Abstrakte Maschinen 1

# The PowerPC 604 RISC Microprocessor

**The PowerPC 604 RISC microprocessor uses out-of-order and speculative execution techniques to extract instruction-level parallelism. Its nonblocking execution pipelines, fast branch misprediction recovery, and decoupled memory queues support speculative execution.**

*S. Peter Song*

*Marvin Denman*

*Joe Chang*

*Somerset Design Center*

The 604 microprocessor is the third member of the PowerPC family being developed jointly by Apple, IBM, and Motorola. Developed for use in desktop personal computers, workstations, and servers, this 32-bit implementation works with the software and bus in the PowerPC 601 and 603 microprocessors.[1-3] While keeping the system interface compatible with the 601 microprocessor, we improved upon it by incorporating a phase-locked loop and an IEEE-Std 1149.1 boundary-scan (JTAG) interface on chip. In addition, an advanced machine organization delivers one and a half to two times the 601's integer performance.

## Performance strategy

Processor performance depends on three factors: the number of instructions in a task, the number of cycles a processor takes to complete the task, and the processor's frequency.[4,5] Our architecture, which we optimized to produce compact code while adhering to the reduced instruction set computer (RISC) philosophy, addresses the first factor. The high instruction execution rate and clock frequency addresses the other two factors. The 604 provides deep pipelines, multiple execution units, register renaming, branch prediction, speculative execution, and serialization.

**Six-stage superscalar pipeline.** As shown in Figure 1, this deep pipeline enables the 604 to achieve its 100-MHz design. The stages are

- *Fetch.* This stage translates an instruction fetch address and accesses the cache for up to four instructions.
- *Decode.* Instruction decoding determines needed resources, such as source operands and an execution unit.
- *Dispatch.* When the resources are available, dispatch sends instructions to a reservation station in the appropriate execution unit. A reservation station permits an instruction to be dispatched before all of its operands are available.[6] As they become available, the reservation station forwards operands to the execution units. Dispatch can send up to four instructions in program order (in-order dispatch) to four of six execution units: two single-cycle integers, a multicycle integer, a load/store, a floating point, and a branch.
- *Issue/execute.* In each execution unit, this stage issues one instruction from its reservation station and executes it to produce results. The instructions can execute out of program order (out-of-order execution) across the six execution units as well as within an execution unit that has an out-of-order issue reservation station. Table 1 lists the latency and throughput of the execution stages.



0740-7475/94/$04.00 © 1994 IEEE

# Abstrakte
# MASCHINEN

2

- *Completion.* An instruction is said to be *finished* when it passes the execute stage. A finished instruction can be *completed* 1) if it does not cause an exception condition and 2) when all instructions that appear earlier in program order complete. This is known as in-order completion.
- *Write back.* This stage writes the results of completed instructions to the architectural state (or the state that is visible to programmer). Bypass logic permits most instructions to complete and write back in one cycle.

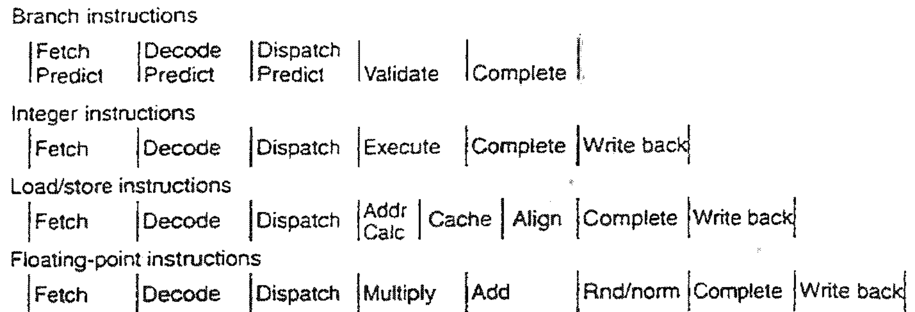



Figure 1. Pipeline description.

Although some designs use even deeper pipelines to achieve higher clock frequencies than the 604 does, we felt that such a design point does not suit today's personal computers. It relies too heavily on one of, or a combination of, a very large on-chip cache, a wide data bus, or a fast memory system to deliver its performance. It would be less than competitive in today's cost-sensitive personal computer market.

**Precise interrupts and register renaming.** Most programmers expect a pipelined processor to behave as a nonpipelined processor, in which one instruction goes through the fetch to write-back stages before the next one begins. A processor meets that expectation if it supports precise interrupts, in which it stops at the first instruction that should not be processed. When it stops (to process an interrupt), the processor's state reflects the results of executing all instructions prior to the interrupt-causing instruction and none of the later instructions, including the interrupt-causing instruction. This is not a trivial problem to solve in multiple, out-of-order execution pipelines. An earlier instruction executing after a later instruction can change the processor's state to make later instruction processing illegal. Sohi gives a general overview of the design issues and solutions.[7]

The 604 uses a variant of the reorder buffer described by Smith and Pleszkun to implement precise interrupts.[8] The 16-entry reorder buffer keeps track of instruction order as well as the state of the instructions. The dispatch stage assigns each instruction a reorder buffer entry as it is dispatched. When the instruction finishes execution, the execution unit records the instruction's execution status in the assigned reorder buffer entry. Since the reorder buffer is managed as a first-in/first-out queue, its examining order matches the instruction flow sequence. To enforce in-order completion, all prior instructions in the reorder buffer must complete before an instruction can be considered for completion. The reorder buffer examines four entries every cycle to allow completion of up to four instructions per cycle.

**Table 1. 604 execution timings.**

| Instruction | Latency | Throughput |
|---|---|---|
| Most integer | 1 | 1 |
| Integer multiply (32x32) | 4 | 2 |
| Integer multiply (others) | 3 | 1 |
| Integer divide | 20 | 19 |
| Integer load | 2 | 1 |
| Floating-point load | 3 | 1 |
| Store | 3 | 1 |
| Floating-point multiply-add | 3 | 1 |
| Single-precision floating-point divide | 18 | 18 |
| Double-precision floating-point divide | 31 | 31 |

Unlike Smith and Pleszkun's reorder buffer, the 604's reorder buffer does not store instruction results. Temporary buffers hold them until the instructions that generated them complete. At that time, the write-back stage copies the results to the architectural registers. The 604 renames registers to achieve this: instead of writing results directly to specified registers, they are written to rename buffers and later copied to specified registers. Since instructions can execute out of order, their results can also be produced and written out of order into the rename buffers. The results are, however, copied from the buffers to the specified registers in program order. Register renaming minimizes architectural resource dependencies, namely the output-dependency (or write-after-write hazard) and antidependency (or write-after-read hazard), that would otherwise limit opportunities for out-of-order execution.[9]

Figure 2 (next page) depicts the format of a rename buffer entry. The 604 contains a 12-entry rename buffer for the general-purpose registers (GPRs) that are used for 32-bit integer operations. The 604 allocates a GPR rename buffer entry upon dispatch of an instruction that modifies a GPR. The dispatch stage writes a destination register number of the

3

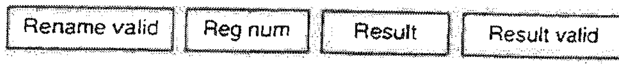



Figure 2. Rename buffer entry format.

instruction to the Reg num field, sets a Rename valid bit, and clears the Result valid bit. When the instruction executes, the execution unit writes its result to the Result field and sets the Result valid bit. After the instruction completes, the write-back stage copies its result from the rename buffer entry to the GPR specified by the Reg num field, freeing the entry for reallocation. For a load-with-update instruction that modifies two GPRs, one for load data and another for address, the 604 allocates two rename buffer entries.

Register renaming complicates the process of locating the source operands for an instruction since they can also reside in rename buffers. In dispatching an integer instruction, the dispatch stage searches its source operands simultaneously from the GPR file and its rename buffer. If a source operand has not been renamed, the processor uses the value read from the GPR file. If a rename exists (indicated by an entry with the Rename valid set and its Reg num field matching the source register number), the Result in the rename buffer is used. It is, however, possible that the result is not yet valid because the instruction that produces the GPR has not yet executed. The dispatch stage still dispatches the instruction since the operand will be supplied by the reservation station when the result is produced. The dispatched instruction contains the rename buffer entry identifier in place of the operand. The GPR file and its rename buffer can use eight read ports for source operands to support dispatching of four integer instructions each cycle.

The 604 also uses a rename buffer for floating-point registers (FPRs) and one more for the condition register (CR). The FPR rename buffer has eight 90-bit-wide entries to hold a double-precision result with its data type and exception status. The FPR file and its rename buffer access three read ports for dispatching one floating-point instruction per cycle. In addition to compare instructions, most integer and floating-point instructions can also generate negative, positive, zero, and overflow condition results. One of the eight fields in the 32-bit CR stores these 4-bit condition results. The 604 treats each field as a 4-bit register and applies register renaming using an eight-entry CR rename buffer.

**Branch prediction and speculative execution.** Because today's application software contain a high percentage of branch instructions, correctly predicting the outcome of these instructions is crucial to keeping the multiple instruction pipelines flowing and for achieving two to three times the execution rate of scalar processors. The 604 uses dynamic branch prediction in the fetch, decode, and dispatch stages to predict as well as correct branch instructions early.

The 604's speculative execution strategy complements its branch prediction mechanisms. The strategy is to fetch and execute beyond two unresolved branch instructions. The results of these speculatively executed instructions reside in rename buffers and in other temporary registers. If the prediction is correct, the write-back stage copies the results of speculatively executed instructions to the specified registers after the instructions complete.

Upon detection of a branch misprediction, the 604 takes quick action to recover in one cycle. It selectively cancels the instructions that belong in the mispredicted path from the reservation stations, execution units, and memory queues. It also discards their results from the temporary buffers. In addition, the processor resumes its previous state to start executing from the correct path even before the mispredicted branch and its earlier instructions have completed. Since the 604 detects a branch misprediction many cycles before the branch instruction completes, its fast recovery scheme helps to maintain performance of those applications with high data cache miss rates and whose branches are difficult to predict.

**Serialization.** A serialization mechanism delays execution of certain instructions that would otherwise be expensive to execute speculatively in the 604's multiple-pipeline, out-of-order execution design. This mechanism delays infrequently used instructions until they can safely execute while permitting later instructions to execute. Some examples are the move to and from special-purpose register instructions, the extended arithmetic instructions that read the carry bit, and the instructions that directly operate on the CR, which the PowerPC architecture provides for calculating complex branch conditions. This mechanism also controls store instructions since it is difficult to undo stores.

The dispatch stage sends a serialized instruction to the proper execution unit with an indication that it should not be executed. When all prior instructions have completed and updated their results to the architectural state, the completion stage allows the serialized instruction to execute. Once the serialized instruction is dispatched, dispatch continues to dispatch the following instructions so they can execute before the serialized instruction. When the serialized instruction is completed, the later instructions also complete upon finishing execution. This minimizes the penalty of serialized instructions.

## Machine organization

Figure 3 shows the fetch address generation logic. The fetch stage selects an address from the addresses generated in the different pipeline stages each cycle. Since an address generated in a later stage belongs to an earlier instruction, its selection precedes an address from an earlier stage.

The completion stage detects exception conditions and generates an exception handler address. This stage also

updates the program counter (PC) with the target address of a taken branch instruction, or advances it by the number of instructions being completed. The branch execute stage may correct the instruction fetch with the branch target address if the branch is mispredicted as not taken and with the sequential address if the branch is mispredicted as taken. The dispatch and decode stages may change the fetch address with either the target or sequential address of a branch instruction being predicted. There are two copies of the target and sequential address registers in the decode, dispatch, and execute stages, since there can be up to two branch instructions in each stage. The completion stage also has two target registers to handle up to two finished branch instructions.

If the fetch address hits in the branch target address cache (BTAC), the target address becomes the fetch address. Otherwise, the instruction fetch continues sequentially. The 64-entry, fully associative BTAC holds the target addresses of the branches that are predicted to be taken. If a branch is predicted as not taken for its next encounter, the branch execute stage removes it from the BTAC. The BTAC is accessed with the fetch address, and not with a branch instruction address, providing a zero-cycle fetch penalty for taken branches. Although there may be multiple branch instructions in the four instructions being fetched, the BTAC provides the target address of the first-predicted taken branch instruction.

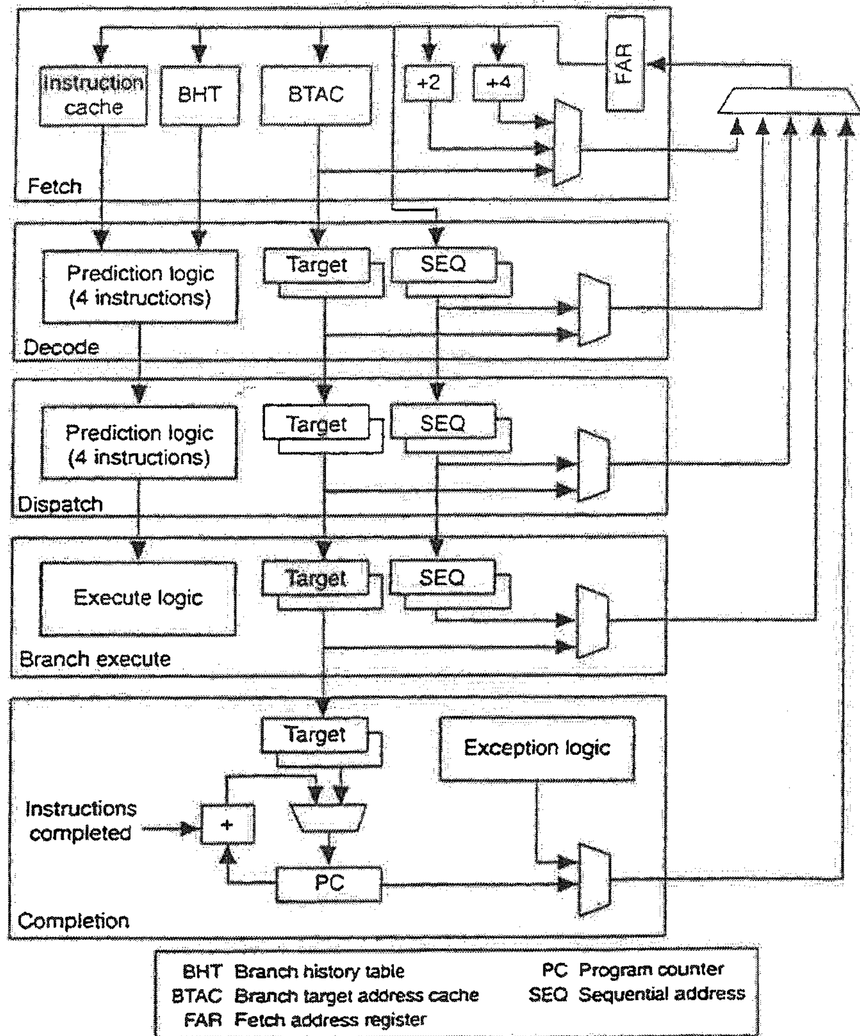



Figure 3. Instruction fetch address generation logic.

**Instruction decode and dispatch.** The pipeline decodes four instructions every cycle to determine exception conditions, as well as the resources needed by the instructions. The resources include the execution unit, source operands, and destination registers. Decoding the instructions before the dispatch stage simplifies the dispatch logic without using predecoded bits in the instruction cache. Predicting branch instructions in the first two entries of the decode buffer minimizes the performance penalty of adding the decode stage.

When the decode stage detects an unconditional branch that was not in the BTAC, it corrects the instruction fetch to the target address of the branch. It also predicts conditional branches with the execution history found in the branch history table (BHT).[10] Each entry of the 512-entry BHT denotes one of the four history states: strong not taken, weak not taken, weak taken, and strong taken. The table updates the history state with the actual outcome of the branch that is mapped to the entry. To simplify the design, each entry in the BHT maps to every 512th instruction address. This allows the BHT to be accessed with the fetch address and to return the four entries mapped to the four instructions starting with the fetch address.

Not all conditional branch instructions use the BHT. The architecture provides a count register (CTR) value as a branch condition to support loops in programs. Only the conditional branch instructions that do not depend on the CTR value use, as well as update, the BHT. Those that do depend on the CTR are predicted, based on the value of the shadow CTR. The shadow CTR has a future state of the CTR that is

updated by speculatively executed move-to-CTR or branch-and-decrement instructions. This prediction scheme is very effective on branches that control loop iteration.

The dispatch buffer sends up to four instructions to four of the six execution units each cycle. As space allows, more instructions advance from the decode stage into the four-entry dispatch buffer. The 604 places only a few restrictions on dispatch to enable a high-speed implementation. They are

1) *One instruction per execution unit.* Since each execution unit can start only one instruction per cycle and an instruction can bypass the reservation station if the execution unit is available, dispatching one instruction per unit simplifies the logic without imposing an undue performance penalty. Two identical single-cycle integer units handle the more frequent instructions.
2) *Resources available.* Each instruction needs a reorder buffer entry, a reservation station entry in the appropriate execution unit, and rename buffer entries to hold its results. Available resources depend on the state of the instructions previously dispatched as well as those currently being dispatched.
3) *Stop dispatch after branch.* Instructions following a branch instruction are not dispatched in the same cycle as the branch is dispatched. This restriction simplifies saving the processor state, which allows immediate canceling of speculatively executed instructions that follow predicted branches.
4) *In-order dispatch.* Dispatching instructions in order results in only a small cost in performance and greatly simplifies resource allocation and dispatch logic. Out-of-order execution is introduced with six independent execution pipelines and out-of-order issue reservation stations to achieve performance comparable to an out-of-order dispatch design.

**Reservation stations and result forwarding.** A two-entry reservation station on every execution unit allows instructions to be dispatched before obtaining all of their operands. Without a reservation station, an instruction cannot be dispatched until all of its source operands become available, either in the register file or in its rename buffer. Without reservation stations, the 604's in-order dispatch design would be more complex, since it would have to detect data dependencies and would frequently stall. The reservation stations in the three integer units can issue instructions out of order to allow a later instruction to bypass an earlier stalled instruction. The branch, load/store, and floating-point unit reservation stations may only issue instructions in order.

Each execution unit provides one result bus for each type of result it produces. For instance, the multicycle integer unit has one result bus for the GPR and another for the CR data types. Figure 4 shows the four GPR result buses and the reservation stations and GPR rename buffer that are connected to them. Each GPR reservation station entry monitors all four GPR result buses for any missing A or B operands, denoted as A op and B op in the figure. When an execution unit returns a result and the associated GPR rename buffer entry identifier, the reservation station compares the identifier against those in its entries. When a match is found, it forwards the result to the waiting instruction. For returning the update address of a load-with-update instruction while executing one load instruction per cycle, the load/store unit shares the result bus of the less frequently used multicycle integer unit.

It is interesting to note why the 604 uses a reorder buffer, rename buffers, and reservation stations to provide the same functions that a DRIS (deferred-scheduling, register-renaming instruction shelf) in the Metaflow architecture provides.[11] A DRIS entry consists of instruction status fields that a reorder buffer entry would have, source operand fields that a reservation station entry would have, and destination fields that a rename buffer entry would have. (The 604's reservation station entry uses a separate source operand to store either an immediate or a copy of the source operand. Although the DRIS figure in the Metaflow article shows only the ID field to indicate the DRIS entry with the source operand, it is likely that another field is needed to store an immediate operand.)

Two disadvantages of the DRIS had we used it in the 604 design are

- *Scheduling overhead.* The DRIS instruction scheduling is more complicated than the 604's dedicated reservation stations since the next instruction for an execution unit must be the first "ready" instruction of the "right" type in all DRIS entries.
- *Single result type.* DRIS supports renaming of only one register type, whereas the 604 needs three. Say that more than one DRIS is used, as described in Popescu et al.,[11] to support separate integer and floating-point registers. One of them would have to house all instructions to provide precise interrupts while not being able to provide register renaming. An alternative is to design one DRIS to accommodate the largest register type.

**Execution units.** The branch execution unit can hold two branch instructions in its reservation station and two more finished branch instructions. It serves to validate branch predictions made in earlier stages, and also verifies that the predicted target matches the actual target address. If a misprediction is detected, the branch execution unit redirects the fetch to the correct address and starts the branch misprediction recovery.

The 604 has a three-stage complex integer unit (CFX) to execute integer multiply, divide, and all move to and from special-purpose register instructions. The CFX can sustain one multiply instruction per cycle for 32×16-bit and those 32×32-bit multiplies whose B operand is representable as a

6

17-bit signed integer. It can sustain one multiply per two cycles for larger 32×32-bit operands. The CFX also uses the multiply pipeline stages to execute a divide instruction in 20 cycles. The 604's two simple integer units execute all other integer instructions in one cycle.

The three-stage floating-point unit can sustain one double-precision multiply-add per cycle, one single-precision divide every 18 cycles, or one double-precision divide every 31 cycles. It complies fully with the IEEE-Std 754 floating-point arithmetic standard. The 604 provides hardware support for denormalization, exceptions, and three graphics instructions. It also provides a non-IEEE mode for graphics support. The non-IEEE mode converts a denormalized result to zero to avoid prenormalization in subsequent operations.

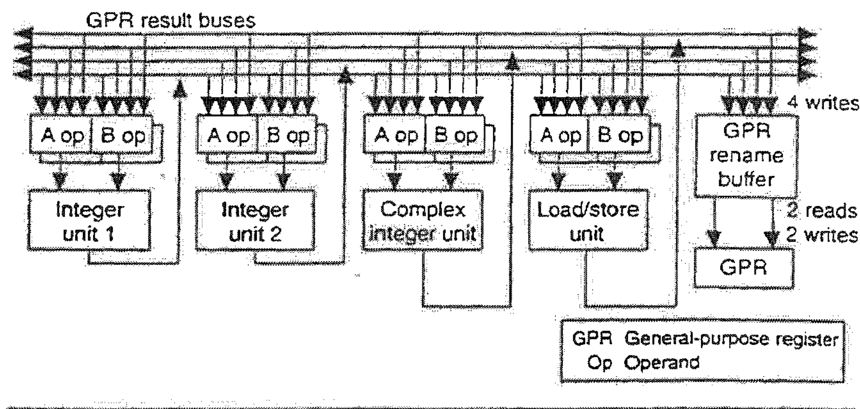



Figure 4. GPR result buses, reservation stations, and rename buffer.

**Instruction completion and write back.** After an instruction executes, the execution unit copies results to its rename buffer entries and the execution status to its reorder buffer entry. Among other things, the execution status indicates whether the instruction finished execution without an exception. Of the four reorder buffer entries examined every cycle, up to four instructions that finished without an exception complete in program order.

Other than the in-order completion necessary to support precise exceptions, the 604 imposes only a few additional restrictions on instruction completion. They are

1) *Stop before a store instruction.* Since a store data operand is read from the register file in the completion stage, a store instruction cannot complete if its store operand is still in the rename buffer. Stopping completion before a store instruction allows the store operand to be written to the register file, even if it is produced by an instruction currently being completed.
2) *Stop after a taken branch instruction.* Since a taken branch instruction sets the program counter to its target address, it is speed critical to advance the program counter from the new target address by the number of instructions completed after the taken branch in one cycle. Stopping completion after a taken branch instruction avoids this logic altogether.

