Sie können dann beliebig weitere Regeln (mit besseren Baummustern) dazufügen, die Grammatik wird vollständig bleiben.

7 Zusammenarbeit mit den anderen Werkzeugen

`iburg` hat keine spezielle Unterstützung für die Zusammenarbeit mit `lex`, `yacc` oder `ox`.

Allgemein können Sie so vorgehen, dass sie einen Operatorbaum für jedes Statement in einem Attribut des Statements aufbauen. Schließlich verwenden Sie ein *parse-tree traversal*, um den Baum in einer bestimmten Reihenfolge abzuklappern, und rufen dabei für jedes Statement und seinen Baum den Labeller und den Reducer auf, wobei der Reducer den erzeugten Code ausgibt. Bei komplizierteren Statements (wie z.B. dem IF-Statement) ist es eventuell nicht so sinnvoll, einen Baum für das ganze Statement zu bauen, sondern statt dessen mehrere Bäume für Unterausdrücke und Unterstatements.

Die Registerbelegung führen Sie am besten während des Reducer-Durchgangs durch.

In den Übungsbeispielen können Sie meistens die Operatoren der Quellsprache 1:1 als Operatoren der Zwischencode-Bäume für die Codeauswahl übernehmen; in realistischen Programmiersprachen dagegen wird öfters ein Quellcode-Konstrukt (z.B. Arrayzugriff) in mehrere Zwischencode-Operatoren (z.B. Multiplikation, Addition, und Speicherzugriff) expandiert.

8 Literatur

[FHP92] ist das ursprüngliche **burg**-Manual. [Pro95] beschreibt, wie `burg` Baumautomaten erzeugt und diskutiert weitere Anwendungen von Baumparsern neben der Befehlsauswahl. [FHP93] beschreibt, wie die von `iburg` und die von ihm erzeugten Parser funktionieren.

Literatur

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992. Available from ftp://kaese.cs.wisc.edu/pub/burg.shar.Z.

[FHP93] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1993. Available from ftp://ftp.cs.princeton.edu/pub/iburg.tar.Z.

[Pro95] Todd A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.