To minimize effects of long execution latency on in-order completion, the completion stage overlaps with the last execution cycle for those instructions with multicycle execution stage. These include the multiply, divide, store, load miss, and execution serialized instructions. A store instruction completes as soon as it is translated without an exception. Similarly, a load instruction that misses in the data cache completes upon translation without an exception. Since the reservation stations can forward the load data when it is available to the dependent instructions, the load miss can safely be completed.

Most superscalar designs impose additional restrictions due to a limited number of ports to register files. For instance, four write ports would be required to complete up to four instructions if each one can update one register. The 604 GPR file would require eight write ports to complete four load-with-update instructions per cycle. The 604 avoids this problem by decoupling instruction completion from register file updates using the write-back stage. Instructions complete without regard to the type or number of registers they update. The completion stage updates their results if ports are available; if not, the write-back stage updates them. The rename buffer entries function as temporary buffers for those instructions that are not completed and as write-back buffers for those that are. All three GPR, FPR, and CR rename buffers contain two read ports for write back. Correspondingly, the three register files have two write ports for write back.

## Memory operations

High-speed superscalar processors require a greater memory bandwidth to sustain their performance. The 604 meets the increased demand with large on-chip caches, nonblocking memory operations, and a high-bandwidth system interface. The 604 takes advantage of the weakly ordered memory model, to which the PowerPC architecture subscribes, to offer efficient memory operations. Although loads and stores that hit in the data cache can bypass earlier loads and stores, program order memory access can be enforced with instructions provided for this purpose.

**Load/store unit.** Figure 5 (next page) shows a block diagram of the load/store unit and the memory queues. This unit has a two-cycle execution stage. It calculates the memory address and translates that address with a 128-entry, two-

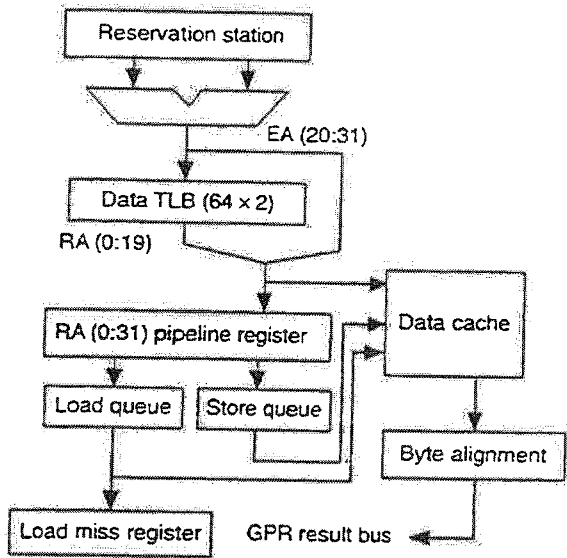



Figure 5. Load/store unit and memory queues.

way set-associative translation lookaside buffer (TLB) in the first cycle. The second cycle processes loads making a speculative cache access and aligns bytes when the access hits in the cache. The pipelined execution stage executes one load or store instruction per cycle.

In the first half cycle, the load/store unit calculates a load instruction's memory address, denoted as EA in the figure. It translates the real address, denoted as RA in the figure, and the data cache access begins in the second half cycle. If the access hits in the cache, the unit aligns the data and forwards it to the rename buffer and the execution units in the second cycle. If the access misses, the unit places the instruction and its real address in the four-entry load queue. When a load miss completes, it accesses the cache a second time. If the load is still a miss, the unit moves it to the load miss register while reloading the missing cache line. This permits a second load miss to access the cache and to initiate the second cache line reload before the first is brought in.

The unit calculates the memory address of a store and translates it in the first cycle. It does not write the data to the cache, however, until after the store instruction completes. The unit places the instruction and its real address in the six-entry store queue. Since the data cache is not accessed in the second cycle, it is available for an earlier store from the store queue (if necessary) or load miss from the load queue (if necessary).

When a store instruction completes, the load/store unit marks it *completed* in the store queue so that instruction completion can continue without waiting for storage to the cache or memory. If the store hits in the cache, the unit writes it to the cache and removes it from the store queue. If the store is a miss, the unit will bypass it in the store queue to allow later stores to take place while cache reloading proceeds. Multiple store misses can be bypassed in the store queue.

Figure 6 shows the store queue structure. Four pointers identify the state of the store instructions in the circular store queue. When a store has finished execution (or successful translation), the load/store unit places it in the finished state. When it completes, the finish pointer advances to place it in the completed state. When it is committed to cache or memory, the completion pointer advances to place it in the committed state. If the store hits in the cache, advancing the commit pointer removes it from the queue. If the store is a miss, the commit pointer does not advance until the missing cache line is reloaded and the store is written to the cache. While the cache line is being reloaded, the next store indicated by the completion pointer can access the cache. If this store hits in the cache, the unit removes it from the queue. If it misses, another cache line reload begins.

**Caches.** The 604 provides separate instruction and data caches to allow simultaneous instruction and data accesses every cycle. Both 16-Kbyte caches provide byte parity protection and a four-way set-associative organization with 32-byte lines. They are indexed with physical addresses, have physical tags, and make use of the least recently used replacement algorithm.

The instruction cache provides a 16-byte interface to the fetch unit to support the four-instruction dispatch design. This nonblocking cache allows subsequent instructions to be fetched while a prior cache line is being reloaded. This design is particularly beneficial if the missing cache line belongs in a mispredicted path, since it allows the correct instructions to be accessed immediately after a branch misprediction recovery. The instruction cache also provides streaming, a mechanism to forward instructions as they are received from off-chip cache or memory.

The instruction cache does not maintain coherency; instead, the architecture provides a set of instructions for software to manage coherency. In particular, the instruction cache block invalidate (ICBI) instruction causes all copies of the addressed cache line to be invalidated in the system. The ICBI generates an invalidation request, to which all coherent caches must comply.

The data cache contains a 64-bit data interface to the load/store unit for data access, MMU for tablewalking, and bus interface unit (BIU) for cache line reloading and snooping. (The architecture specifies an algorithm to traverse page table entries that define the virtual-to-physical memory mappings. "Tablewalk" refers to a hardware implementation of the algorithm.) The data cache's two copy-back buffers support nonblocking cache operations. A copy-back buffer holds a dirty (modified) cache line that is being replaced or that hits on a snoop request. The data cache moves an entire cache line into a copy-back buffer in one cycle to minimize the

8

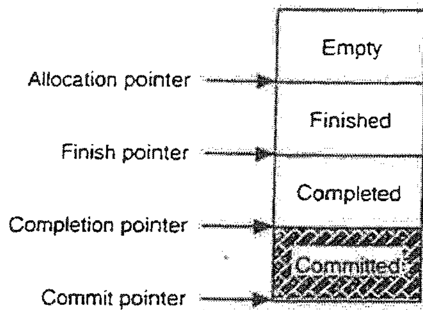



Figure 6. Store queue structure.

cycles the cache is unavailable.

The MESI (modified, exclusive, shared, invalid) coherency protocol defined in the *PowerPC 60x Processor Interface Specification* keeps the data cache coherent.[12] A duplicated data tag array supports two-cycle snoop response with minimum performance impact to the normal cache operations.

To further support nonblocking cache operations, we've extended the MESI coherency protocol with one more state. As illustrated in a simplified state diagram in Figure 7, the protocol assigns the new state, Allocated, to the block selected to hold the missing cache line. All necessary information for this particular miss, including the address and set number, remains in the memory queue and later completes the cache line reload. The cache line becomes either shared or exclusive, based on the coherency response.

The software can individually disable, invalidate, or lock both instruction and data caches. While a cache is disabled, all accesses bypass it and directly access the off-chip cache or memory. While a cache is locked, it is accessible but its contents cannot be replaced. All cache misses, in this case, are accessed from off chip as cache-inhibited accesses. Coherency is, however, maintained even when a cache is locked. The data cache supports the cache touch instructions, which initiate reloading of the specified cache line if it is not in the cache. These instructions can effectively shorten cache miss rates and latency.

**Bus interface unit and memory queues.** The 604's BIU implements the *PowerPC 60X Processor Interface Specification* to ensure bus compatibility with the 601 and 603 microprocessors. A split transaction mode allows the address bus to operate independently of the data bus, freeing the data bus during memory wait states. To support the split transaction mode, the BIU uses the address and data buses only during what are known as address tenure and data tenure cycles. The BIU also provides a pipelined mode, in which up to three address tenures can be outstanding before data for the first address is received. If permitted, the BIU will complete one or more write transactions between the address and data tenures of a read transaction. Byte parity protects its 32-bit address and 64-bit data buses.

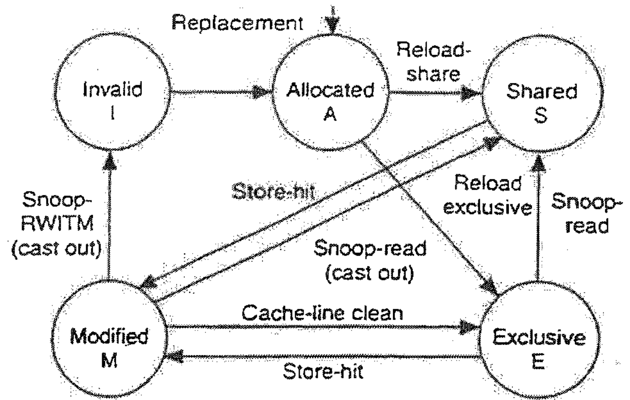



Figure 7. Simplified data cache state diagram.

Figure 8 (next page) shows the address and data queues that implement split and pipelined transaction modes. Four types of memory queues support the four types of operations: line fill, write, copy back, and cache control. For a line-fill operation, the line-fill address from either the instruction or data cache remains in the memory address queue until the address can be sent out in an address tenure. After the address tenure, the address transfers to the line-fill address queue. This releases the address bus for other transactions in the split transaction mode. As each double word for the cache line returns, it moves to the line-fill buffer and also forwards to the load/store unit.

During a write operation the address stays in the memory address queue, and the data in the write buffer, until both the address and data can move out in a write transaction. The size of a write transaction can vary from 1 to 8 bytes to handle nondouble-word aligned writes. Similarly during a copy-back operation, the address remains in the copy-back address queue, and the data in the copy-back buffer, until both the address and data transfer in a 32-byte burst write transaction. For a cache control instruction or a store to a shared cache line, the address stays in the cache control address queue until an address-only transaction broadcasts the cache control command. Since all address queues in the 604 are considered part of the coherent memory system, the BIU checks them against data cache and snoop addresses to ensure data consistency and maintain MESI coherency protocol.

## System support features

The 604 provides several features to support robust system design such as instruction and data address breakpoints and single-step and branch instruction tracing facilities for software debugging. It also provides performance-monitoring functions for the system to profile software performance

9

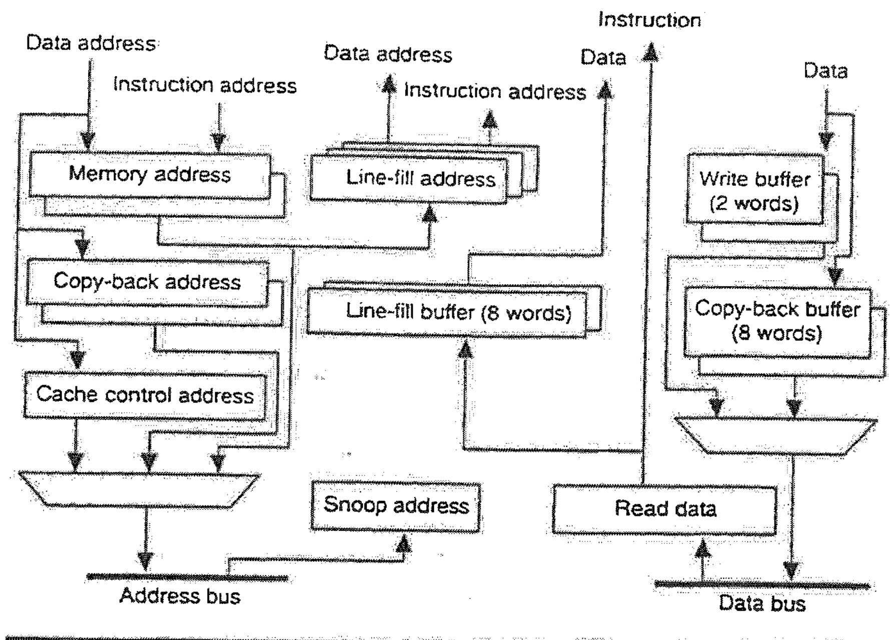



Figure 8. Address and data queue organization.

**Table 2. The 604 physical characteristics.**

| Item | Characteristics |
|---|---|
| Technology | 0.5-μm CMOS, 4 metal layers |
| Die size | 196 mm², 12.4x15.8 mm |
| Transistor count | 3.6 million |
| Cache size | 16-Kbyte I-cache and D-cache |
| Voltage | 3.3V, 5V I/O tolerant |
| Power dissipation | Less than 10W at 100 MHz |
| Signal I/Os | 171; CMOS/TTL compatible |
| Package | 304-pin CQFP |

without additional hardware. These functions can determine many key performance parameters, such as instruction execution rate, branch prediction rate, cache hit rates, and average cache miss latency.

The 604 design follows the level-sensitive scan design methodology to provide high test coverage. As required by LSSD rules, every storage element, except in arrays, connects to a scan chain that starts with a chip input pin and ends on a chip output pin. During test mode, storage elements in a scan chain behave as a shift register that can also capture inputs to exercise a sequential digital network in a combinational manner. The 604's common on-chip processor (COP) provides many functions to control and observe the storage elements. Some of the functions useful for chip and system debugging include setting instruction or data address breakpoints, single stepping, running *N* cycles, and reading and writing system memory locations as well as any storage element within the processor. The COP functions are implemented as an extension to the IEEE-1149.1 specification, and are controlled entirely through that interface.

System designers can configure the 604 processor operating frequency as one, one-and-a-half, two, or three times the system bus frequency. The on-chip phase-locked loop generates the necessary processor clocks from the bus clock. The 604 also provides a nap mode, which clocks only external interrupt detection logic and the phase-locked loop. It enters nap mode under software control and exits from the mode upon detecting an interrupt. The 604 can still service snoop requests if the system asserts the RUN pin to run the clocks while in the nap mode. We estimate nap mode power consumption at less than 0.4 watts.

Table 2 lists some of key physical characteristics of the 604. Figure 9 shows the 604 die photo.

DESIGNED TO MEET LOW-COST needs of the personal computer market, the 604 performs well with inexpensive, as well as expensive, memory systems. The 604's large on-chip caches help to maintain performance of well-behaved applications that exhibit localities. For those with erratic behaviors and access patterns, speculative execution guided by dynamic branch prediction helps to reduce on-chip cache miss latency. The nonblocking execution pipelines and the memory queues that decouple the pipelines from memory access further help to reduce the effects of cache misses. The split and pipelined modes use the system bus to provide greater bandwidth while maintaining compatibility with the 601 and 603 microprocessors.

## References


1. E. Silha, "The PowerPC Architecture," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, Austin, Tex., 1993.
2. C. Moore, "The PowerPC 601 Microprocessor," *IBM RISC System/6000 Technology: Volume II*, IBM Corporation, 1993.
3. B. Burgess et al., "The PowerPC 603 Microprocessor: A High Performance, Low Power, Superscalar RISC Microprocessor,"

*Proc. Compcon*, IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. [illegible]

4. S. White et al., "How Does Processor MHz Relate to End-User Performance?" Part 1 of 2, *IEEE Micro*, Aug. 1993, pp. [illegible]
5. S. White et al., "How Does Processor MHz Relate to End-User Performance?" Part 2 of 2, *IEEE Micro*, Oct. 1993, pp. [illegible]-89
6. R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J.*, Vol. 11, Jan. 1967, pp. 25-33.
7. G. Sohi, "Instruction Issue Logic for High-Performance Interruptible, Multiple Function Unit, Pipelined Computers," *IEEE Trans. Computers*, Mar. 1990, pp. 349-359.
8. J. Smith and A. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. Computer Architecture*, IEEE, Piscataway, N.J., 1985, pp. 36-44.
9. M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.
10. J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Jan. 1984, pp. 6-22.
11. V. Popescu et al., "The Metaflow Architecture," *IEEE Micro*, June 1991, pp. 10-13, 63-73.
12. M.S. Allen, et al., *Overview of the PowerPC Bus Interface*, *IEEE Micro*, this issue, pp. 42-51.

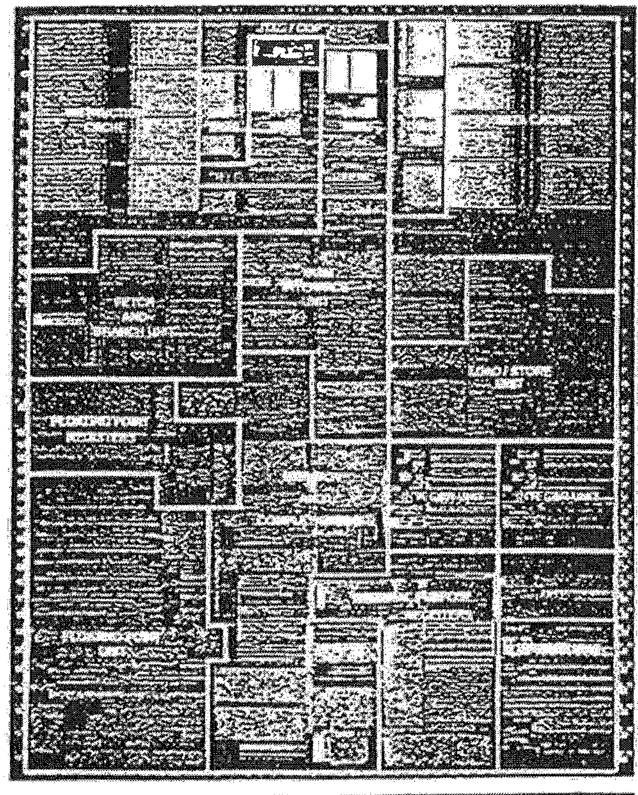


Figure 9. Die photo of the PowerPC 604.

**S. Peter Song** is a senior engineer in the Systems Technology and Architecture Division of IBM. He led the definition of the 604 microarchitecture and later designed the speculative execution, completion, and exception control logic. Song holds BS, MS, and PhD degrees in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society, Eta Kappa Nu, and Tau Beta Pi.

**Marvin Denman** is a principal staff engineer in the RISC Microprocessor Division of Motorola, Inc. He has contributed to the definition of the 604 microarchitecture and later designed the fetch and branch-processing logic. Denman holds a BS degree in computer science from Texas A&M University and an MS in electrical engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society.

**Joe Chang**'s biography, photograph, and address appear on p. 51 of this issue.

Direct questions concerning this article to S. Peter Song, Somerset Design Center, 11400 Burnet Road, M/S 973, Austin, TX 78758; spsong@ibmoto.com

## Reader Interest Survey

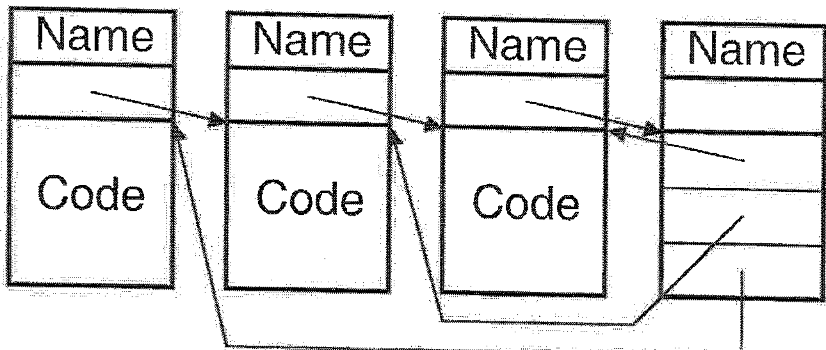
Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 154 Medium 155 High 156

# Threaded Code

threaded = aufgefädelt

Die interne Darstellung eines threaded Interpreters ist eine Liste von Adressen vorher definierter interner Darstellungen (Unterprogrammen). Diese Darstellungen sind in einer linearen Liste aufgefädelt.

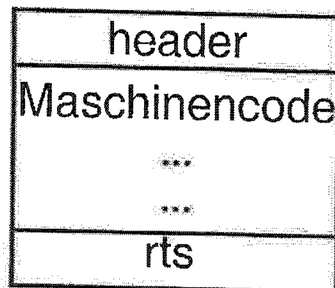



Die einzelnen Elemente werden übersetzt. Die abstrakte Maschine ist meistens eine Stack-maschine.

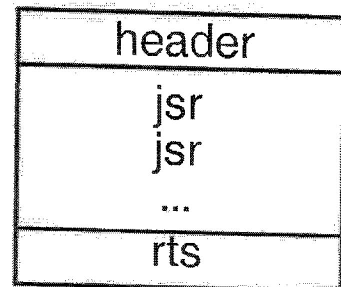
FORTH

## subroutine threaded code

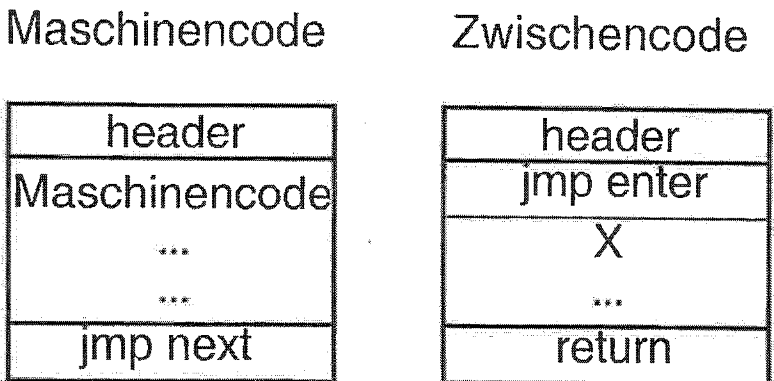
Maschinencode




Zwischencode



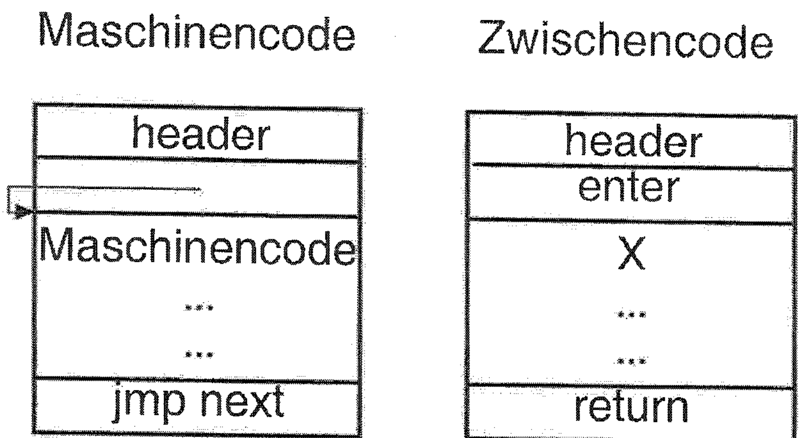
## direct threaded code




```
enter:   push   ic
         move   ic = instr + jmp_len
next:    move   instr = (ic)+
         jmp    (instr)

return:  pop    ic
         jmp    next
```

## indirect threaded code




```
enter:   push   ic
         move   ic = instr
next:    move   instr = (ic)+
         move   ind = (instr)+
         jmp    (ind)

return:  pop    ic
         jmp    next
```

# token threaded code

Maschinencode

| header |
| --- |
| Maschinencode<br>...<br>... |
| jmp next |

Zwischencode

| header |
| --- |
| jmp enter |
| X<br>...<br>... |
| return_token |

```
enter:   push   ic
         move   ic = addr + jmp_len
next:    move   instr = (ic)+
         move   addr = tab(instr)
         jmp    (addr)

return:  pop    ic
         jmp    next
```

# indirect token threaded code

Maschinencode

| header |
| --- |
| token |
| Maschinencode<br>...<br>... |
| jmp next |

Zwischencode

| header |
| --- |
| enter_token |
| X<br>...<br>... |
| return_token |

```
enter:   push   ic
         move   ic = addr + token_len
next:    move   instr = (ic)+
         move   addr = tab(instr)
         move   addr = (addr)
         move   addr = tab(addr)
         jmp    (addr)

return:  pop    ic
         jmp    next
```

# Kosten auf dem MC68000

subroutine threaded    220 Zyklen    4 byte

```
enter:
next:     bsr + rts
return:   rts
```

enter = 0    next = 34    return = 16

direct threaded    162 Zyklen    2 byte

```
enter:    move.w   a6,-(a7)
          lea      a6,#4(a5)
next:     move.w   (a6)+,a5
          jmp      (a5)
return:   move.w   (a7)+,a6
          move.w   (a6)+,a5
          jmp.w    (a5)
```

enter = 42    next = 16    return = 24

indirect threaded    212 Zyklen    2 byte

```
enter:    move.w   a6,-(a7)
          move     a5,a6
next:     move.w   (a6)+,a5
          move.w   (a5)+,a4
          jmp      (a4)
return:   move.w   (a7)+,a6
          move.w   (a6)+,a5
          move.w   (a5)+,a4
          jmp      (a4)
```

enter = 36    next = 24    return = 32

token threaded    248 Zyklen    1 byte

```
enter:    move.b   a6,-(a7)
          lea      a6,#4(a4)
next:     move.b   (a6)+,a5
          move.w   tab(a5),a4
          jmp      (a4)
return:   move.w   (a7)+,a6
          move.b   (a6)+,a5
          move.w   tab(a5),a4
          jmp      (a4)
```

enter = 44    next = 28    return = 36

14

indirect token threaded 408 Zyklen 1 byte

```
enter:   move.b  a6,-(a7)
         lea     a6,#1(a4)
next:    move.b  (a6)+,a5
         move.w  tab(a5),a4
         move.b  (a4)+,a5
         move.w  tab(a5),a3
         jmp     (a3)
return:  move.w  (a7)+,a6
         move.b  (a6)+,a5
         move.w  tab(a5),a4
         move.b  (a4)+,a5
         move.w  tab(a5),a3
         jmp     (a4)
```

enter = 64 next = 48 return = 56

# Kosten auf dem MIPS R3000

subroutine threaded 17 Zyklen 8 byte

Ein delay-slot für Lade- und Sprungbefehle

`r31` return address register

```
enter:   add   r30,r30,-4
         sw    r31,(r30)

next:    bal label + j r31
return:  lw    r31,(r30)
         add   r30,r30,4
         j     r31
```

enter = 2 next = 2 return = 3

direct threaded    30 Zyklen    4 byte

```
enter:   sw      r1,-4(r30)
         add     r1,r2,8
         lw      r2,4(r2)
         add     r30,r30,-4
         j       r2
         nop
next:    lw      r2,(r1)
         delay slot
         jmp     r2
         add     r1,r1,4
return:  lw      r1,(r30)
         add     r30,r30,4
         next
```

enter = 6    next = 3    return = 6

# Die Sprache ZIP
# (Z80 Interpretative Processor)

Zip ist eine FORTH artige Sprache

Wichtigste Merkmale:

Indirekt threaded code

Stack orientiert

linear verkettete 3 Zeichen lange Namen

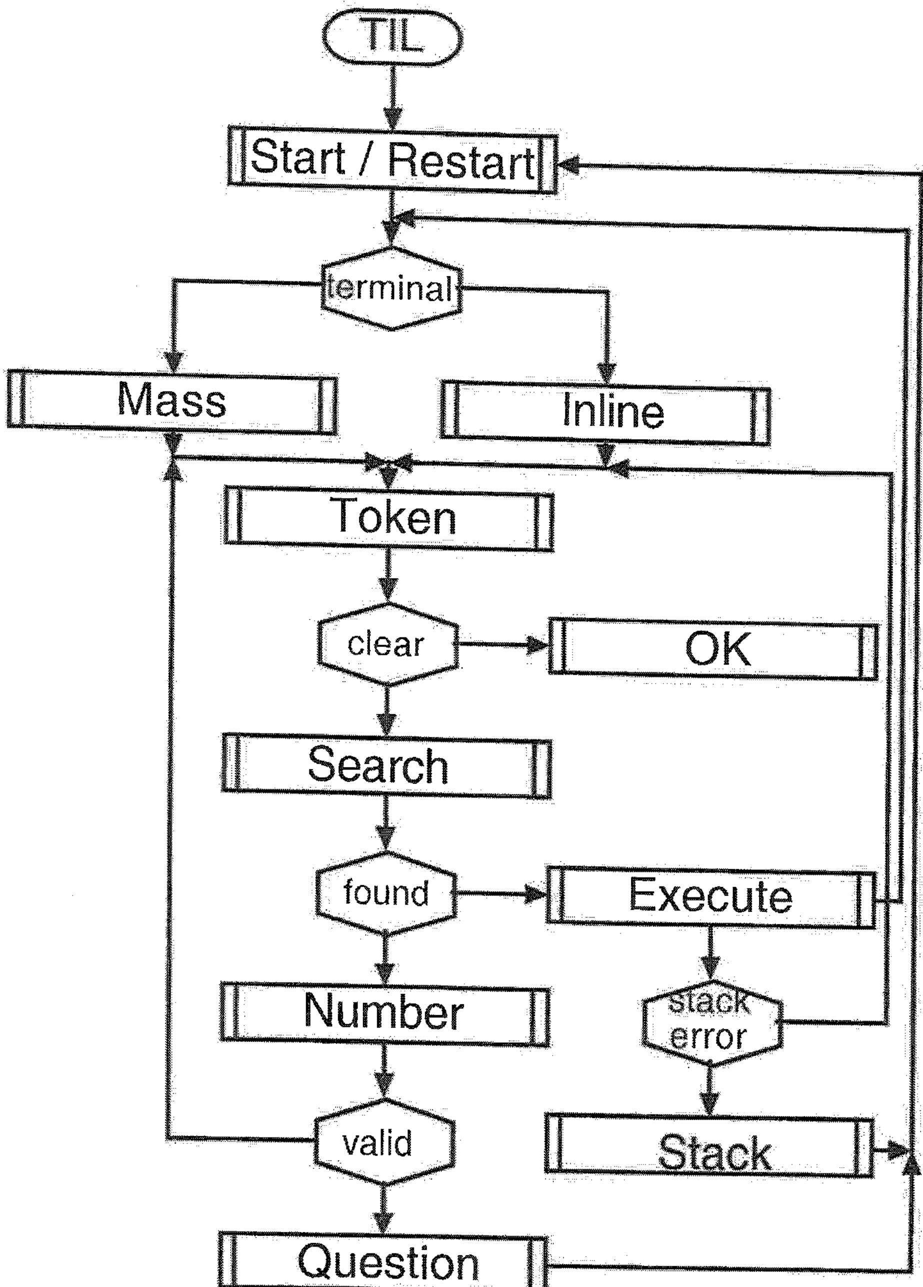
Z80 Maschiencode und ZIP

4 Kb Memory

integer Arithmetik

Loop-Anweisungen (kein goto)

einfacher Line Editor




# ZIP Funktionen

Start/Restart: initialisiert Stack pointer, System Variable und Execute-Modus

Mass/Inline: liest eine Zeile in Zeilenbuffer

Token: liest den nächsten token aus dem Zeilenbuffer und stellt ihn in Dictionary

Ok: gibt eine Ok-Meldung aus

Search: durchsucht Dictionary nach dem token, gibt die Adresse und ein gefunden Flag auf den Stack

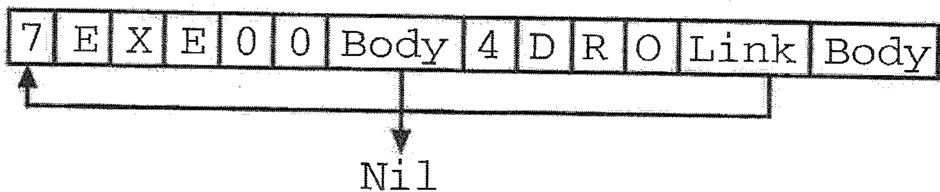
Execute: Wenn execute-Mode, dann Kommando ausführen, wenn compile-Mode, dann wenn immediate-Kommando, dann ausführen, sonst übersetzen

Stack: gibt Fehlermeldung aus, geht zu Start

Number: stellt Nummer auf Stack oder übersetzt sie

Question: gibt token und Fehlermeldung aus

# Dictionary Format




Nummern: Integer, 2 byte auf Stack, im Body zuerst literal handler und dann der Wert

Strings: Auf Stack letzter Buchstabe zuunterst mit high-byte gesetzt, in Body zuerst Länge, dann die einzelnen Buchstaben

Boolesche Werte: 1..true, 0..false, bei Abfragen jeder Wert ≠ 0 true

# Konstanten und Variablen

Konstante: n CONSTANT name

DECIMAL 255 CONSTANT max

: ... max ...;
: ... 255 ...;
Name und Wert mit Zeiger auf Konstanten-aktivierungscode gespeichert

Variable: n VARIABLE name

Bei Aktivierung des Namens wird Adresse auf den Stack gelegt.

Arrays: n VARIABLE name size DP +!

Variable wird angelegt, dann wird der Dictionaryzeiger um size erhöht. Bei Indizierung muß der Index zur Adresse addiert werden.

Users: USERS n

Addiert n zur Adresse vom User-Speicher-block. Dieser Speicherblock kann beliebig im Speicher stehen.

# Systemparameter

Können als Variable oder als Register verwendet werden

| | |
|---|---|
| IR: | Instruction Register |
| WA: | Word-Address Variable |
| SP: | Daten Stackpointer |
| RSP: | Return Stackpointer |
| MODE: | Execute/Compile Mode false/true |
| STATE: | Immediate Wort wenn MODE=STATE |
| DP: | Dictionary Zeiger Variable |
| CONTEXT: | Suchvokabular |
| CURRENT: | Definitionsvokabular |
| START: | true, wenn erster Start |
| LPB: | Zeilenbuffer Zeigervariable |
| BASE: | Zahlenbasis Variable |

# Stackbefehle

| | |
|---|---|
| DROP: | Entferne oberstes Stackelement |
| DUP: | verdopple oberstes Stackelement |
| SWAP: | vertausche obere 2 Stackelemente |
| OVER: | kopiere 2. Stackelement auf den Stack |
| 2DUP: | verdreifache oberstes Stackelement |
| 2SWAP: | vertausche oberstes und 3. Stack-element |
| 2OVER: | kopiere 3. Stackelement auf den Stack |
| RROT: | ABC ⇒ CAB |
| LROT: | ABC ⇒ BCA |
| !: | `*stack[top] = stack[top-1]; top-=2;` |
| +!: | `*stack[top] += stack[top-1]; top-=2;` |
| 0SET: | `*stack[top--] = 0;` |
| 1SET: | `*stack[top--] = 1;` |
| @: | `stack[top] = *stack[top];` |
| <R: | Pop Daten- und Push Returnstack |
| R>: | Pop Return- und Push Datenstack |

# Rechenbefehle

ABS, MINUS, +, -, *, /, /MOD, MOD/, */, */MOD

MAX, MIN, 2*, 2/, 1+, 2+, 1- 2-

AND, OR, XOR, NOT

=, >, <, 0=, 0<

KEY, ECHO, CLEAR, CRETURN, SPACE, TYPE, DISPLAY

#: pop x, y=x mod base, chr(y), x div base

#S: führt solange # aus, bis stack[top] == 0

#>: zeigt die Zahl mittels DISPLAY an

.: gibt stack[top] aus

?: gibt *stack[top] aus

,: Speichert oberstes Stackelement auf Adresse des DP ab und erhöht DP

HERE: legt DP auf den Stack

?SP:legt Stackpointer auf den Stack

?RS: legt Returnstackpointer auf den Stack

TOKEN: Interpreter

´: sucht den token im Vokabular und gibt Adresse auf den Stack

ABORT, ASPACE, SEARCH

ENTRY: legt Adresse des zuletzt definierten Wertes auf den Stack

CA!: legt die Adresse, die auf dem Stack ist und speichert es an der WA des letzten Wortes

WAIT, FILL, ERASE, DUMP, ADUMP

## Kontrollstrukturen

BEGIN ... END Schleife

BEGIN ... flag END

flag IF ... ELSE THEN

flag IF ... THEN

BEGIN ... flag IF ... WHILE

BEGIN ... flag IF ... ELSE ... WHILE

end start DO ... LOOP

end start DO ... inc +LOOP

Schleifenindex steht auf dem Returnstack und wird um 1 oder inc erhöht

LEAVE: Schleifenexit für DO-Loops

switch mit Sprungtabelle implementieren

## Compiling Keywords

CREATE: stellt den nächsten token in den Dictionary

:: startet Compilemode, macht ein CREATE und setzt die Adresse von enter in den Dictionary

;: setzt Adresse von return in den Dictionary und beendet Compilermode

;CODE: setzt Adresse von SCODE in Dictionary und beendet Compilermode

Assembler
Screens
andere Dictionaryformate
module threaded code

21

22

# Pascal P4 System

model for most other Pascal systems (UCSD)
compiler generates assembly language P4 intermediate code
assembler/interpreter assembles and executes P4 code

advantages

- readable intermediate code
- resolving of forward references in single pass in interpreter
- portable system

problems

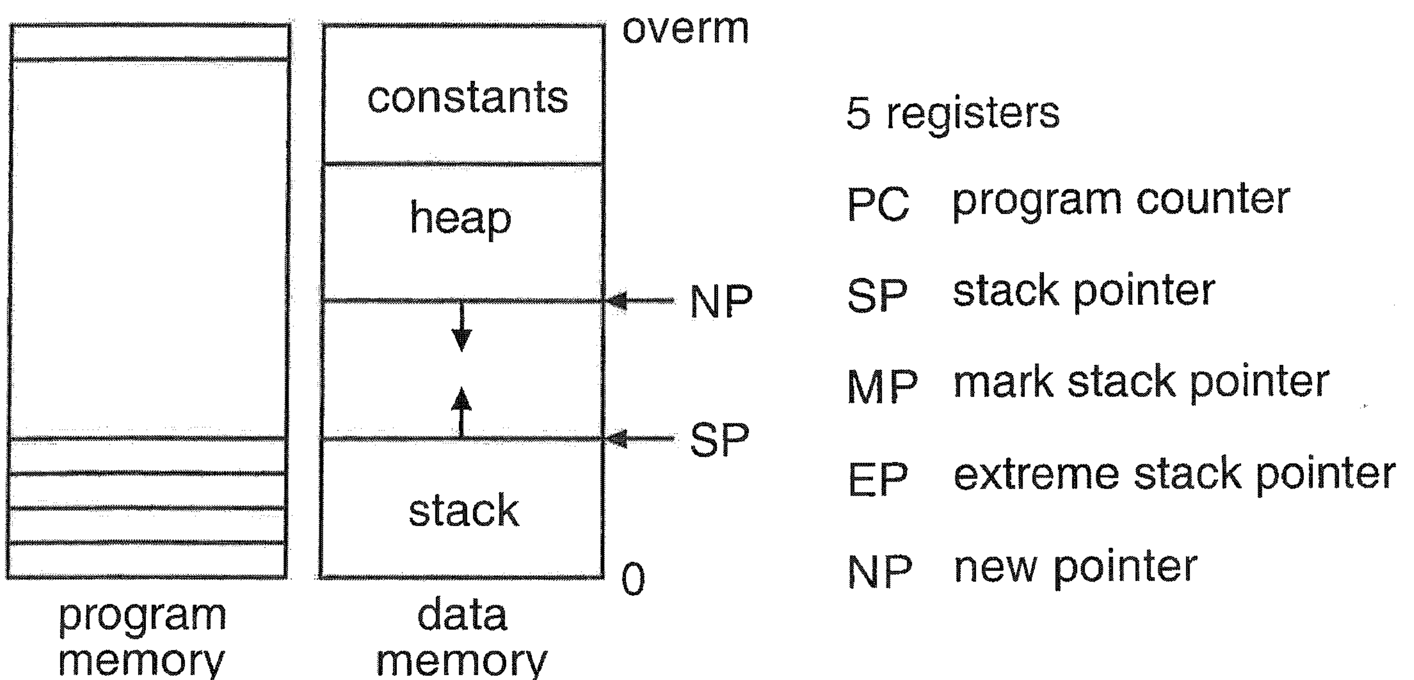
- very slow

possible improvements

- compiler generates binary P4 code
- direct threaded code interpreter or JIT compiler

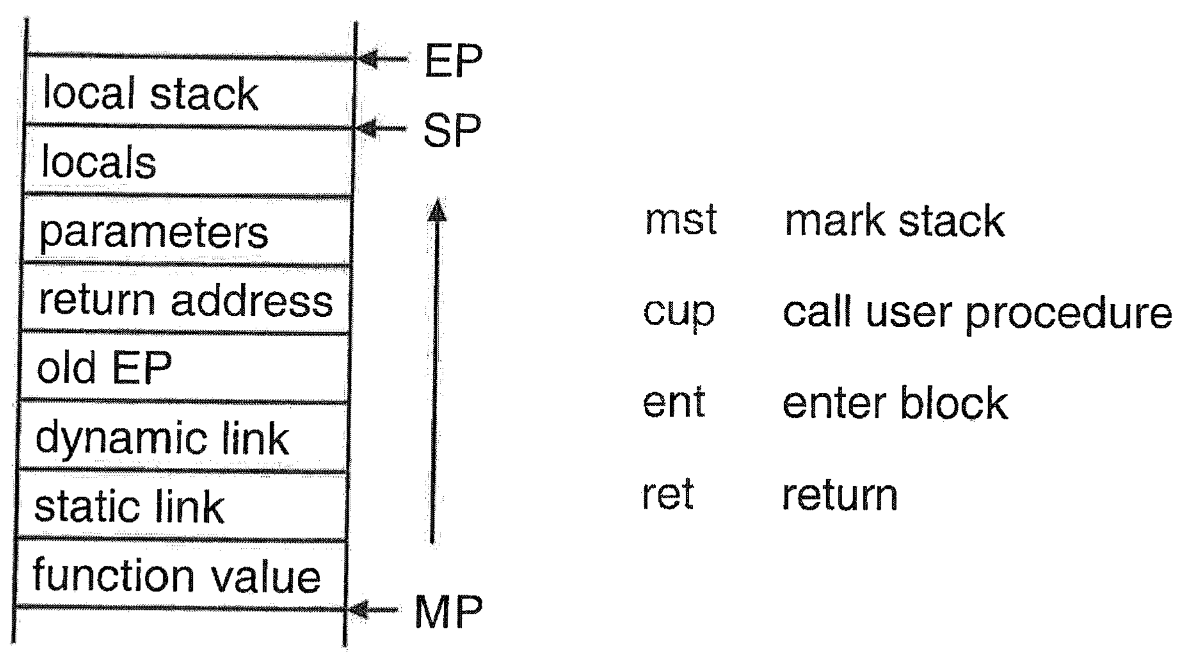


# The P4 Virtual Machine



23

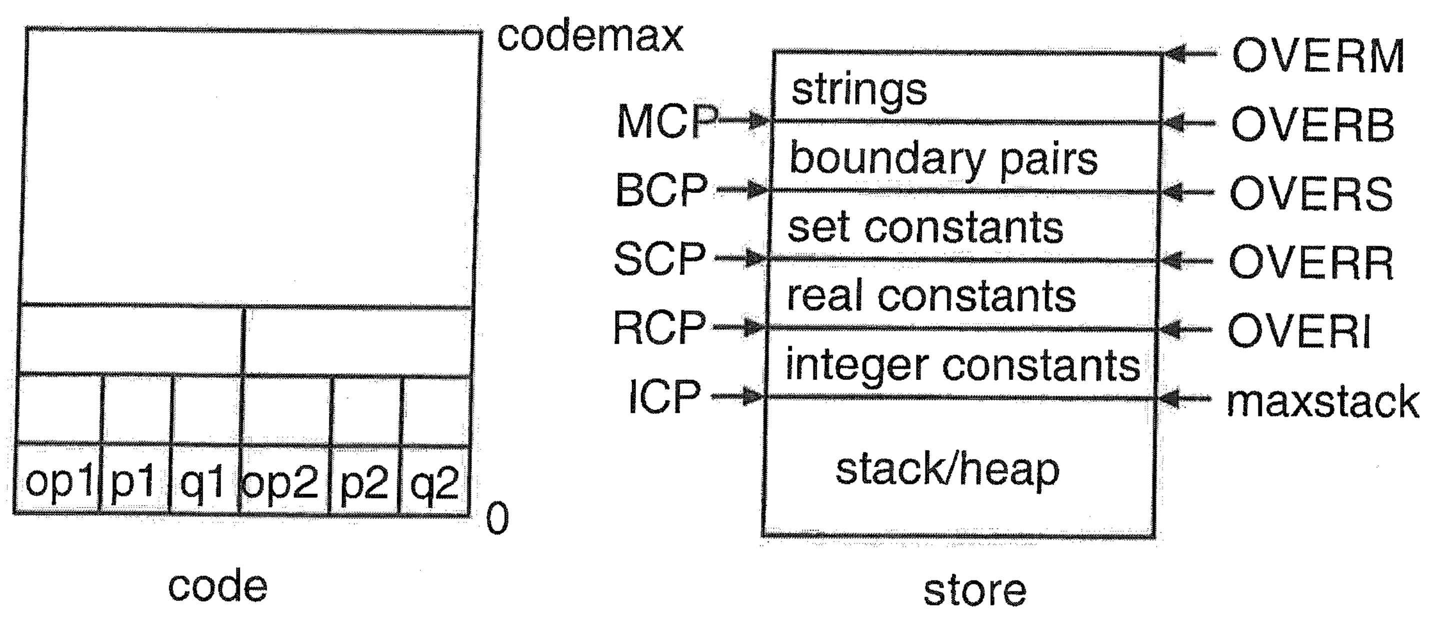
# Stack Frame





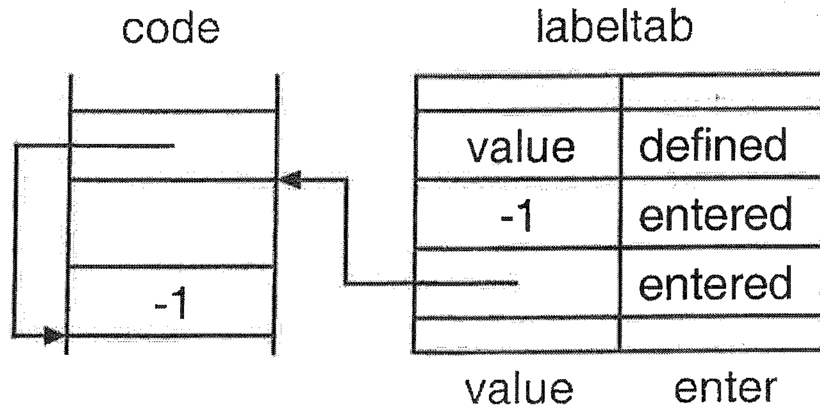

# Assembler/Interpreter

2 instructions with 2 operands are stored in one machine word



24

# Assembler




instruction names stored in linear table

multiple type instructions are translated into different instructions

identical constants are stored only once



# Interpreter

maximum of 4 files

subroutines for

- post mortem dump
- computation of base
- string comparison
- standard input/output procedures

instruction fetch

case statement

Thus the reference in a Pascal program to a file variable for example Input↑, will be a direct reference to the contents of one of these locations.

## P-CODE INSTRUCTION SET

The following table describes the complete instruction set showing the parameters and also the effect of the execution of each instruction on the stack. Only a brief description of each instruction is given here – a detailed version is given in Chapter 11.

| Instruction | Operation on stack Before | After | Parameters if present | Description of instruction |
|---|---|---|---|---|
| *abi* | (i) | i | | Absolute value of integer |
| *abr* | (r) | r | | Absolute value of real |
| *adi* | (i,i) | i | | Adds two integers on the top of the stack and leaves an integer result |
| *adr* | (r,r) | r | | Adds two reals on the top of the stack and leaves a real result |
| *chk* C | No change | | | Checks value is between upper and lower bounds |
| *chr* | (i) | c | | Converts integer to character |
| *csp* | special | | Q | Call standard procedure |
| *cup* | special | | PQ | Call user procedure |
| *dec* C | (x) | x | Q | Decrement |
| *dif* | (s,s) | s | | Set difference |
| *dvi* | (i,i) | i | | Integer division |
| *dvr* | (r,r) | r | | Real division |
| *ent* | special | | PQ | Enter block |
| *eof* | (a) | b | | Test on end of file |
| *equ* C | (x,x) | b | Q | Compare on equal |
| *fjp* | (b) | | | False jump |
| *flo* | (i,r) | r,r | | Float next to the top |
| *flt* | (i) | r | | Float top of the stack |
| *geq* C | (x,x) | b | Q | Compare on greater or equal |
| *inc* C | (x) | x | Q | Increment |
| *ind* C | (a) | x | Q | Indexed fetch |
| *inn* | (i,s) | b | | Test set membership |
| *int* | (s,s) | s | | Set intersection |
| *ior* | (b,b) | b | | Boolean inclusive OR |
| *ixa* | (a,i) | a | Q | Compute indexed address |
| *lao* | | a | Q | Load base-level address |
| *lca* | | a | Q | Load address of constant |
| *lci* | | x | PQ | Load constant indirect – assembler generated |
| *lda* | | a | PQ | Load address with level P |
| *ldc* C | | x | Q | Load constant |
| *ldo* C | | x | Q | Load contents of base-level address |
| *leq* C | (x,x) | b | Q | Compare on less than or equal |
| *les* C | (x,x) | b | Q | Compare on less than |
| *lod* C | | x | PQ | Load contents of address |
| *mod* | (i,i) | i | | Modulo |
| *mov* | (a,a) | | Q | Move |
| *mpi* | (i,i) | i | | Integer multiplication |
| *mpr* | (r,r) | r | | Real multiplication |



| Instruction | Operation on stack Before | After | Parameters if present | Description of instruction |
|---|---|---|---|---|
| *mst* | special | | P | Mark stack |
| *neq* C | (x,x) | b | Q | Compare on not equal |
| *ngi* | (i) | i | | Integer sign inversion |
| *ngr* | (r) | r | | Real sign inversion |
| *not* | (b) | b | | Boolean not |
| *odd* | (i) | b | | Test on odd |
| *ord* C | (x) | i | | Convert to integer |
| *ret* C | special | | | Return from block |
| *sbi* | (i,i) | i | | Integer subtraction |
| *sbr* | (r,r) | r | | Real subtraction |
| *sgs* | (i) | s | | Generate singleton set |
| *sqi* | (i) | i | | Square integer |
| *sqr* | (r) | r | | Square real |
| *sro* C | (x) | | Q | Store at base level address |
| *sto* C | (a,x) | | | Store at base level address |
| *stp* | No effect | | | Stop |
| *str* C | (x) | | PQ | Store at level P |
| *trc* | (r) | i | | Truncation |
| *ujc* | No effect | | | Error in case statement |
| *ujp* | No effect | | Q | Unconditional jump |
| *uni* | (s,s) | s | | Set union |
| *xjp* | (i) | | Q | Indexed jump |

Key to effect on stack:

| | |
|---|---|
| a | address |
| b | boolean |
| c | character |
| i | integer |
| r | real |
| s | set |
| x | any of the above types |

The C parameter denotes one of the primitive types.

# Assembler/Interpreter Listing

```
 1 (*Assembler and interpretor of Pascal code*)
 2 (*K. Jenson, N. Wirth, Ch. Jacobi, ETH May 76*)
 3
 4 program pcode(input,output,prd,prr);
 5
 6 (* Note for the implementation.
 7    ===========================
 8 This interpreter is written for the case where all the fundamental types
 9 take one storage unit.
10 In an actual implementation, the handling of the sp pointer has to take
11 into account the fact that the types may have lengths different from one:
12 in push and pop operations the sp has to be increased and decreased not
13 by 1, but by a number depending on the type concerned.
14 However, where the number of units of storage has been computed by the
15 compiler, the value must not be corrected, since the lengths of the types
16 involved have already been taken into account.
17
18
19                                                                       *)
20
21
22 label 1;
23 const codemax     = 8650;
24       pcmax       = 17500;
25       maxstk      = 13650; (* size of variable store *)
26       overi       = 13655; (* size of integer constant table = 5 *)
27       overr       = 13660; (* size of real constant table = 5 *)
28       overs       = 13730; (* size of set constant table = 70 *)
29       overb       = 13820;
30       overm       = 18000;
31       maxstr,     = 18001;
32       largeint    = 26144;
33       begincode   = 3;
34       inputadr    = 5;
35       outputadr   = 6;
36       prdadr      = 7;
37       prradr      = 8;
38       duminst     = 62;
39
40 type  bit4       =-0..15;
41       bit6       = 0..127;
42       bit20      = -26143..26143;
43       datatype   = (undef,int,real,bool,sett,adr,mark,car);
44       address    = -1..maxstr;
45       beta       = packed array[1..25] of char; (*error message*)
46
47 var   code       : array[0..codemax] of  (* the program *)
48                     packed record  op1   :bit6;
49                                    p1    :bit4;
50                                    q1    :bit20;
51                                    op2   :bit6;
52                                    p2    :bit4;
53                                    q2    :bit20
54                                end;
55       pc         : 0..pcmax;        (*program address register*)
56       op : bit6; p : bit4, q : bit20;   (*instruction ...
```



```
 953                   end;
 954
 955        46 (*int*): begin  sp := sp-1;
 956                     store[sp].vs := store[sp].vs * store[sp+1].vs
 957                    end;
 958
 959        47 (*uni*): begin  sp := sp-1;
 960                     store[sp].vs := store[sp].vs + store[sp+1].vs
 961                    end;
 962
 963        48 (*inn*): begin
 964                     sp := sp - 1; i := store[sp].vi;
 965                     store[sp].vb := i in store[sp+1].vs;
 966                    end;
 967
 968        49 (*mod*): begin  sp := sp-1;
 969                     store[sp].vi := store[sp].vi mod store[sp+1].vi
 970                    end;
 971
 972        50 (*odd*): store[sp].vb := odd(store[sp].vi);
 973
 974        51 (*mpi*): begin  sp := sp-1;
 975                     store[sp].vi := store[sp].vi * store[sp+1].vi
 976                    end;
 977
 978        52 (*mpr*): begin  sp := sp-1;
 979                     store[sp].vr := store[sp].vr * store[sp+1].vr
 980                    end;
 981
 982        53 (*dvi*): begin  sp := sp-1;
 983                     store[sp].vi := store[sp].vi div store[sp+1].vi
 984                    end;
 985
 986        54 (*dvr*): begin  sp := sp-1;
 987                     store[sp].vr := store[sp].vr / store[sp+1].vr
 988                    end;
 989
 990        55 (*mov*): begin i1 := store[sp-1].va;
 991                     i2 := store[sp].va; sp := sp-2;
 992                     for i := 0 to q-1 do store[i1+i] := store[i2+i]
 993                     (* q is a number of storage units *)
 994                    end;
 995
 996        56 (*lca*): begin  sp := sp+1;
 997                     store[sp].va := q;
 998                    end;
 999
1000        100,101,102,103,104,
1001        57 (*dec*): store[sp].vi := store[sp].vi-q;
1002
1003        58 (*stp*): interpreting := false;
1004
1005        59 (*ord*): (*only used to change the tagfield*)
1006                    begin
1007                    end;
1008
1009        60 (*chr*): begin
1010                    end;
1011
1012        61 (*ujc*): errori(' case - error           ');
1013     end
1014   end; (*while interpreting*)
1015
1016 1 :
```

```
57
58      store        : array [0..overm] of
59                         record case datatype of
60                               int         :(vi :integer),
61                               reel        :(vr :real),
62                               bool        :(vb :boolean),
63                               sett        :(vs :set of 0..47);
64                               car         :(vc :char);
65                               adr         :(va :address);
66                                              (*address in store*)
67                               mark        :(vm :integer)
68                         end;
69      mp,sp,np,ep : address.  (* address registers *)
70      (*mp  points to beginning of a data segment
71        sp  points to top of the stack
72        ep  points to the maximum extent of the stack
73        np  points to top of the dynamically allocated area*)
74
75      interpreting: boolean;
76      prd,prr      : text;(*prd for read only, prr for write only *)
77
78      instr        : array[bit6] of alfa; (* mnemonic instruction codes *)
79      cop          : array[bit6] of integer;
80      sptable      : array[0..20] of alfa; (*standard functions and procedures*)
81
82     (*locally used for interpreting one instruction*)
83      ad,adl       : address;
84      b            : boolean;
85      i,j,i1,i2    : integer;
86      c            : char;
87
88 (*--------------------------------------------------------------------------*)
89
90 procedure load;
91    const maxlabel = 1850;
92    type  labelst  = (entered,defined); (*label situation*)
93          labelrg  = 0..maxlabel;       (*label range*)
94          labelrec = record
95                          val: address;
96                           st: labelst
97                     end;
98    var  icp,rcp,scp,bcp,mcp  : address;  (*pointers to next free position*)
99         word : array[1..10] of char; i   integer;  ch   : char;
100        labeltab: array[labelrg] of labelrec;
101        labelvalue: address;
102
103   procedure init;
104     var i: integer;
105   begin instr[ 0]:='lod       ';       instr[ 1]:='ldo       ';
106         instr[ 2]:='str       ';       instr[ 3]:='sro       ';
107         instr[ 4]:='lda       ';       instr[ 5]:='lao       ';
108         instr[ 6]:='sto       ';       instr[ 7]:='ldc       ';
109         instr[ 8]:='...       ';       instr[ 9]:='ind       ';
110         instr[10]:='inc       ';       instr[11]:='mst       ';
111         instr[12]:='cup       ';       instr[13]:='ent       ';
112         instr[14]:='ret       ';       instr[15]:='csp       ';
113         instr[16]:='ixa       ';       instr[17]:='equ       ';
114         instr[18]:='neq       ';       instr[19]:='geq       ';
115         instr[20]:='grt       ';       instr[21]:='leq       ';
116         instr[22]:='les       ';       instr[23]:='ujp       ';
117         instr[24]:='fjp       ';       instr[25]:='xjp       ';
118         instr[26]:='chk       ';       instr[27]:='eof       ';
119         instr[28]:='adi       ';       instr[29]:='adr       ';
120         instr[30]:='sbi       ';       instr[31]:='sbr       ';
```



```
121         instr[32]:='sgs       ';       instr[33]:='flt       ';
122         instr[34]:='flo       ';       instr[35]:='trc       ';
123         instr[36]:='ngi       ';       instr[37]:='ngr       ';
124         instr[38]:='sqi       ';       instr[39]:='sqr       ';
125         instr[40]:='abi       ';       instr[41]:='abr       ';
126         instr[42]:='not       ';       instr[43]:='and       ';
127         instr[44]:='ior       ';       instr[45]:='dif       ';
128         instr[46]:='int       ';       instr[47]:='uni       ';
129         instr[48]:='inn       ';       instr[49]:='mod       ';
130         instr[50]:='odd       ';       instr[51]:='mpi       ';
131         instr[52]:='mpr       ';       instr[53]:='dvi       ';
132         instr[54]:='dvr       ';       instr[55]:='mov       ';
133         instr[56]:='lca       ';       instr[57]:='dec       ';
134         instr[58]:='stp       ';       instr[59]:='ord       ';
135         instr[60]:='chr       ';       instr[61]:='ujc       ';
136
137         sptable[ 0]:='get       ';     sptable[ 1]:='put       ';
138         sptable[ 2]:='rst       ';     sptable[ 3]:='rln       ';
139         sptable[ 4]:='new       ';     sptable[ 5]:='wln       ';
140         sptable[ 6]:='wrs       ';     sptable[ 7]:='eln       ';
141         sptable[ 8]:='wri       ';     sptable[ 9]:='wrr       ';
142         sptable[10]:='wrc       ';     sptable[11]:='rdi       ';
143         sptable[12]:='rdr       ';     sptable[13]:='rdc       ';
144         sptable[14]:='sin       ';     sptable[15]:='cos       ';
145         sptable[16]:='exp       ';     sptable[17]:='log       ';
146         sptable[18]:='sqt       ';     sptable[19]:='atn       ';
147         sptable[20]:='sav       ';
148
149         cop[ 0] := 105;  cop[ 1] :=  65;
150         cop[ 2] :=  70;  cop[ 3] :=  75;
151         cop[ 6] :=  80;  cop[ 9] :=  85;
152         cop[10] :=  90;  cop[26] :=  95;
153         cop[57] := 100;
154
155         pc  := begincode;
156         icp := maxstk + 1;
157         rcp := overi + 1;
158         scp := overr + 1;
159         bcp := overs + 2;
160         mcp := overb + 1;
161         for i:= 1 to 10 do word[i]:= ' ';
162         for i:= 0 to maxlabel do
163             with labeltab[i] do begin val:=-1; st:= entered end;
164         reset(prd);
165   end;(*init*)
166
167   procedure errorl(string: beta); (*error in loading*)
168   begin writeln;
169      write(string);
170      halt
171   end; (*errorl*)
172
173   procedure update(x: labelrg); (*when a label definition lx is found*)
174      var curr,succ: -1..pcmax;  (*resp. current element and successor element
175                                    of a list of future references*)
176          endlist: boolean;
177   begin
178      if labeltab[x].st=defined then errorl(' duplicated label         ')
179      else begin
180             if labeltab[x].val<>-1 then (*forward reference(s)*)
181             begin curr:= labeltab[x].val; endlist:= false;
182                while not endlist do
183                      with code[curr div 2] do
184                      begin
```

```
185                     if odd(curr) then begin succ:= q2;
186                                             q2:= labelvalue
187                                       end
188                                 else begin succ:= q1;
189                                             q1:= labelvalue
190                                       end;
191                     if succ=-1 then endlist:= true
192                                else curr:= succ
193                  end;
194            end;
195            labeltab[x].st := defined;
196            labeltab[x].val:= labelvalue;
197         end
198   end,(*update*)
199
200   procedure assemble; forward;
201
202   procedure generate;(*generate segment of code*)
203      var x: integer; (* label number *)
204         again: boolean;
205   begin
206      again := true;
207      while again do
208            begin read(prd,ch),(* first character of line*)
209                  case ch of
210                       'i': readln(prd),
211                       'l': begin read(prd,x);
212                                  if not eoln(prd) then read(prd,ch);
213                                  if ch='=' then read(prd,labelvalue)
214                                            else labelvalue:= pc;
215                                  update(x); readln(prd);
216                            end;
217                       'q': begin again := false; readln(prd) end;
218                       ' ': begin read(prd,ch); assemble end
219                  end;
220            end
221   end; (*generate*)
222
223   procedure assemble; (*translate symbolic code into machine code and store*)
224      label 1;         (*goto 1 for instructions without code generation*)
225      var name :alfa;  b :boolean;  r :real;  s :set of 0..58;
226          cl :char;  i,sl,lb,ub :integer;
227
228      procedure lookup(x: labelrg); (* search in label table*)
229      begin case labeltab[x].st of
230                entered: begin q := labeltab[x].val;
231                               labeltab[x].val := pc
232                         end;
233                defined: q:= labeltab[x].val
234         end(*case label..*)
235      end;(*lookup*)
236
237      procedure labelsearch;
238         var x: labelrg;
239      begin while (ch<>'l') and not eoln(prd) do read(prd,ch);
240            read(prd,x); lookup(x)
241      end;(*labelsearch*)
242
243      procedure getname;
244      begin  word[1] := ch;
245         read(prd,word[2],word[3]);
246         if not eoln(prd) then read(prd,ch) (*next character*);
247         pack(word,1,name)
248      end; (*getname*)
```



```
249
250      procedure typesymbol;
251        var i: integer;
252      begin
253        if ch <> 'i' then
254          begin
255            case ch of
256              'a': i := 0;
257              'r': i := 1;
258              's': i := 2;
259              'b': i := 3;
260              'c': i := 4;
261            end;
262            op := cop[op]+i;
263          end;
264      end (*typesymbol*) ;
265
266   begin  p := 0;  q := 0;  op := 0;
267      getname;
268      instr[duminst] := name;
269      while instr[op]<>name do op := op+1;
270      if op = duminst then errorl(' illegal instruction     ');
271
272      case op of  (* get parameters p,q *)
273
274         (*equ,neq,geq,grt,leq,les*)
275         17,18,19,
276         20,21,22: begin case ch of
277                             'a': ; (*p = 0*)
278                             'i': p := 1;
279                             'r': p := 2;
280                             'b': p := 3;
281                             's': p := 4;
282                             'c': p := 6;
283                             'm': begin p := 5;
284                                        read(prd,q)
285                                  end
286                         end
287                   end;
288
289         (*lod,str*)
290         0,2: begin typesymbol; read(prd,p,q)
291              end;
292
293         4  (*lda*): read(prd,p,q);
294
295         12 (*cup*): begin read(prd,p); labelsearch end;
296
297         11 (*mst*): read(prd,p);
298
299         14 (*ret*): case ch of
300                          'p': p:=0;
301                          'i': p:=1;
302                          'r': p:=2;
303                          'c': p:=3;
304                          'b': p:=4;
305                          'a': p:=5
306                     end;
307
308         (*lao,ixa,mov*)
309         5,16,55: read(prd,q);
310
311         (*ldo,sro,ind,inc,dec*)
312         1,3,9,10,57: begin typesymbol; read(prd,q)
```

```
313                  end;
314
315      (*ujp,fjp,xjp*)
316      23,24,25: labelsearch;
317
318      13 (*ent*): begin read(prd,p), labelsearch end;
319
320      15 (*csp*): begin for i:=1 to 9 do read(prd,ch); getname;
321                       while name<>sptable[q] do  q := q+1
322                  end;
323
324      7 (*ldc*): begin case ch of  (*get q*)
325                   'i': begin  p := 1;  read(prd,i);
326                          if abs(i)>=largeint then
327                          begin  op := 8;
328                            store[icp].vi := i;  q := maxstk;
329                            repeat  q := q+1  until store[q].vi=i;
330                            if q=icp then
331                            begin  icp := icp+1;
332                              if icp=overi then
333                                errorl(' integer table overflow  ');
334                            end
335                          end  else q := 1
336                        end;
337
338                   'r': begin  op := 8; p := 2;
339                          read(prd,r);
340                          store[rcp].vr := r;  q := overi;
341                          repeat  q := q+1  until store[q].vr=r;
342                          if q=rcp then
343                          begin  rcp := rcp+1;
344                            if rcp = overr then
345                              errorl(' real table overflow     ');
346                          end
347                        end;
348
349                   'n': ; (*p,q = 0*)
350
351                   'b': begin p := 3;  read(prd,q)  end;
352
353                   'c': begin p := 6;
354                          repeat read(prd,ch); until ch <> ' ';
355                          if ch <> '''' then
356                            errorl(' illegal character       ');
357                          read(prd,ch);  q := ord(ch);
358                          read(prd,ch);
359                          if ch <> '''' then
360                            errorl(' illegal character       ');
361                        end;
362                   '(': begin  op := 8;  p := 4;
363                          s := [ ];  read(prd,ch),
364                          while ch<>')' do
365                          begin read(prd,sl,ch); s := s + [sl]
366                          end;
367                          store[scp].vs := s;  q := overr;
368                          repeat  q := q+1  until store[q].vs=s;
369                          if q=scp then
370                          begin  scp := scp+1;
371                            if scp=overs then
372                              errorl(' set table overflow      ');
373                          end
374                        end
375                 end (*case*)
376            end;
377
378      26 (*chk*): begin typesymbol;
379                    read(prd,lb,ub);
380                    if op = 95 then q := lb
381                    else
382                    begin
383                      store[bcp-1].vi := lb; store[bcp].vi := ub;
384                      q := overs;
385                      repeat  q := q+2
386                      until (store[q-1].vi=lb)and (store[q].vi=ub).
387                      if q=bcp then
388                      begin  bcp := bcp+2;
389                        if bcp=overb then
390                          errorl(' boundary table overflow ');
391                      end
392                    end
393                  end;
394
395      56 (*lca*): begin
396                    if mcp + 16 >= overm then
397                      errorl(' multiple table overflow '),
398                    mcp := mcp+16;
399                    q := mcp;
400                    for i := 0 to 15 (*stringlgth*) do
401                    begin read(prd,ch);
402                      store[q+i].vc := ch
403                    end;
404                  end;
405
406      6 (*sto*): typesymbol;
407
408      27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
409      48,49,50,51,52,53,54,58:  ;
410
411      (*ord,chr*)
412      59,60: goto 1;
413
414      61 (*ujc*): ; (*must have same length as ujp*)
415
416    end; (*case*)
417
418    (* store instruction *)
419    with code[pc div 2] do
420      if odd(pc) then
421      begin  op2 := op; p2 := p; q2 := q
422      end  else
423      begin  op1 := op; p1 := p; q1 := q
424      end;
425    pc := pc+1;
426    1: readln(prd);
427  end; (*assemble*)
428
429 begin (*load*)
430   init;
431   generate;
432   pc := 0;
433   generate;
434 end; (*load*)
435
436 (*--------------------------------------------------------------------*)
437
438 procedure pmd;
439   var s :integer; i: integer;
440
```

```
569                   8: errori(' readln on prr file      ')
570                end;
571                sp:= sp-1
572             end;
573    4 (*new*): begin ad:= np-store[sp].va;
574             (*top of stack gives the length in units of storage *)
575                if ad <= ep then
576                  errori(' store overflow          ');
577                np:= ad; ad:= store[sp-1].va;
578                store[ad].va := np;
579                sp:=sp-2
580             end;
581    5 (*wln*): begin case store[sp].va of
582                      5: errori(' writeln on input file   '),
583                      6: writeln(output);
584                      7: errori(' writeln on prd file     ');
585                      8: writeln(prr)
586                end;
587                sp:= sp-1
588             end;
589    6 (*wrs*): case store[sp].va of
590                   5: errori(' write on input file     ');
591                   6: writestr(output);
592                   7: errori(' write on prd file       ');
593                   8: writestr(prr)
594             end;
595    7 (*eln*): begin case store[sp].va of
596                      5: line:= eoln(input);
597                      6: errori(' eoln output file        '),
598                      7: line:=eoln(prd);
599                      8: errori(' eoln on prr file        ')
600                end;
601                store[sp].vb := line
602             end;
603    8 (*wri*): begin case store[sp].va of
604                      5: errori(' write on input file     ');
605                      6: write(output,
606                           store[sp-2].vi: store[sp-1].vi);
607                      7: errori(' write on prd file       ');
608                      8: write(prr,
609                           store[sp-2].vi: store[sp-1].vi)
610                end;
611                sp:=sp-3
612             end;
613    9 (*wrr*): begin case store[sp].va of
614                      5: errori(' write on input file     '),
615                      6: write(output,
616                           store[sp-2].vr: store[sp-1].vi);
617                      7: errori(' write on prd file       ');
618                      8: write(prr,
619                           store[sp-2].vr: store[sp-1].vi)
620                end;
621                sp:=sp-3
622             end;
623   10(*wrc*): begin case store[sp].va of
624                      5: errori(' write on input file     ');
625                      6: write(output,store[sp-2].vc:
626                           store[sp-1].vi);
627                      7: errori(' write on prd file       ');
628                      8: write(prr,chr(store[sp-2].vi):
629                           store[sp-1].vi);
630                end;
631                sp.=sp-3
632             end;
```



```
633        11(*rdi*): case store[sp].va of
634                       5: readi(input),
635                       6: errori(' read on output file     ');
636                       7: readi(prd),
637                       8: errori(' read on prr file        ')
638                    end;
639        12(*rdr*): case store[sp].va of
640                       5: readr(input);
641                       6: errori(' read on output file     '),
642                       7: readr(prd),
643                       8: errori(' read on prr file        ')
644                    end;
645        13(*rdc*): case store[sp].va of
646                       5: readc(input);
647                       6: errori(' read on output file     '),
648                       7: readc(prd);
649                       8: errori(' read on prr file        ')
650                    end;
651        14(*sin*): store[sp].vr:= sin(store[sp].vr);
652        15(*cos*): store[sp].vr:= cos(store[sp].vr);
653        16(*exp*): store[sp].vr:= exp(store[sp].vr),
654        17(*log*): store[sp].vr:= ln(store[sp].vr);
655        18(*sqt*): store[sp].vr:= sqrt(store[sp].vr);
656        19(*atn*): store[sp].vr:= arctan(store[sp].vr);
657        20(*sav*): begin ad:=store[sp].va;
658                      store[ad].va := np;
659                      sp:= sp-1
660                   end;
661      end;(*case q*)
662 end;(*callsp*)
663
664 begin (* main *)
665   rewrite(prr);
666   load; (* assembles and stores code *)
667   writeln(output); (* for testing *)
668   pc := 0; sp := -1; mp := 0; np := maxstk+1; ep := 5;
669   store[inputadr].vc := input^;
670   store[prdadr].vc := prd^;
671   interpreting := true;
672
673   while interpreting do
674   begin
675     (*fetch*)
676     with code[pc div 2] do
677       if odd(pc) then
678       begin op := op2; p := p2, q := q2
679       end else
680       begin op := op1; p := p1; q := q1
681       end;
682     pc := pc+1;
683
684     (*execute*)
685     case op of
686
687         105,106,107,108,109,
688         0 (*lod*): begin  ad := base(p) + q;
689                          sp := sp+1;
690                          store[sp] := store[ad]
691                       end;
692
693         65,66,67,68,69,
694         1 (*ldo*): begin
695                          sp := sp+1;
696                          store[sp] := store[q]
```

```
441   procedure pt;
442   begin  write(s:6);
443      if abs(store[s].vi) < maxint then write(store[s].vi)
444      else write('too big ');
445      s := s - 1;
446      i := i + 1;
447      if i = 4 then
448         begin writeln(output); i := 0 end;
449   end; (*pt*)
450
451 begin
452   write(' pc =',pc-1:5,' op =',op:3,'  sp =',sp:5,'  mp =',mp:5,
453         '  np =',np:5);
454   writeln; writeln('--------------------------------------');
455
456   s := sp; i := 0;
457   while s>=0 do pt;
458   s := maxstk;
459   while s>=np do pt;
460 end; (*pmd*)
461
462 procedure errori(string: beta);
463 begin writeln; writeln(string);
464       pmd; goto 1
465 end;(*errori*)
466
467 function base(ld :integer):address;
468    var ad :address;
469 begin  ad := mp;
470    while ld>0 do
471    begin  ad := store[ad+1].vm;  ld := ld-1
472    end;
473    base := ad
474 end; (*base*)
475
476 procedure compare;
477 (*comparing is only correct if result by comparing integers will be*)
478 begin
479   i1 := store[sp].va;
480   i2 := store[sp+1].va;
481   i := 0; b := true;
482   while b and (i<>q) do
483     if store[i1+i].vi = store[i2+i].vi then i := i+1
484     else b := false
485 end; (*compare*)
486
487 procedure callsp;
488    var line: boolean; adptr,adelnt: address;
489        i: integer;
490
491    procedure readi(var f:text);
492       var ad: address;
493    begin ad:= store[sp-1].va;
494          read(f,store[ad].vi);
495          store[store[sp].va].vc := f^;
496          sp:= sp-2
497    end;(*readi*)
498
499    procedure readr(var f: text);
500       var ad: address;
501    begin ad:= store[sp-1].va;
502          read(f,store[ad].vr);
503          store[store[sp].va].vc := f^;
504          sp:= sp-2
```



```
505    end;(*readr*)
506
507    procedure readc(var f: text);
508       var c: char; ad: address;
509    begin read(f,c);
510          ad:= store[sp-1].va;
511          store[ad].vc := c;
512          store[store[sp].va].vc := f^;
513          store[store[sp].va].vi := ord(f^);
514          sp:= sp-2
515    end;(*readc*)
516
517    procedure writestr(var f: text);
518       var i,j,k: integer;
519           ad: address;
520    begin ad:= store[sp-3].va;
521          k := store[sp-2].vi; j := store[sp-1].vi;
522          (* j and k are numbers of characters *)
523          if k>j then for i:=1 to k-j do write(f,' ')
524                 else j:= k;
525          for i := 0 to j-1 do write(f,store[ad+i].vc);
526          sp:= sp-4
527    end;(*writestr*)
528
529    procedure getfile(var f: text);
530       var ad: address;
531    begin ad:=store[sp].va;
532          get(f), store[ad].vc := f^;
533          sp:=sp-1
534    end;(*getfile*)
535
536    procedure putfile(var f: text);
537       var ad: address;
538    begin ad:= store[sp].va;
539          f^:= store[ad].vc; put(f);
540          sp:= sp-1;
541    end;(*putfile*)
542
543 begin (*callsp*)
544       case q of
545            0 (*get*): case store[sp].va of
546                            5: getfile(input);
547                            6: errori(' get on output file      ');
548                            7: getfile(prd);
549                            8: errori(' get on prr file         ')
550                       end;
551            1 (*put*): case store[sp].va of
552                            5: errori(' put on read file        ');
553                            6: putfile(output);
554                            7: errori(' put on prd file         ');
555                            8: putfile(prr)
556                       end;
557            2 (*rst*): begin
558                         (*for testphase*)
559                         np := store[sp].va; sp := sp-1
560                       end;
561            3 (*rln*): begin case store[sp].va of
562                            5: begin readln(input);
563                                  store[inputadr].vc := input^
564                               end;
565                            6: errori(' readln on output file   ');
566                            7: begin readln(input);
567                                  store[inputadr].vc := input^
568                               end;
```

```
697                    end;
698
699        70,71,72,73,74,
700        2 (*str*): begin  store[base(p)+q] := store[sp];
701                    sp := sp-1
702                   end;
703
704        75,76,77,78,79,
705        3 (*sro*): begin  store[q] := store[sp];
706                    sp := sp-1
707                   end;
708
709        4 (*lda*): begin sp := sp+1;
710                    store[sp].va := base(p) + q
711                   end;
712
713        5 (*lao*): begin sp := sp+1;
714                    store[sp].va := q
715                   end;
716
717        80,81,82,83,84,
718        6 (*sto*): begin
719                    store[store[sp-1].va] := store[sp];
720                    sp := sp-2;
721                   end;
722
723        7 (*ldc*): begin sp := sp+1;
724                    if p=1 then
725                    begin store[sp].vi := q;
726                    end else
727                        if p = 6 then store[sp].vc := chr(q)
728                        else
729                          if p = 3 then store[sp].vb := q = 1
730                          else (* load nil *) store[sp].va := maxstr
731                   end;
732
733        8 (*lci*): begin sp := sp+1;
734                    store[sp] := store[q]
735                   end;
736
737        85,86,87,88,89,
738        9 (*ind*): begin ad := store[sp].va + q;
739                    (* q is a number of storage units *)
740                    store[sp] := store[ad]
741                   end;
742
743        90,91,92,93,94,
744        10 (*inc*): store[sp].vi := store[sp].vi+q;
745
746        11 (*mst*): begin (*p=level of calling procedure minus level of called
747                            procedure + 1;  set dl and sl, increment sp*)
748                     (* then length of this element is
749                        max(intsize,realsize,boolsize,charsize,ptrsize *)
750                     store[sp+2].vm := base(p);
751                     (* the length of this element is ptrsize *)
752                     store[sp+3].vm := mp;
753                     (* idem *)
754                     store[sp+4].vm := ep;
755                     (* idem *)
756                     sp := sp+5
757                    end;
758
759        12 (*cup*): begin (*p=no of locations for parameters, q=entry point*)
760                     mp := sp-(p+4);
```



```
761                  store[mp+4].vm := pc;
762                  pc := q
763                 end;
764
765     13 (*ent*): if p = 1 then
766                  begin sp := mp + q; (*q = length of dataseg*)
767                    if sp > np then errori(' store overflow          ');
768                  end
769                 else
770                  begin ep := sp+q;
771                    if ep > np then errori(' store overflow          ');
772                  end;
773                  (*q = max space required on stack*)
774
775     14 (*ret*): begin case p of
776                          0: sp:= mp-1;
777                          1,2,3,4,5: sp:= mp
778                       end;
779                       pc := store[mp+4].vm;
780                       ep := store[mp+3].vm;
781                       mp:= store[mp+2].vm;
782                 end;
783
784     15 (*csp*): callsp;
785
786     16 (*ixa*): begin
787                  i := store[sp].vi;
788                  sp := sp-1;
789                  store[sp].va := q*i+store[sp].va;
790                 end;
791
792     17 (*equ*): begin  sp := sp-1;
793                  case p of
794                    1: store[sp].vb := store[sp].vi = store[sp+1].vi;
795                    0: store[sp].vb := store[sp].va = store[sp+1].va;
796                    6: store[sp].vb := store[sp].vc = store[sp+1].vc;
797                    2: store[sp].vb := store[sp].vr = store[sp+1].vr;
798                    3: store[sp].vb := store[sp].vb = store[sp+1].vb;
799                    4: store[sp].vb := store[sp].vs = store[sp+1].vs;
800                    5: begin  compare;
801                         store[sp].vb := b;
802                       end;
803                  end; (*case p*)
804                 end;
805
806     18 (*neq*): begin  sp := sp-1,
807                  case p of
808                    0: store[sp].vb := store[sp].va <> store[sp+1].va;
809                    1: store[sp].vb := store[sp].vi <> store[sp+1].vi;
810                    6: store[sp].vb := store[sp].vc <> store[sp+1].vc;
811                    2: store[sp].vb := store[sp].vr <> store[sp+1].vr;
812                    3: store[sp].vb := store[sp].vb <> store[sp+1].vb;
813                    4: store[sp].vb := store[sp].vs <> store[sp+1].vs;
814                    5: begin  compare;
815                         store[sp].vb := not b;
816                       end
817                  end; (*case p*)
818                 end;
819
820     19 (*geq*): begin  sp := sp-1;
821                  case p of
822                    0: errori(' <,<=,>,>= for address   ');
823                    1: store[sp].vb := store[sp].vi >= store[sp+1].vi;
824                    6: store[sp].vb := store[sp].vc >= store[sp+1].vc;
```

```
825                      2: store[sp].vb := store[sp].vr >= store[sp+1].vr;
826                      3: store[sp].vb := store[sp].vb >= store[sp+1].vb;
827                      4: store[sp].vb := store[sp].vs >= store[sp+1]]vs;
828                      5: begin compare;
829                           store[sp].vb := b or
830                             (store[i1+i].vi >= store[i2+i].vi)
831                         end
832                 end; (*case p*)
833               end;
834
835   20 (*grt*): begin  sp := sp-1;
836                 case p of
837                   0: errori(' <,<=,>,>= for address   ');
838                   1: store[sp].vb := store[sp].vi > store[sp+1].vi;
839                   6: store[sp].vb := store[sp].vc > store[sp+1].vc;
840                   2: store[sp].vb := store[sp].vr > store[sp+1].vr;
841                   3: store[sp].vb := store[sp].vb > store[sp+1].vb;
842                   4: errori(' set inclusion           ');
843                   5: begin  compare;
844                        store[sp].vb := not b and
845                          (store[i1+i].vi > store[i2+i].vi)
846                      end
847                 end; (*case p*)
848               end;
849
850   21 (*leq*): begin  sp := sp-1;
851                 case p of
852                   0: errori(' <,<=,>,>= for address   ');
853                   1: store[sp].vb := store[sp].vi <= store[sp+1].vi;
854                   6: store[sp].vb := store[sp].vc <= store[sp+1].vc;
855                   2: store[sp].vb := store[sp].vr <= store[sp+1].vr;
856                   3: store[sp].vb := store[sp].vb <= store[sp+1].vb;
857                   4: store[sp].vb := store[sp].vs <= store[sp+1].vs,
858                   5: begin  compare;
859                        store[sp].vb := b or
860                          (store[i1+i].vi <= store[i2+i].vi)
861                      end;
862                 end; (*case p*)
863               end;
864
865   22 (*les*): begin  sp := sp-1;
866                 case p of
867                   0: errori(' <,<=,>,>= for address   ');
868                   1: store[sp].vb := store[sp].vi < store[sp+1].vi;
869                   6: store[sp].vb := store[sp].vc < store[sp+1].vc;
870                   2: store[sp].vb := store[sp].vr < store[sp+1].vr;
871                   3: store[sp].vb := store[sp].vb < store[sp+1].vb;
872                   5: begin  compare;
873                        store[sp].vb := not b and
874                          (store[i1+i].vi < store[i2+i].vi)
875                      end
876                 end; (*case p*)
877               end;
878
879   23 (*ujp*): pc := q;
880
881   24 (*fjp*): begin  if not store[sp].vb then pc := q;
882                 sp := sp-1
883               end;
884
885   25 (*xjp*): begin
886                 pc := store[sp].vi + q;
887                 sp := sp-1
888               end;
```



```
889
890   95 (*chka*): if (store[sp].va < np) or
891                   (store[sp].va > (maxstr-q)) then
892                  errori(' bad pointer value        ');
893
894   96,97,98,99,
895   26 (*chk*): if (store[sp].vi < store[q-1].vi) or
896                  (store[sp].vi > store[q].vi) then
897                 errori(' value out of range       ');
898
899   27 (*eof*): begin  i := store[sp].vi;
900                 if i=inputadr then
901                 begin store[sp].vb := eof(input),
902                 end else errori(' code in error          ');
903               end;
904
905   28 (*adi*): begin  sp := sp-1;
906                 store[sp].vi := store[sp].vi + store[sp+1].vi
907               end;
908
909   29 (*adr*): begin  sp := sp-1;
910                 store[sp].vr := store[sp].vr + store[sp+1].vr
911               end;
912
913   30 (*sbi*): begin sp := sp-1;
914                 store[sp].vi := store[sp].vi - store[sp+1].vi
915               end;
916
917   31 (*sbr*): begin  sp := sp-1;
918                 store[sp].vr := store[sp].vr - store[sp+1].vr
919               end;
920
921   32 (*sgs*): store[sp].vs := [store[sp].vi];
922
923   33 (*flt*): store[sp].vr := store[sp].vi;
924
925   34 (*flo*): store[sp-1].vr := store[sp-1].vi;
926
927   35 (*trc*): store[sp].vi := trunc(store[sp].vr);
928
929   36 (*ngi*): store[sp].vi := -store[sp].vi;
930
931   37 (*ngr*): store[sp].vr := -store[sp].vr;
932
933   38 (*sqi*): store[sp].vi := sqr(store[sp].vi);
934
935   39 (*sqr*): store[sp].vr := sqr(store[sp].vr);
936
937   40 (*abi*): store[sp].vi := abs(store[sp].vi);
938
939   41 (*abr*): store[sp].vr := abs(store[sp].vr);
940
941   42 (*not*): store[sp].vb := not store[sp].vb;
942
943   43 (*and*): begin  sp := sp-1;
944                 store[sp].vb := store[sp].vb and store[sp+1].vb
945               end;
946
947   44 (*ior*): begin  sp := sp-1;
948                 store[sp].vb := store[sp].vb or store[sp+1].vb
949               end;
950
951   45 (*dif*): begin  sp := sp-1;
952                 store[sp].vs := store[sp].vs - store[sp+1].vs
```

34

# Pascal P4 Compiler

single pass compiler

control part: syntactic analysis calls lexical analysis (insymbol), semantic check and code generation

generated code: assembly language source

about 4000 lines of Pascal code

portable through constant definitions



# Lexical Analysis

program driven lexical analysis (main routine: insymbol)

determines identifiers, keywords, numbers and other symbols

skips comments (option recognition)

output of source and error messages

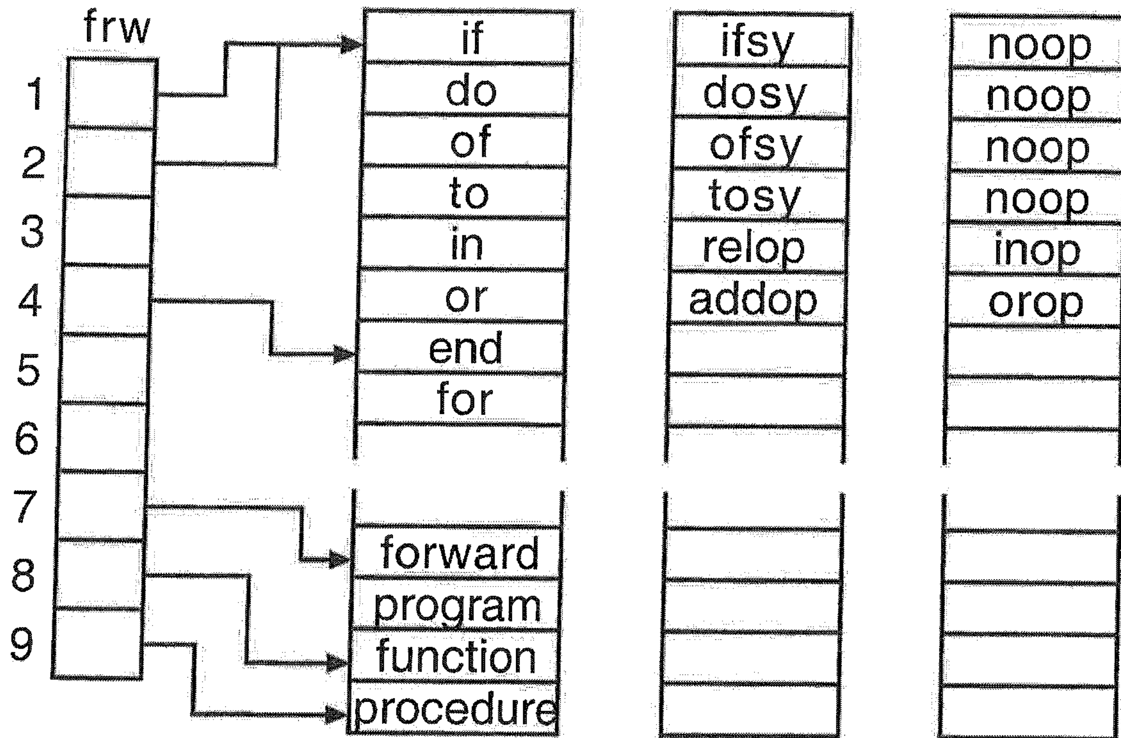
spaces in identifiers are added economically

storate of constants

integer computation can cause overflow

35

# Tables of Lexical Analysis





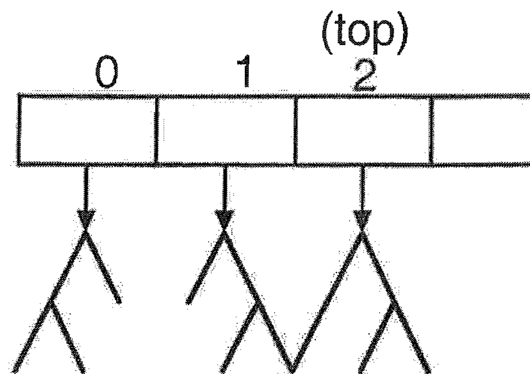

# Syntax Analysis

program driven: recursive descent parser

```
procedure whilestatement;
begin
   expressioin (fsys+[dosy]);
   if sy = dosy
   then insymbol
   else error (54);
   statement(fsys)
end;
```

skip skips symbol until continuation is possible

# Semantic Analysis




`enterid`, `searchid`, `searchsection`, `getbounds`, `equalbounds`, `comptypes`

no endless recursion for cyclic date structures (pointers)



# Code Generation

`gen0`, `gen1`, `gen2`, `gen0t`, `gen1t`, `gen2t`

generate code for 0, 1 or 2 parameters with or without types

`mes`: computes maximum step depth

`genfjp`, `genujpxjp`, `gencupent`: branch switch and procedure call

`alignquot`, `align`: address computations

`load`, `store`, `loadaddress`: operand loads and stores

`checkbnds`: checks bounds

`genlabel`, `putlabel`: generation of labels

## TYPES

Types are represented internally by the type *structure* [118-32].

All types use the *size* field, which contains the run-time store-size needed to hold an object of that type. (The *marked* field is used only by the procedure *printtables* [676-845] when printing out the compiler tables, if the *t* option is switched on.)

All other fields depend on the *form* of the type, if it is a pointer, or an array, and so on.

### Scalar

A scalar type is either *declared*, that is, an enumeration, in which case *fconst* points to the last identifier in the list (they are linked together by their *next* field) [1061]; or it is *standard*, when the type is *integer*, *real*, or *char* (*boolean* is *declared*). These latter four can be distinguished by comparing the pointer value to the *structure* with one of the four pointers *intptr*, *realptr*, *charptr*, *boolptr*, which are initialised [3646 *et seq.*] (see for example [652-7]).

### Subrange

Here *rangetype* points to the type of which this is a subrange (for example, *integer* in *1 . . 10*); and *min* and *max* hold the minimum and maximum values (*1* and *10* in the above case).

### Pointer

*Eltype* points to the type pointed at (*integer* in↑*integer*).

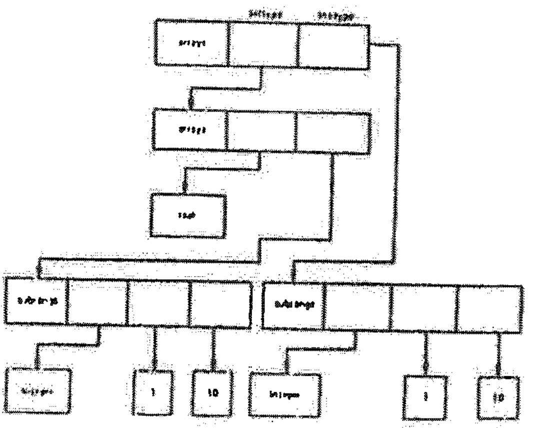
### Power

For sets, *elset* points to the element type of the set (for example, *char* in set of *char*).

### Arrays

*Inxtype* points to the index type of the array and *aeltype* to the element type, (e.g. *char* and *integer* respectively in array[*char*] of *integer*).

A multi-dimensional array, like array [*1. .10, 1. .10*] of *real* is treated identically to array [*1. .10*] of array [*1. .10*] of *real* so here would give





### Files

*Filtype* points to the file type (for example *integer* in file of *integer*).

### Records

*Fstfld* points to the first field of the record, the other fields being linked to the binary tree described before.

*Recvar* points to a *structure* of form *tagfld* representing the variant part of the record (it is nil if there is no variant part).

A *tagfld* has two fields: *tagfldp* points to the *identifier* for the tag field, and *fstvar* points to a list of *structures* of form *variant*, each representing one of the case labels.

As an example:

```
type r:= record i: integer;
          case b: colour of
            red, blue: (j: integer);
            green: (k: integer;
                    case c: boolean of
                      true: (a: real)
        end;
```

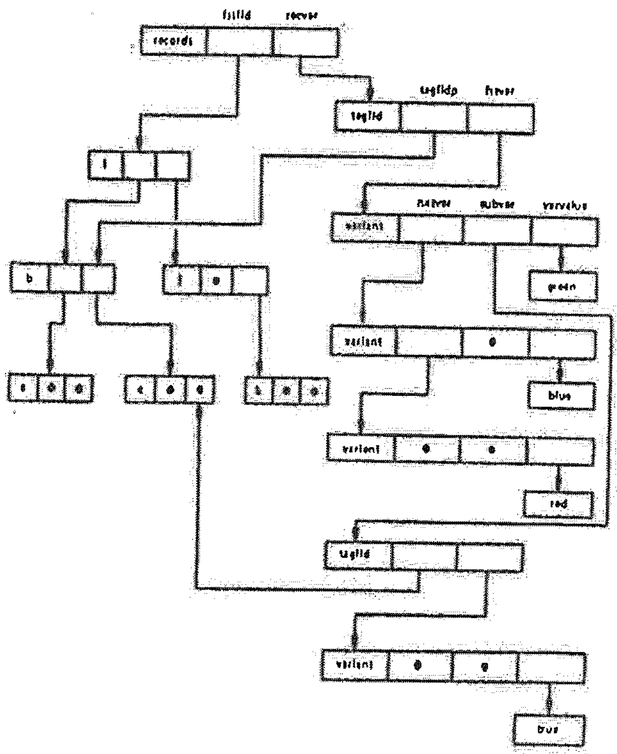
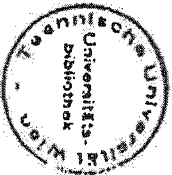


TABLE OF CONTENTS


Compiler Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3

Assembler/Interpreter Listing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .67


The sources of the Pascal program are available in machine-readable form on magnetic tape on application to the publishers.





British Library Cataloguing in Publication Data
Pemberton, Steven
Pascal implementation. – (Ellis Horwood series in computer science)
1. PASCAL (Computer program language)
I. Title II. Daniels, Martin
001.64'24 QA76.73P2

Library of Congress Card No. 81-20184 AACR2

ISBN 0-85312-358-6 Book (Ellis Horwood Ltd., Publishers)
ISBN 0-85312-437-X Compiler (Ellis Horwood Ltd., Publishers)
ISBN 0-470-27325-9 (Halsted Press)


# Compiler Listing

```
 1 (*$c+,t-,d-,l-*)
 2  (**********************************************
 3   *                                            *
 4   *                                            *
 5   *        Portable Pascal compiler            *
 6   *        ************************            *
 7   *                                            *
 8   *              Pascal P4                     *
 9   *                                            *
10   *                                            *
11   *     Authors:                               *
12   *             Urs Ammann                     *
13   *             Kesav Nori                     *
14   *             Christian Jacobi               *
15   *                                            *
16   *     Address:                               *
17   *                                            *
18   *         Institut Fuer Informatik           *
19   *         Eidg. Technische Hochschule        *
20   *         CH-8096 Zuerich                    *
21   *                                            *
22   *                                            *
23   *                                            *
24   *                                            *
25   *                                            *
26   **********************************************)
27
28
29 program pascalcompiler(input,output,prr);
30
31
32
33 const displimit = 20; maxlevel = 10;
34    intsize     =      1;
35    intal       =      1;
36    realsize    =      1;
37    realal      =      1;
38    charsize    =      1;
39    charal      =      1;
40    charmax     =      1;
41    boolsize    =      1;
42    boolal      =      1;
43    ptrsize     =      1;
44    adral       =      1;
45    setsize     =      1;
46    setal       =      1;
47    stackal     =      1;
48    stackelsize =      1;
49    strglgth    =     16;
50    sethigh     =     47;
51    setlow      =      0;
52    ordmaxchar  =     63;
53    ordminchar  =      0;
54    maxint      =  32767;
55    lcaftermarkstack = 5;
```

```
 57    (* stackelsize = minimum size for 1 stackelement
 58                   = k*stackal
 59       stackal     = scm(all other al-constants)
 60       charmax     = scm(charsize,charal)
 61                     scm = smallest common multiple
 62       lcaftermarkstack >= 4*ptrsize+max(x-size)
 63                        = k1*stackelsize          *)
 64    maxstack    =       1;
 65    parmal      = stackal;
 66    parmsize    = stackelsize;
 67    recal       = stackal;
 68    filebuffer  =       4;
 69    maxaddr     =  maxint;
 70
 71
 72
 73 type
 74                                                        (*describing:*)
 75                                                        (*************)
 76
 77                                                        (*basic symbols*)
 78                                                        (***************)
 79
 80    symbol = (ident,intconst,realconst,stringconst,notsy,mulop,addop,relop,
 81              lparent,rparent,lbrack,rbrack,comma,semicolon,period,arrow,
 82              colon,becomes,labelsy,constsy,typesy,varsy,funcsy,progsy,
 83              procsy,setsy,packedsy,arraysy,recordsy,filesy,forwardsy,
 84              beginsy,ifsy,casesy,repeatsy,whilesy,forsy,withsy,
 85              gotosy,endsy,elsesy,untilsy,ofsy,dosy,tosy,downtosy,
 86              thensy,othersy);
 87    operator = (mul,rdiv,andop,idiv,imod,plus,minus,orop,ltop,leop,geop,gtop,
 88                neop,eqop,inop,noop);
 89    setofsys = set of symbol;
 90    chtp = (letter,number,special,illegal,
 91            chstrquo,chcolon,chperiod,chlt,chgt,chlparen,chspace),
 92
 93                                                        (*constants*)
 94                                                        (***********)
 95
 96    cstclass = (reel,pset,strg);
 97    csp = ^ constant;
 98    constant = record case cclass: cstclass of
 99                         reel: (rval: packed array [1..strglgth] of char);
100                         pset: (pval: set of setlow..sethigh);
101                         strg: (slgth: 0..strglgth;
102                                sval: packed array [1..strglgth] of char)
103                      end;
104
105    valu = record case intval: boolean of  (*intval never set nor tested*)
106                 true:  (ival: integer);
107                 false: (valp: csp)
108               end;
109
110                                                        (*data structures*)
111                                                        (*****************)
112    levrange = 0..maxlevel; addrrange = 0..maxaddr;
113    structform = (scalar,subrange,pointer,power,arrays,records,files,
114                  tagfld,variant);
115    declkind = (standard,declared);
116    stp = ^ structure; ctp = ^ identifier;
117
118    structure = packed record
119                  marked: boolean;   (*for test phase only*)
120                  size: addrrange;
```



```
121                  case form: structform of
122                    scalar:   (case scalkind: declkind of
123                                 declared: (fconst: ctp));
124                    subrange: (rangetype: stp; min,max: valu),
125                    pointer:  (eltype: stp),
126                    power:    (elset: stp),
127                    arrays:   (aeltype,inxtype: stp);
128                    records:  (fstfld: ctp; recvar: stp);
129                    files:    (filtype: stp);
130                    tagfld:   (tagfieldp: ctp; fstvar: stp);
131                    variant:  (nxtvar,subvar: stp; varval: valu)
132                  end;
133
134                                                        (*names*)
135                                                        (*******)
136
137    idclass = (types,konst,vars,field,proc,func);
138    setofids = set of idclass;
139    idkind = (actual,formal);
140    alpha = packed array [1..8] of char;
141
142    identifier = packed record
143                   name: alpha; llink, rlink: ctp;
144                   idtype: stp; next: ctp;
145                   case klass: idclass of
146                     konst: (values: valu);
147                     vars:  (vkind: idkind; vlev: levrange; vaddr: addrrange),
148                     field: (fldaddr: addrrange);
149                     proc,
150                     func:  (case pfdeckind: declkind of
151                               standard: (key: 1..15);
152                               declared: (pflev: levrange; pfname: integer;
153                                          case pfkind: idkind of
154                                           actual: (forwdecl, extern:
155                                                    boolean)))
156                 end;
157
158
159    disprange = 0..displimit;
160    where = (blck,crec,vrec,rec);
161
162                                                        (*expressions*)
163                                                        (*************)
164    attrkind = (cst,varbl,expr);
165    vaccess = (drct,indrct,inxd);
166
167    attr = record typtr: stp;
168             case kind: attrkind of
169               cst:   (cval: valu);
170               varbl: (case access: vaccess of
171                         drct: (vlevel: levrange; dplmt: addrrange);
172                         indrct: (idplmt: addrrange))
173             end;
174
175    testp = ^ testpointer;
176    testpointer = packed record
177                    elt1,elt2 : stp;
178                    lasttestp : testp
179                  end;
180
181                                                        (*labels*)
182                                                        (********)
183    lbp = ^ labl;
184    labl = record nextlab: lbp; defined: boolean;
```

```
185                 labval, labname: integer
106             end;
187
188       extfilep = ^filerec;
189       filerec = record filename:alpha; nextfile:extfilep end;
190
191  (*-----------------------------------------------------------------------*)
192
193
194  var
195                                     (*returned by source program scanner
196                                      insymbol:
197                                      **********)
198
199       sy: symbol;                   (*last symbol*)
200       op: operator;                 (*classification of last symbol*)
201       val: valu;                    (*value of last constant*)
202       lgth: integer;                (*length of last string constant*)
203       id: alpha;                    (*last identifier (possibly truncated)*)
204       kk: 1..8;                     (*nr of chars in last identifier*)
205       ch: char;                     (*last character*)
206       eol: boolean;                 (*end of line flag*)
207
208
209                                     (*counters:*)
210                                     (***********)
211
212       chcnt: integer;               (*character counter*)
213       lc,ic: addrrange;             (*data location and instruction counter*)
214       linecount: integer;
215
216
217                                     (*switches:*)
218                                     (***********)
219
220       dp,                           (*declaration part*)
221       prterr,                       (*to allow forward references in pointer type
222                                       declaration by suppressing error message*)
223       list,prcode,prtables: boolean; (*output options for
224                                         -- source program listing
225                                         -- printing symbolic code
226                                         -- displaying ident and struct tables
227                                         --> procedure option*)
228       debug: boolean;
229
230
231                                     (*pointers:*)
232                                     (***********)
233       parmptr,
234       intptr,realptr,charptr,
235       boolptr,nilptr,textptr: stp;  (*pointers to entries of standard ids*)
236       utypptr,ucstptr,uvarptr,
237       ufldptr,uprcptr,ufctptr,      (*pointers to entries for undeclared ids*)
238       fwptr: ctp;                   (*head of chain of forw decl type ids*)
239       fextfilep: extfilep;          (*head of chain of external files*)
240       globtestp: testp;             (*last testpointer*)
241
242
243                                     (*bookkeeping of declaration levels:*)
244                                     (************************************)
245
246       level: levrange;              (*current static level*)
247       disx,                         (*level of last id searched by searchid*)
248       top: disprange;               (*top of display*)
249
250       display:                      (*where:   means:*)
251         array [disprange] of
252           packed record             (*=blck:   id is variable id*)
253             fname: ctp; flabel: lbp; (*=crec:  id is field id in record with*)
254             case occur: where of    (*         constant address*)
255               crec: (clev: levrange; (*=vrec:  id is field id in record with*)
256                      cdspl: addrrange);(*       variable address*)
257               vrec: (vdspl: addrrange)
258           end;                      (* --> procedure withstatement*)
259
260
261
262                                     (*error messages:*)
263                                     (*****************)
264       errinx: 0..10;                (*nr of errors in current source line*)
265       errlist:
266         array [1..10] of
267           packed record pos: integer;
260                         nmr: 1..400
269                  end;
270
271
272
273                                     (*expression compilation:*)
274                                     (*************************)
275
276       gattr: attr;                  (*describes the expr currently compiled*)
277
278
279                                     (*structured constants:*)
280                                     (***********************)
281
282       constbegsys,simptypebegsys,typebegsys,blockbegsys,selectsys,facbegsys,
283       statbegsys,typedels: setofsys;
284       chartp : array[char] of chtp;
285       rw:  array [1..35(*nr. of res. words*)] of alpha;
286       frw: array [1..9] of 1..36(*nr. of res. words + 1*);
287       rsy: array [1..35(*nr. of res. words*)] of symbol;
288       ssy: array [char] of symbol;
289       rop: array [1..35(*nr. of res. words*)] of operator;
290       sop: array [char] of operator;
291       na:  array [1..35] of alpha;
292       mn:  array [0..60] of packed array [1..4] of char;
293       sna: array [1..23] of packed array [1..4] of char;
294       cdx: array [0..60] of -4..+4;
295       pdx: array [1..23] of -7..+7;
296       ordint: array [char] of integer;
297
298       intlabel,mxint10,digmax: integer;
299
300  (*-----------------------------------------------------------------------*)
301
302
303     procedure endofline;
304       var lastpos,freepos,currpos,currnmr,f,k: integer;
305     begin
306       if errinx > 0 then   (*output error messages*)
307         begin write(output,' ****  ':15);
308           lastpos := 0; freepos := 1;
309           for k := 1 to errinx do
310             begin
311               with errlist[k] do
312                 begin currpos := pos; currnmr := nmr end;
```

```
313              if currpos = lastpos then write(output,',')
314              else
315                begin
316                  while freepos < currpos do
317                    begin write(output,' '); freepos := freepos + 1 end;
318                  write(output,'^');
319                  lastpos := currpos
320                end;
321              if currnmr < 10 then f := 1
322              else if currnmr < 100 then f := 2
323                else f := 3;
324              write(output,currnmr:f);
325              freepos := freepos + f + 1
326            end;
327          writeln(output); errinx := 0
328        end;
329      linecount := linecount + 1;
330      if list and (not eof(input)) then
331        begin write(output,linecount:6,'  ':2);
332          if dp then write(output,lc:7) else write(output,ic:7);
333          write(output,' ')
334        end;
335      chcnt := 0
336    end  (*endofline*) ;
337
338    procedure error(ferrnr: integer);
339    begin
340      if errinx >= 9 then
341        begin errlist[10].nmr := 255; errinx := 10 end
342      else
343        begin errinx := errinx + 1;
344          errlist[errinx].nmr := ferrnr
345        end;
346      errlist[errinx].pos := chcnt
347    end (*error*) ;
348
349    procedure insymbol;
350      (*read next basic symbol of source program and return its
351      description in the global variables sy, op, id, val and lgth*)
352      label 1,2,3;
353      var i,k: integer;
354          digit: packed array [1..strglgth] of char;
355          string: packed array [1..strglgth] of char;
356          lvp: csp; test: boolean;
357
358      procedure nextch;
359      begin if eol then
360        begin if list then writeln(output); endofline
361        end;
362        if not eof(input) then
363         begin eol := eoln(input); read(input,ch);
364          if list then write(output,ch);
365          chcnt := chcnt + 1
366         end
367        else
368          begin writeln(output,'   *** eof ','encountered');
369            test := false
370          end
371      end;
372
373      procedure options;
374      begin
375        repeat nextch;
376          if ch <> '*' then
```



```
377          begin
378            if ch = 't' then
379              begin nextch; prtables := ch = '+' end
380            else
381              if ch = 'l' then
382                begin nextch; list := ch = '+';
383                  if not list then writeln(output)
384                end
385              else
386              if ch = 'd' then
387                begin nextch; debug := ch = '+' end
388              else
389                if ch = 'c' then
390                  begin nextch; prcode := ch = '+' end;
391            nextch
392          end
393        until ch <> ','
394      end (*options*) ;
395
396    begin (*insymbol*)
397    1:
398      repeat while (ch = ' ') and not eol do nextch;
399        test := eol;
400        if test then nextch
401      until not test;
402      if chartp[ch] = illegal then
403        begin sy := othersy; op := noop;
404          error(399); nextch
405        end
406      else
407      case chartp[ch] of
408        letter:
409          begin k := 0;
410            repeat
411              if k < 8 then
412               begin k := k + 1; id[k] := ch end ;
413              nextch
414            until chartp[ch] in [special,illegal,chstrquo,chcolon,
415                                 chperiod,chlt,chgt,chlparen,chspace];
416            if k >= kk then kk := k
417            else
418              repeat id[kk] := ' '; kk := kk - 1
419              until kk = k;
420            for i := frw[k] to frw[k+1] - 1 do
421              if rw[i] = id then
422                begin sy := rsy[i]; op := rop[i]; goto 2 end;
423              sy := ident; op := noop;
424  2:      end;
425        number:
426          begin op := noop; i := 0;
427            repeat i := i+1; if i<= digmax then digit[i] := ch; nextch
428            until chartp[ch] <> number;
429            if (ch = '.') or (ch = 'e') then
430              begin
431                    k := i;
432                    if ch = '.' then
433                      begin k := k+1; if k <= digmax then digit[k] := ch;
434                        nextch; if ch = '.' then begin ch := ':'; goto 3 end;
435                        if chartp[ch] <> number then error(201)
436                        else
437                          repeat k := k + 1;
438                            if k <= digmax then digit[k] := ch; nextch
439                          until chartp[ch] <>  number
440                    end;
```

```
441                 if ch = 'e' then
442                   begin k := k+1; if k <= digmax then digit[k] := ch;
443                     nextch;
444                     if (ch = '+') or (ch ='-') then
445                       begin k := k+1; if k <= digmax then digit[k] := ch;
446                         nextch
447                       end;
448                     if chartp[ch] <> number then error(201)
449                     else
450                       repeat k := k+1;
451                         if k <= digmax then digit[k] := ch; nextch
452                       until chartp[ch] <> number
453                   end;
454                 new(lvp,reel); sy:= realconst; lvp^.cclass := reel;
455                 with lvp^ do
456                   begin for i := 1 to strglgth do rval[i] := ' ';
457                     if k <= digmax then
458                       for i := 2 to k + 1 do rval[i] := digit[i-1]
459                     else begin error(203); rval[2] := '0';
460                            rval[3] := '.'; rval[4] := '0'
461                          end
462                   end;
463                 val.valp := lvp
464             end
465           else
466   3:        begin
467               if i > digmax then begin error(203); val.ival := 0 end
468               else
469                 with val do
470                   begin ival := 0;
471                     for k := 1 to i do
472                       begin
473                         if ival <= mxint10 then
474                           ival := ival*10+ordint[digit[k]]
475                         else begin error(203); ival := 0 end
476                       end;
477                     sy := intconst
478                   end
479             end
480         end;
481       chstrquo:
482         begin lgth := 0; sy := stringconst;  op := noop;
483           repeat
484             repeat nextch; lgth := lgth + 1;
485                    if lgth <= strglgth then string[lgth] := ch
486             until (eol) or (ch = '''');
487             if eol then error(202) else nextch
488           until ch <> '''';
489           lgth := lgth - 1;   (*now lgth = nr of chars in string*)
490           if lgth = 0 then error(205) else
491           if lgth = 1 then val.ival := ord(string[1])
492           else
493             begin new(lvp,strg); lvp^.cclass:=strg;
494               if lgth > strglgth then
495                 begin error(399); lgth := strglgth end;
496               with lvp^ do
497                 begin slgth := lgth;
498                   for i := 1 to lgth do sval[i] := string[i]
499                 end;
500               val.valp := lvp
501             end
502         end;
503       chcolon:
504         begin op := noop; nextch;
```



```
505           if ch = '=' then
506             begin sy := becomes; nextch end
507           else sy := colon
508         end;
509       chperiod:
510         begin op := noop; nextch;
511           if ch = '.' then
512             begin sy := colon; nextch end
513           else sy := period
514         end;
515       chlt:
516         begin nextch; sy := relop;
517           if ch = '=' then
518             begin op := leop; nextch end
519           else
520             if ch = '>' then
521               begin op := neop; nextch end
522             else op := ltop
523         end;
524       chgt:
525         begin nextch; sy := relop;
526           if ch = '=' then
527             begin op := geop; nextch end
528           else op := gtop
529         end;
530       chlparen:
531        begin nextch;
532          if ch = '*' then
533            begin nextch;
534              if ch = '$' then options;
535              repeat
536                while (ch <> '*') and not eof(input) do nextch;
537                nextch
538              until (ch = ')') or eof(input);
539              nextch; goto 1
540            end;
541          sy := lparent; op := noop
542        end;
543       special:
544         begin sy := ssy[ch]; op := sop[ch];
545           nextch
546         end;
547       chspace: sy := othersy
548     end (*case*)
549   end (*insymbol*) ;
550
551   procedure enterid(fcp: ctp);
552     (*enter id pointed at by fcp into the name-table,
553      which on each declaration level is organised as
554      an unbalanced binary tree*)
555     var nam: alpha; lcp, lcp1: ctp; lleft: boolean;
556   begin nam := fcp^.name;
557     lcp := display[top].fname;
558     if lcp = nil then
559       display[top].fname := fcp
560     else
561       begin
562         repeat lcp1 := lcp;
563           if lcp^.name = nam then   (*name conflict, follow right link*)
564             begin error(101); lcp := lcp^.rlink; lleft := false end
565           else
566             if lcp^.name < nam then
567               begin lcp := lcp^.rlink; lleft := false end
568             else begin lcp := lcp^.llink; lleft := true end
```

```
569        until lcp = nil;
570        if lleft then lcp1^.llink := fcp else lcp1^.rlink := fcp
571      end;
572    fcp^.llink := nil; fcp^.rlink := nil
573  end (*enterid*) ;
574
575  procedure searchsection(fcp: ctp; var fcp1: ctp);
576    (*to find record fields and forward declared procedure id's
577     --> procedure proceduredeclaration
578     --> procedure selector*)
579    label 1;
580  begin
581    while fcp <> nil do
582      if fcp^.name = id then goto 1
583      else if fcp^.name < id then fcp := fcp^.rlink
584        else fcp := fcp^.llink;
585 1:  fcp1 := fcp
586  end (*searchsection*) ;
587
588  procedure searchid(fidcls: setofids; var fcp: ctp);
589    label 1;
590    var lcp: ctp;
591  begin
592    for disx := top downto 0 do
593      begin lcp := display[disx].fname;
594        while lcp <> nil do
595          if lcp^.name = id then
596            if lcp^.klass in fidcls then goto 1
597            else
598              begin if prterr then error(103);
599                lcp := lcp^.rlink
600              end
601          else
602            if lcp^.name < id then
603              lcp := lcp^.rlink
604            else lcp := lcp^.llink
605      end;
606    (*search not successful; suppress error message in case
607     of forward referenced type id in pointer type definition
608     --> procedure simpletype*)
609    if prterr then
610      begin error(104);
611        (*to avoid returning nil, reference an entry
612         for an undeclared id of appropriate class
613         --> procedure enterundecl*)
614        if types in fidcls then lcp := utypptr
615        else
616          if vars in fidcls then lcp := uvarptr
617          else
618            if field in fidcls then lcp := ufldptr
619            else
620              if konst in fidcls then lcp := ucstptr
621              else
622                if proc in fidcls then lcp := uprcptr
623                else lcp := ufctptr;
624      end;
625 1:  fcp := lcp
626  end (*searchid*) ;
627
628  procedure getbounds(fsp: stp; var fmin,fmax: integer);
629    (*get internal bounds of subrange or scalar type*)
630    (*assume fsp<>intptr and fsp<>realptr*)
631  begin
632    fmin := 0; fmax := 0;
```



```
633    if fsp <> nil then
634    with fsp^ do
635      if form = subrange then
636        begin fmin := min.ival; fmax := max.ival end
637      else
638          if fsp = charptr then
639            begin fmin := ordminchar; fmax := ordmaxchar
640            end
641          else
642            if fconst <> nil then
643              fmax := fconst^.values.ival
644  end (*getbounds*) ;
645
646  function alignquot(fsp: stp): integer;
647  begin
648    alignquot := 1;
649    if fsp <> nil then
650      with fsp^ do
651        case form of
652          scalar:   if fsp=intptr then alignquot := intal
653                    else if fsp=boolptr then alignquot := boolal
654                    else if scalkind=declared then alignquot := intal
655                    else if fsp=charptr then alignquot := charal
656                    else if fsp=realptr then alignquot := realal
657                    else (*parmptr*) alignquot := parmal;
658          subrange: alignquot := alignquot(rangetype);
659          pointer:  alignquot := adral;
660          power:    alignquot := setal;
661          files:    alignquot := fileal;
662          arrays:   alignquot := alignquot(aeltype);
663          records:  alignquot := recal;
664          variant,tagfld: error(501)
665        end
666  end (*alignquot*);
667
668  procedure align(fsp: stp; var flc: integer);
669    var k,l: integer;
670  begin
671    k := alignquot(fsp);
672    l := flc-1;
673    flc := l + k  -  (k+l) mod k
674  end (*align*);
675
676  procedure printtables(fb: boolean);
677    (*print data structure and name table*)
678    var i, lim: disprange;
679
680    procedure marker;
681      (*mark data structure entries to avoid multiple printout*)
682      var i: integer;
683
684      procedure markctp(fp: ctp); forward;
685
686      procedure markstp(fp: stp);
687        (*mark data structures, prevent cycles*)
688      begin
689        if fp <> nil then
690          with fp^ do
691            begin marked := true;
692              case form of
693                scalar:   ;
694                subrange: markstp(rangetype);
695                pointer:  (*don't mark eltype: cycle possible; will be marked
696                           anyway, if fp = true*) ;
```

```
825                        else write(output,'formal':10)
826                      end
827                    end
828               end (*case*);
829               writeln(output);
830               followctp(llink); followctp(rlink);
831               followstp(idtype)
832             end (*with*)
833         end (*followctp*);
834
835     begin (*printtables*)
836       writeln(output); writeln(output); writeln(output);
837       if fb then lim := 0
838       else begin lim := top; write(output,' local') end;
839       writeln(output,' tables '); writeln(output);
840       marker;
841       for i := top downto lim do
842         followctp(display[i].fname);
843       writeln(output);
844       if not eol then write(output,' ':chcnt+16)
845     end (*printtables*);
846
847     procedure genlabel(var nxtlab: integer);
848     begin intlabel := intlabel + 1;
849       nxtlab := intlabel
850     end (*genlabel*);
851
852     procedure block(fsys: setofsys; fsy: symbol; fprocp: ctp);
853       var lsy: symbol; test: boolean;
854
855       procedure skip(fsys: setofsys);
856         (*skip input string until relevant symbol found*)
857       begin
858         if not eof(input) then
859           begin while not(sy in fsys) and (not eof(input)) do insymbol;
860             if not (sy in fsys) then insymbol
861           end
862       end (*skip*) ;
863
864       procedure constant(fsys: setofsys; var fsp: stp; var fvalu: valu);
865         var lsp: stp; lcp: ctp; sign: (none,pos,neg);
866             lvp: csp; i: 2..strglgth;
867       begin lsp := nil; fvalu.ival := 0;
868         if not(sy in constbegsys) then
869           begin error(50); skip(fsys+constbegsys) end;
870         if sy in constbegsys then
871           begin
872             if sy = stringconstsy then
873               begin
874                 if lgth = 1 then lsp := charptr
875                 else
876                   begin
877                     new(lsp,arrays);
878                     with lsp^ do
879                       begin aeltype := charptr; inxtype := nil;
880                         size := lgth*charsize; form := arrays
881                       end
882                   end;
883                 fvalu := val; insymbol
884               end
885             else
886               begin
887                 sign := none;
888                 if (sy = addop) and (op in [plus,minus]) then
```



```
889                   begin if op = plus then sign := pos else sign := neg;
890                     insymbol
891                   end;
892                 if sy = ident then
893                   begin searchid([konst],lcp);
894                     with lcp^ do
895                       begin lsp := idtype; fvalu := values end;
896                     if sign <> none then
897                       if lsp = intptr then
898                         begin if sign = neg then fvalu.ival := -fvalu.ival end
899                       else
900                         if lsp = realptr then
901                           begin
902                             if sign = neg then
903                               begin new(lvp,reel);
904                                 if fvalu.valp^.rval[1] = '-' then
905                                   lvp^.rval[1] := '+'
906                                 else lvp^.rval[1] := '-';
907                                 for i := 2 to strglgth do
908                                   lvp^.rval[i] := fvalu.valp^.rval[i];
909                                 fvalu.valp := lvp;
910                               end
911                           end
912                         else error(105);
913                     insymbol;
914                   end
915                 else
916                   if sy = intconst then
917                     begin if sign = neg then val.ival := -val.ival;
918                       lsp := intptr; fvalu := val; insymbol
919                     end
920                   else
921                     if sy = realconst then
922                       begin if sign = neg then val.valp^.rval[1] := '-';
923                         lsp := realptr; fvalu := val; insymbol
924                       end
925                     else
926                       begin error(106); skip(fsys) end
927               end;
928             if not (sy in fsys) then
929               begin error(6); skip(fsys) end
930             end;
931         fsp := lsp
932       end (*constant*) ;
933
934       function equalbounds(fsp1,fsp2: stp): boolean;
935         var lmin1,lmin2,lmax1,lmax2: integer;
936       begin
937         if (fsp1=nil) or (fsp2=nil) then equalbounds := true
938         else
939           begin
940             getbounds(fsp1,lmin1,lmax1);
941             getbounds(fsp2,lmin2,lmax2);
942             equalbounds := (lmin1=lmin2) and (lmax1=lmax2)
943           end
944       end (*equalbounds*) ;
945
946       function comptypes(fsp1,fsp2: stp) : boolean;
947         (*decide whether structures pointed at by fsp1 and fsp2 are compatible*)
948         var nxt1,nxt2: ctp; comp: boolean;
949           ltestp1,ltestp2 : testp;
950       begin
951         if fsp1 = fsp2 then comptypes := true
952         else
```

```
697               power:    markstp(elset) ;
698               arrays:   begin markstp(aeltype); markstp(inxtype) end;
699               records:  begin markstp(fstfld); markstp(recvar) end;
700               files:    markstp(filtype);
701               tagfld:   markstp(fstvar);
702               variant:  begin markstp(nxtvar); markstp(subvar) end
703               end (*case*)
704             end (*with*)
705     end (*markstp*);
706
707     procedure markctp;
708     begin
709       if fp <> nil then
710         with fp^ do
711           begin markctp(llink); markctp(rlink);
712             markstp(idtype)
713           end
714     end (*markctp*);
715
716   begin (*marker*)
717     for i := top downto lim do
718       markctp(display[i].fname)
719   end (*marker*);
720
721   procedure followctp(fp: ctp); forward;
722
723   procedure followstp(fp: stp);
724   begin
725     if fp <> nil then
726       with fp^ do
727         if marked then
728           begin marked := false; write(output,' ':4,ord(fp):6,size:10);
729             case form of
730             scalar:   begin write(output,'scalar':10);
731                         if scalkind = standard then
732                           write(output,'standard':10)
733                         else write(output,'declared':10,' ':4,ord(fconst):6);
734                         writeln(output)
735                       end;
736             subrange: begin
737                         write(output,'subrange':10,' ':4,ord(rangetype):6);
738                         if rangetype <> realptr then
739                           write(output,min.ival,max.ival)
740                         else
741                           if (min.valp <> nil) and (max.valp <> nil) then
742                             write(output,' ',min.valp^.rval:9,
743                                   ' ',max.valp^.rval:9);
744                         writeln(output); followstp(rangetype);
745                       end;
746             pointer:  writeln(output,'pointer':10,' ':4,ord(eltype):6);
747             power:    begin writeln(output,'set':10,' ':4,ord(elset):6);
748                         followstp(elset)
749                       end;
750             arrays:   begin
751                         writeln(output,'array':10,' ':4,ord(aeltype):6,' ':4,
752                           ord(inxtype):6);
753                         followstp(aeltype); followstp(inxtype)
754                       end;
755             records:  begin
756                         writeln(output,'record':10,' ':4,ord(fstfld):6,' ':4,
757                           ord(recvar):6); followctp(fstfld);
758                         followstp(recvar)
759                       end;
760             files:    begin write(output,'file':10,' ':4,ord(filtype):6);
```



```
761                         followstp(filtype)
762                       end;
763             tagfld:   begin writeln(output,'tagfld':10,' ':4,ord(tagfieldp):6,
764                         ' ':4,ord(fstvar):6);
765                         followstp(fstvar)
766                       end;
767             variant:  begin writeln(output,'variant':10,' ':4,ord(nxtvar):6,
768                         ' ':4,ord(subvar):6,varval.ival);
769                         followstp(nxtvar); followstp(subvar)
770                       end
771             end (*case*)
772           end (*if marked*)
773   end (*followstp*);
774
775   procedure followctp;
776     var i: integer;
777   begin
778     if fp <> nil then
779       with fp^ do
780         begin write(output,' ':4,ord(fp):6,' ',name:9,' ':4,ord(llink):6,
781           ' ':4,ord(rlink):6,' ':4,ord(idtype):6);
782           case klass of
783             types: write(output,'type':10);
784             konst: begin write(output,'constant':10,' ':4,ord(next):6);
785                      if idtype <> nil then
786                        if idtype = realptr then
787                          begin
788                            if values.valp <> nil then
789                              write(output,' ',values.valp^.rval:9)
790                          end
791                        else
792                          if idtype^.form = arrays then  (*stringconst*)
793                            begin
794                              if values.valp <> nil then
795                                begin write(output,' ');
796                                  with values.valp^ do
797                                    for i := 1 to slgth do
798                                      write(output,sval[i])
799                                end
800                            end
801                          else write(output,values.ival)
802                    end;
803             vars:  begin write(output,'variable':10);
804                      if vkind = actual then write(output,'actual':10)
805                      else write(output,'formal':10);
806                      write(output,' ':4,ord(next):6,vlev,' ':4,vaddr:6 );
807                    end;
808             field: write(output,'field':10,' ':4,ord(next):6,' ':4,fldaddr:6);
809             proc,
810             func:  begin
811                      if klass = proc then write(output,'procedure':10)
812                      else write(output,'function':10);
813                      if pfdeckind = standard then
814                        write(output,'standard':10, key:10)
815                      else
816                        begin write(output,'declared':10,' ':4,ord(next):6);
817                          write(output,pflev,' ':4,pfname:6);
818                          if pfkind = actual then
819                            begin write(output,'actual':10);
820                              if forwdecl then write(output,'forward':10)
821                              else write(output,'notforward':10);
822                              if extern then write(output,'extern':10)
823                              else write(output,'not extern':10);
824                            end
```

```
 953        if (fsp1 <> nil) and (fsp2 <> nil) then
 954          if fsp1^.form = fsp2^.form then
 955            case fsp1^.form of
 956              scalar:
 957                comptypes := false;
 958                (* identical scalars declared on different levels are
 959                 not recognized to be compatible*)
 960              subrange:
 961                comptypes := comptypes(fsp1^.rangetype,fsp2^.rangetype),
 962              pointer:
 963                  begin
 964                    comp := false; ltestp1 := globtestp;
 965                    ltestp2 := globtestp;
 966                    while ltestp1 <> nil do
 967                      with ltestp1^ do
 968                        begin
 969                          if (elt1 = fsp1^.eltype) and
 970                             (elt2 = fsp2^.eltype) then comp := true;
 971                          ltestp1 := lasttestp
 972                        end;
 973                    if not comp then
 974                      begin new(ltestp1);
 975                        with ltestp1^ do
 976                          begin elt1 := fsp1^.eltype;
 977                            elt2 := fsp2^.eltype;
 978                            lasttestp := globtestp
 979                          end;
 980                        globtestp := ltestp1;
 981                        comp := comptypes(fsp1^.eltype,fsp2^.eltype)
 982                      end;
 983                    comptypes := comp; globtestp := ltestp2
 984                  end;
 985              power:
 986                comptypes := comptypes(fsp1^.elset,fsp2^.elset);
 987              arrays:
 988                begin
 989                  comp := comptypes(fsp1^.aeltype,fsp2^.aeltype)
 990                      and comptypes(fsp1^.inxtype,fsp2^.inxtype);
 991                  comptypes := comp and (fsp1^.size = fsp2^.size) and
 992                      equalbounds(fsp1^.inxtype,fsp2^.inxtype)
 993                end;
 994              records:
 995                begin nxt1 := fsp1^.fstfld; nxt2 := fsp2^.fstfld; comp:=true;
 996                  while (nxt1 <> nil) and (nxt2 <> nil) do
 997                    begin comp:=comp and comptypes(nxt1^.idtype,nxt2^.idtype),
 998                      nxt1 := nxt1^.next; nxt2 := nxt2^.next
 999                    end;
1000                  comptypes := comp and (nxt1 = nil) and (nxt2 = nil)
1001                               and(fsp1^.recvar = nil)and(fsp2^.recvar = nil)
1002                end;
1003                (*identical records are recognized to be compatible
1004                 iff no variants occur*)
1005              files:
1006                comptypes := comptypes(fsp1^.filtype,fsp2^.filtype)
1007            end (*case*)
1008          else (*fsp1^.form <> fsp2^.form*)
1009            if fsp1^.form = subrange then
1010              comptypes := comptypes(fsp1^.rangetype,fsp2)
1011            else
1012              if fsp2^.form = subrange then
1013                comptypes := comptypes(fsp1,fsp2^.rangetype)
1014              else comptypes := false
1015        else comptypes := true
1016      end (*comptypes*) ;
```



```
1017
1018      function string(fsp: stp) : boolean;
1019      begin string := false;
1020        if fsp <> nil then
1021          if fsp^.form = arrays then
1022            if comptypes(fsp^.aeltype,charptr) then string := true
1023      end (*string*) ;
1024
1025      procedure typ(fsys: setofsys; var fsp: stp; var fsize: addrrange);
1026        var lsp,lsp1,lsp2: stp; oldtop: disprange; lcp: ctp;
1027            lsize,displ: addrrange; lmin,lmax: integer;
1028
1029        procedure simpletype(fsys:setofsys; var fsp:stp; var fsize:addrrange);
1030          var lsp,lsp1: stp; lcp,lcp1: ctp; ttop: disprange;
1031              lcnt: integer; lvalu: valu;
1032        begin fsize := 1;
1033          if not (sy in simptypebegsys) then
1034            begin error(1); skip(fsys + simptypebegsys) end;
1035          if sy in simptypebegsys then
1036            begin
1037              if sy = lparent then
1038                begin ttop := top;   (*decl. consts local to innermost block*)
1039                  while display[top].occur <> blck do top := top - 1;
1040                  new(lsp,scalar,declared);
1041                  with lsp^ do
1042                    begin size := intsize; form := scalar;
1043                      scalkind := declared
1044                    end;
1045                  lcp1 := nil; lcnt := 0;
1046                  repeat insymbol;
1047                    if sy = ident then
1048                      begin new(lcp,konst);
1049                        with lcp^ do
1050                          begin name := id; idtype := lsp; next := lcp1,
1051                            values.ival := lcnt; klass := konst
1052                          end;
1053                        enterid(lcp);
1054                        lcnt := lcnt + 1;
1055                        lcp1 := lcp; insymbol
1056                      end
1057                    else error(2);
1058                    if not (sy in fsys + [comma,rparent]) then
1059                      begin error(6); skip(fsys + [comma,rparent]) end
1060                  until sy <> comma;
1061                  lsp^.fconst := lcp1; top := ttop;
1062                  if sy = rparent then insymbol else error(4)
1063                end
1064              else
1065                begin
1066                  if sy = ident then
1067                    begin searchid([types,konst],lcp);
1068                      insymbol;
1069                      if lcp^.klass = konst then
1070                        begin new(lsp,subrange);
1071                          with lsp^, lcp^ do
1072                            begin rangetype := idtype; form := subrange;
1073                              if string(rangetype) then
1074                                begin error(148); rangetype := nil end;
1075                              min := values; size := intsize
1076                            end;
1077                          if sy = colon then insymbol else error(5);
1078                          constant(fsys,lsp1,lvalu);
1079                          lsp^.max := lvalu;
1080                          if lsp^.rangetype <> lsp1 then error(107)
```

```
1721              begin name := id; idtype := nil;
1722                extern := false; pflev := level; genlabel(lbname);
1723                pfdeckind := declared; pfkind := actual; pfname := lbname;
1724                if fsy = procsy then klass := proc
1725                else klass := func
1726              end;
1727            enterid(lcp)
1728          end
1729        else
1730          begin lcp1 := lcp^.next;
1731            while lcp1 <> nil do
1732              begin
1733                with lcp1^ do
1734                  if klass = vars then
1735                    if idtype <> nil then
1736                      begin lcm := vaddr + idtype^.size;
1737                        if lcm > lc then lc := lcm
1738                      end;
1739                lcp1 := lcp1^.next
1740              end
1741          end;
1742        insymbol
1743      end
1744    else
1745      begin error(2); lcp := ufctptr end;
1746    oldlev := level; oldtop := top;
1747    if level < maxlevel then level := level + 1 else error(251);
1748    if top < displimit then
1749      begin top := top + 1;
1750        with display[top] do
1751          begin
1752            if forw then fname := lcp^.next
1753            else fname := nil;
1754            flabel := nil;
1755            occur := blck
1756          end
1757      end
1758    else error(250);
1759    if fsy = procsy then
1760      begin parameterlist([semicolon],lcp1);
1761        if not forw then lcp^.next := lcp1
1762      end
1763    else
1764      begin parameterlist([semicolon,colon],lcp1);
1765        if not forw then lcp^.next := lcp1;
1766        if sy = colon then
1767          begin insymbol;
1768            if sy = ident then
1769              begin if forw then error(122);
1770                searchid([types],lcp1);
1771                lsp := lcp1^.idtype;
1772                lcp^.idtype := lsp;
1773                if lsp <> nil then
1774                  if not (lsp^.form in [scalar,subrange,pointer]) then
1775                    begin error(120); lcp^.idtype := nil end;
1776                insymbol
1777              end
1778            else begin error(2); skip(fsys + [semicolon]) end
1779          end
1780        else
1781          if not forw then error(123)
1782      end;
1783    if sy = semicolon then insymbol else error(14);
1784    if sy = forwardsy then
```

## Compiler Listing



```
1785      begin
1786        if forw then error(161)
1787        else lcp^.forwdecl := true;
1788        insymbol;
1789        if sy = semicolon then insymbol else error(14);
1790        if not (sy in fsys) then
1791          begin error(6); skip(fsys) end
1792      end
1793    else
1794      begin lcp^.forwdecl := false; mark(markp);
1795        repeat block(fsys,semicolon,lcp);
1796          if sy = semicolon then
1797            begin if prtables then printtables(false); insymbol;
1798              if not (sy in [beginsy,procsy,funcsy]) then
1799                begin error(6); skip(fsys) end
1800            end
1801          else error(14)
1802        until (sy in [beginsy,procsy,funcsy]) or eof(input);
1803        release(markp); (* return local entries on runtime heap *)
1804      end;
1805    level := oldlev; top := oldtop; lc := llc;
1806  end (*procdeclaration*) ;
1807
1808  procedure body(fsys: setofsys);
1809    const cstoccmax=65; cixmax=1000;
1810    type oprange = 0..63;
1811    var
1812        llcp:ctp; saveid:alpha;
1813        cstptr: array [1..cstoccmax] of csp;
1814        cstptrix: 0..cstoccmax;
1815        (*allows referencing of noninteger constants by an index
1816         (instead of a pointer), which can be stored in the p2-field
1817         of the instruction record until writeout.
1818         --> procedure load, procedure writeout*)
1819        i, entname, segsize: integer;
1820        stacktop, topnew, topmax: integer;
1821        lcmax,llc1: addrrange; lcp: ctp;
1822        llp: lbp;
1823
1824
1825    procedure mes(i: integer);
1826    begin topnew := topnew + cdx[i]*maxstack;
1827      if topnew > topmax then topmax := topnew
1828    end;
1829
1830    procedure putic;
1831    begin if ic mod 10 = 0 then writeln(prr,'i',ic:5) end;
1832
1833    procedure gen0(fop: oprange);
1834    begin
1835      if prcode then begin putic; writeln(prr,mn[fop]:4) end;
1836      ic := ic + 1; mes(fop)
1837    end (*gen0*) ;
1838
1839    procedure gen1(fop: oprange; fp2: integer);
1840      var k: integer;
1841    begin
1842      if prcode then
1843        begin putic; write(prr,mn[fop]:4);
1844          if fop = 30 then
1845            begin writeln(prr,sna[fp2]:12);
1846              topnew := topnew + pdx[fp2]*maxstack;
1847              if topnew > topmax then topmax := topnew
1848            end
```

```
1849            else
1850              begin
1851                if fop = 38 then
1852                  begin write(prr,'''');
1853                    with cstptr[fp2]^ do
1854                    begin
1855                      for k := 1 to slgth do write(prr,sval[k]:1);
1856                      for k := slgth+1 to strglgth do write(prr,' ');
1857                    end;
1858                    writeln(prr,'''')
1859                  end
1860                else if fop = 42 then writeln(prr,chr(fp2))
1861                    else writeln(prr,fp2:12);
1862                mes(fop)
1863              end
1864          end;
1865      ic := ic + 1
1866    end (*gen1*) ;
1867
1868    procedure gen2(fop: oprange; fp1,fp2: integer);
1869      var k : integer;
1870    begin
1871      if prcode then
1872        begin putic; write(prr,mn[fop]:4);
1873          case fop of
1874            45,50,54,56:
1875              writeln(prr,' ',fp1:3,fp2:8);
1876            47,48,49,52,53,55:
1877              begin write(prr,chr(fp1));
1878                if chr(fp1) = 'm' then write(prr,fp2:11);
1879                writeln(prr)
1880              end;
1881            51:
1882              case fp1 of
1883                1: writeln(prr,'i ',fp2);
1884                2: begin write(prr,'r ');
1885                     with cstptr[fp2]^ do
1886                       for k := 1 to strglgth do write(prr,rval[k]);
1887                     writeln(prr)
1888                   end;
1889                3: writeln(prr,'b ',fp2);
1890                4: writeln(prr,'n');
1891                6: writeln(prr,'c ''':3,chr(fp2),'''');
1892                5: begin write(prr,'(');
1893                     with cstptr[fp2]^ do
1894                       for k := setlow to sethigh do
1895                         if k in pval then write(prr,k:3);
1896                     writeln(prr,')')
1897                   end
1898              end
1899          end;
1900        end;
1901      ic := ic + 1; mes(fop)
1902    end (*gen2*) ;
1903
1904    procedure gentypindicator(fsp: stp);
1905    begin
1906      if fsp<>nil then
1907        with fsp^ do
1908          case form of
1909           scalar: if fsp=intptr then write(prr,'i')
1910                   else
1911                     if fsp=boolptr then write(prr,'b')
1912                     else
```



```
1913                       if fsp=charptr then write(prr,'c')
1914                       else
1915                         if scalkind = declared then write(prr,'i')
1916                         else write(prr,'r');
1917            subrange: gentypindicator(rangetype);
1918            pointer:  write(prr,'a');
1919            power:    write(prr,'s');
1920            records,arrays: write(prr,'m');
1921            files,tagfld,variant: error(500)
1922          end
1923    end (*typindicator*);
1924
1925    procedure gen0t(fop: oprange; fsp: stp);
1926    begin
1927      if prcode then
1928        begin putic;
1929          write(prr,mn[fop]:4);
1930          gentypindicator(fsp);
1931          writeln(prr);
1932        end;
1933      ic := ic + 1; mes(fop)
1934    end (*gen0t*);
1935
1936    procedure gen1t(fop: oprange; fp2: integer; fsp: stp);
1937    begin
1938      if prcode then
1939        begin putic;
1940          write(prr,mn[fop]:4);
1941          gentypindicator(fsp);
1942          writeln(prr,fp2:11)
1943        end;
1944      ic := ic + 1; mes(fop)
1945    end (*gen1t*);
1946
1947    procedure gen2t(fop: oprange; fp1,fp2: integer; fsp: stp);
1948    begin
1949      if prcode then
1950        begin putic;
1951          write(prr,mn[fop]: 4);
1952          gentypindicator(fsp);
1953          writeln(prr,fp1:3+5*ord(abs(fp1)>99),fp2:8);
1954        end;
1955      ic := ic + 1; mes(fop)
1956    end (*gen2t*);
1957
1958    procedure load;
1959    begin
1960      with gattr do
1961        if typtr <> nil then
1962          begin
1963            case kind of
1964              cst:   if (typtr^.form = scalar) and (typtr <> realptr) then
1965                       if typtr = boolptr then gen2(51(*ldc*),3,cval.ival)
1966                       else
1967                         if typtr=charptr then
1968                           gen2(51(*ldc*),6,cval.ival)
1969                         else gen2(51(*ldc*),1,cval.ival)
1970                     else
1971                       if typtr = nilptr then gen2(51(*ldc*),4,0)
1972                       else
1973                         if cstptrix >= cstoccmax then error(254)
1974                         else
1975                           begin cstptrix := cstptrix + 1;
1976                             cstptr[cstptrix] := cval.valp;
```

```
1977                         if typtr = realptr then
1978                           gen2(51(*ldc*),2,cstptrix)
1979                         else
1980                           gen2(51(*ldc*),5,cstptrix)
1981                       end;
1982          varbl: case access of
1983                   drct:  if vlevel<=1 then
1984                            gen1t(39(*ldo*),dplmt,typtr)
1985                          else gen2t(54(*lod*),level-vlevel,dplmt,typtr
1986                   indrct: gen1t(35(*ind*),idplmt,typtr);
1987                   inxd:  error(400)
1988                 end;
1989          expr:
1990        end;
1991        kind := expr
1992      end
1993  end (*load*) ;
1994
1995  procedure store(var fattr: attr);
1996  begin
1997    with fattr do
1998      if typtr <> nil then
1999        case access of
2000          drct:   if vlevel <= 1 then gen1t(43(*sro*),dplmt,typtr)
2001                  else gen2t(56(*str*),level-vlevel,dplmt,typtr);
2002          indrct: if idplmt <> 0 then error(400)
2003                  else gen0t(26(*sto*),typtr);
2004          inxd:   error(400)
2005        end
2006  end (*store*) ;
2007
2008  procedure loadaddress;
2009  begin
2010    with gattr do
2011      if typtr <> nil then
2012        begin
2013          case kind of
2014            cst:  if string(typtr) then
2015                    if cstptrix >= cstoccmax then error(254)
2016                    else
2017                      begin cstptrix := cstptrix + 1;
2018                        cstptr[cstptrix] := cval.valp;
2019                        gen1(38(*lca*),cstptrix)
2020                      end
2021                  else error(400);
2022            varbl: case access of
2023                     drct:  if vlevel <= 1 then gen1(37(*lao*),dplmt)
2024                            else gen2(50(*lda*),level-vlevel,dplmt);
2025                     indrct: if idplmt <> 0 then
2026                               gen1t(34(*inc*),idplmt,nilptr);
2027                     inxd:  error(400)
2028                   end;
2029            expr:  error(400)
2030          end;
2031          kind := varbl; access := indrct; idplmt := 0
2032        end
2033  end (*loadaddress*) ;
2034
2035
2036  procedure genfjp(faddr: integer);
2037  begin load;
2038    if gattr.typtr <> nil then
2039      if gattr.typtr <> boolptr then error(144);
2040    if prcode then begin putic; writeln(prr,mn[33]:4,' l':8,faddr:4) end;
```



```
2041    ic := ic + 1; mes(33)
2042  end (*genfjp*) ,
2043
2044  procedure genujpxjp(fop: oprange; fp2: integer);
2045  begin
2046   if prcode then
2047     begin putic; writeln(prr, mn[fop]:4, ' l':8,fp2:4) end;
2048   ic := ic + 1; mes(fop)
2049  end (*genujpxjp*);
2050
2051
2052  procedure gencupent(fop: oprange; fp1,fp2: integer);
2053  begin
2054    if prcode then
2055      begin putic;
2056        writeln(prr,mn[fop]:4,fp1:4,'l':4,fp2:4)
2057      end;
2058    ic := ic + 1; mes(fop)
2059  end;
2060
2061
2062  procedure checkbnds(fsp: stp);
2063    var lmin,lmax: integer;
2064  begin
2065    if fsp <> nil then
2066      if fsp <> intptr then
2067        if fsp <> realptr then
2068          if fsp^.form <= subrange then
2069            begin
2070              getbounds(fsp,lmin,lmax);
2071              gen2t(45(*chk*),lmin,lmax,fsp)
2072            end
2073  end (*checkbnds*),
2074
2075
2076  procedure putlabel(labname: integer);
2077  begin if prcode then writeln(prr, 'l', labname:4)
2078  end (*putlabel*);
2079
2080  procedure statement(fsys: setofsys);
2081    label 1;
2082    var lcp: ctp; llp: lbp;
2083
2084    procedure expression(fsys: setofsys); forward;
2085
2086    procedure selector(fsys: setofsys; fcp: ctp);
2087    var lattr: attr; lcp: ctp; lsize,lmin,lmax: integer;
2088    begin
2089      with fcp^, gattr do
2090        begin typtr := idtype; kind := varbl;
2091          case klass of
2092            vars:
2093              if vkind = actual then
2094                begin access := drct; vlevel := vlev;
2095                  dplmt := vaddr
2096                end
2097              else
2098                begin gen2t(54(*lod*),level-vlev,vaddr,nilptr);
2099                  access := indrct; idplmt := 0
2100                end;
2101            field:
2102              with display[disx] do
2103                if occur = crec then
2104                  begin access := drct; vlevel := clev;
```

```
3385        store(lattr); genujpxjp(57(*ujp*),laddr); putlabel(lcix);
3386        lc := llc;
3387      end (*forstatement*) ;
3388
3389
3390      procedure withstatement;
3391        var lcp: ctp; lcnt1: disprange; llc: addrrange;
3392      begin lcnt1 := 0; llc := lc;
3393        repeat
3394          if sy = ident then
3395            begin searchid([vars,field],lcp), insymbol end
3396          else begin error(2); lcp := uvarptr end;
3397          selector(fsys + [comma,dosy],lcp);
3398          if gattr.typtr <> nil then
3399            if gattr.typtr^.form = records then
3400              if top < displimit then
3401                begin top := top + 1; lcnt1 := lcnt1 + 1;
3402                  with display[top] do
3403                    begin fname := gattr.typtr^.fstfld;
3404                      flabel := nil
3405                    end;
3406                  if gattr.access = drct then
3407                    with display[top] do
3408                      begin occur := crec; clev := gattr.vlevel;
3409                        cdspl := gattr.dplmt
3410                      end
3411                  else
3412                    begin loadaddress;
3413                      align(nilptr,lc);
3414                      gen2t(56(*str*),0,lc,nilptr);
3415                      with display[top] do
3416                        begin occur := vrec; vdspl := lc end;
3417                      lc := lc+ptrsize;
3418                      if lc > lcmax then lcmax := lc
3419                    end
3420                end
3421              else error(250)
3422            else error(140);
3423          test := sy <> comma;
3424          if not test then insymbol
3425        until test;
3426        if sy = dosy then insymbol else error(54);
3427        statement(fsys);
3428        top := top-lcnt1; lc := llc;
3429      end (*withstatement*) ;
3430
3431    begin (*statement*)
3432      if sy = intconst then (*label*)
3433        begin llp := display[level].flabel;
3434          while llp <> nil do
3435            with llp^ do
3436              if labval = val.ival then
3437                begin if defined then error(165);
3438                  putlabel(labname); defined := true;
3439                  goto 1
3440                end
3441              else llp := nextlab;
3442          error(167);
3443  1:      insymbol;
3444          if sy = colon then insymbol else error(5)
3445        end;
3446      if not (sy in fsys + [ident]) then
3447        begin error(6); skip(fsys) end;
3448      if sy in statbegsys + [ident] then
```



```
3449        begin
3450          case sy of
3451            ident:    begin searchid([vars,field,func,proc],lcp), insymbol;
3452                        if lcp^.klass = proc then call(fsys,lcp)
3453                        else assignment(lcp)
3454                      end;
3455            beginsy:  begin insymbol; compoundstatement end;
3456            gotosy:   begin insymbol; gotostatement end;
3457            ifsy:     begin insymbol; ifstatement end;
3458            casesy:   begin insymbol; casestatement end;
3459            whilesy:  begin insymbol; whilestatement end;
3460            repeatsy: begin insymbol; repeatstatement end;
3461            forsy:    begin insymbol; forstatement end;
3462            withsy:   begin insymbol; withstatement end
3463          end;
3464          if not (sy in [semicolon,endsy,elsesy,untilsy]) then
3465            begin error(6); skip(fsys) end
3466        end
3467    end (*statement*) ;
3468
3469  begin (*body*)
3470    if fprocp <> nil then entname := fprocp^.pfname
3471    else genlabel(entname);
3472    cstptrix := 0; topnew := lcaftermarkstack; topmax := lcaftermarkstack;
3473    putlabel(entname); genlabel(segsize); genlabel(stacktop);
3474    gencupent(32(*ent1*),1,segsize); gencupent(32(*ent2*),2,stacktop);
3475    if fprocp <> nil then (*copy multiple values into local cells*)
3476      begin llc1 := lcaftermarkstack;
3477        lcp := fprocp^.next;
3478        while lcp <> nil do
3479          with lcp^ do
3480            begin
3481              align(parmptr,llc1);
3482              if klass = vars then
3483                if idtype <> nil then
3484                  if idtype^.form > power then
3485                    begin
3486                      if vkind = actual then
3487                        begin
3488                          gen2(50(*lda*),0,vaddr);
3489                          gen2t(54(*lod*),0,llc1,nilptr);
3490                          gen1(40(*mov*),idtype^.size);
3491                        end;
3492                      llc1 := llc1 + ptrsize
3493                    end
3494                  else llc1 := llc1 + idtype^.size;
3495              lcp := lcp^.next;
3496            end;
3497      end;
3498    lcmax := lc;
3499    repeat
3500      repeat statement(fsys + [semicolon,endsy])
3501      until not (sy in statbegsys);
3502      test := sy <> semicolon;
3503      if not test then insymbol
3504    until test;
3505    if sy = endsy then insymbol else error(13);
3506    llp := display[top].flabel; (*test for undefined labels*)
3507    while llp <> nil do
3508      with llp^ do
3509        begin
3510          if not defined then
3511            begin error(168);
3512              writeln(output); writeln(output,' label ',labval);
```

# The Java Virtual Machine Specification

*Release 1.0 Beta*
*DRAFT*

*August 21, 1995*

© 1993, 1994, 1995 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This BETA quality release and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this release or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc. and the University of California, respectively. Third-party font software in this release is protected by copyright and licensed from Sun's Font Suppliers

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The release described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications

TRADEMARKS
Sun, Sun Microsystems, Sun Microsystems Computer Corporation, the Sun logo, the Sun Microsystems Computer Corporation logo, WebRunner, Java, FirstPerson and the FirstPerson logo and agent are trademarks or registered trademarks of Sun Microsystems, Inc. The "Duke" character is a trademark of Sun Microsystems, Inc. and Copyright (c) 1992-1995 Sun Microsystems, Inc. All Rights Reserved. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Comapny, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

# Contents

Preface ........ 5

Chapter 1: Java Virtual Machine Architecture ........ 7
- 1.1 Supported Data Types ........ 7
- 1.2 Registers ........ 7
- 1.3 Local Variables ........ 8
- 1.4 The Operand Stack ........ 8
- 1.5 Execution Environment ........ 8
- 1.6 Garbage Collected Heap ........ 9
- 1.7 Method Area ........ 10
- 1.8 The Java Instruction Set ........ 10
- 1.9 Limitations ........ 10

Chapter 2: Class File Format ........ 11
- 2.1 Format ........ 11
- 2.2 Signatures ........ 13
- 2.3 Constant Pool ........ 15
- 2.4 Fields ........ 19
- 2.5 Methods ........ 19
- 2.6 Attributes ........ 20

Chapter 3: The Virtual Machine Instruction Set ........ 27
- 3.1 Format for the Instructions ........ 27
- 3.2 Pushing Constants onto the Stack ........ 27
- 3.3 Loading Local Variables Onto the Stack ........ 30
- 3.4 Storing Stack Values Into Local Variables ........ 33
- 3.5 Wider index for Loading, Storing and Incrementing ........ 35
- 3.6 Managing Arrays ........ 36
- 3.7 Stack Instructions ........ 42
- 3.8 Arithmetic Instructions ........ 44
- 3.9 Logical Instructions ........ 50
- 3.10 Conversion Operations ........ 52
- 3.11 Control Transfer Instructions ........ 56
- 3.12 Function Return ........ 63
- 3.13 Table Jumping ........ 65
- 3.14 Manipulating Object Fields ........ 66
- 3.15 Method Invocation ........ 68
- 3.16 Exception Handling ........ 70
- 3.17 Miscellaneous Object Operations ........ 71
- 3.18 Monitors ........ 72

Appendix A: An Optimization ........ 73
- A.1 Constant Pool Resolution ........ 73
- A.2 Pushing Constants onto the Stack (_quick variants) ........ 74
- A.3 Managing Arrays (_quick variants) ........ 75
- A.4 Manipulating Object Fields (_quick variants) ........ 76
- A.5 Method Invocation (_quick variants) ........ 78
- A.6 Miscellaneous Object Operations (_quick variants) ........ 80

Index of Instructions ........ 83

12

# Preface

This document describes version 1.0 of the Java Virtual Machine and its instruction set. We have written this document to act as a specification for both compiler writers, who wish to target the machine, and as a specification for others who may wish to implement a compliant Java Virtual Machine.

The Java Virtual Machine is an imaginary machine that is implemented by emulating it in software on a real machine. Code for the Java Virtual Machine is stored in `.class` files, each of which contains the code for at most one `public` class.

Simple and efficient emulations of the Java Virtual Machine are possible because the machine's format is compact and efficient bytecodes. Implementations whose native code speed approximates that of compiled C are also possible, by translating the bytecodes to machine code, although Sun has not released such implementations at this time.

The rest of this document is structured as follows:

- Chapter 1 describes the architecture of the Java Virtual Machine.
- Chapter 2 describes the `.class` file format.
- Chapter 3 describes the bytecodes.
- Appendix A contains some instructions generated internally by Sun's implementation of the Java Virtual Machine. While not strictly part of the specification we describe these here so that this specification can serve as a reference for our implementation. As more implementations of the Java Virtual Machine become available, we may remove Appendix A from future releases.

Sun will license the Java Virtual Machine trademark and logo for use with compliant implementations of this specification. If you are considering constructing your own implementation of the Java Virtual Machine please contact us, at the email address below, so that we can work together to insure 100% compatibility of your implementation.

Send comments on this specification or questions about implementing the Java Virtual Machine to our electronic mail address: java@java.sun.com.

13

# 1 *Java Virtual Machine Architecture*

## 1.1 Supported Data Types

The virtual machine data types include the basic data types of the Java language:

```
byte          // 1-byte signed 2's complement integer
short         // 2-byte signed 2's complement integer
int           // 4-byte signed 2's complement integer
long          // 8-byte signed 2's complement integer
float         // 4-byte IEEE 754 single-precision float
double        // 8-byte IEEE 754 double-precision float
char          // 2-byte unsigned Unicode character
```

Nearly all Java type checking is done at compile time. Data of the primitive types shown above need not be tagged by the hardware to allow execution of Java. Instead, the bytecodes that operate on primitive values indicate the types of the operands so that, for example, the `iadd`, `ladd`, `fadd`, and dadd instructions each add two numbers, whose types are `int`, `long`, `float`, and `double`, respectively

The virtual machine doesn't have separate instructions for `boolean` types. Intead, integer instructions, including integer returns, are used to operate on `boolean` values; `byte` arrays are used for arrays of `boolean`.

The virtual machine specifies that floating point be done in IEEE 754 format, with support for gradual underflow. Older computer architectures that do not have support for IEEE format may run Java numeric programs very slowly.

Other virtual machine data types include:

```
object          // 4-byte reference to a Java object
returnAddress   // 4 bytes, used with jsr/ret/jsr_w/ret_w instructions
```

Note: Java arrays are treated as objects.

This specification does not require any particular internal structure for objects. In our implementation an object reference is to a handle, which is a pair of pointers: one to a method table for the object, and the other to the data allocated for the object. Other implementations may use inline caching, rather than method table dispatch; such methods are likely to be faster on hardware that is emerging between now and the year 2000.

Programs represented by Java Virtual Machine bytecodes are expected to maintain proper type discipline and an implementation may refuse to execute a bytecode program that appears to violate such type discipline.

While the Java Virtual Machines would appear to be limited by the bytecode definition to running on a 32-bit address space machine, it is possible to build a version of the Java Virtual Machine that automatically translates the bytecodes into a 64-bit form. A description of this transformation is beyond the scope of this specification.

## 1.2 Registers

At any point the virtual machine is executing the code of a single method, and the pc register contains the address of the next bytecode to be executed.

Each method has memory space allocated for it to hold:

- a set of local variables, referenced by a `vars` register,
- an operand stack, referenced by an `optop` register, and
- a execution environment structure, referenced by a `frame` register.

All of this space can be allocated at once, since the size of the local variables and operand stack are known at compile time, and the size of the execution environment structure is well-known to the interpreter.

All of these registers are 32 bits wide.



## 1.3 Local Variables

Each Java method uses a fixed-sized set of local variables. They are addressed as word offsets from the `vars` register. Local variables are all 32 bits wide.

Long integers and double precision floats are considered to take up two local variables but are addressed by the index of the first local variable. (For example, a local variable with index *n* containing a double precision float actually occupies storage at indices *n* and *n*+1.) The virtual machine specification does not require 64-bit values in local variables to be 64-bit aligned. Implementors are free to decide the appropriate way to divide long integers and double precision floats into two words.

Instructions are provided to load the values of local variables onto the operand stack and store values from the operand stack into local variables.

## 1.4 The Operand Stack

The machine instructions all take operands from an operand stack, operate on them, and return results to the stack. We chose a stack organization so that it would be easy to emulate the machine efficiently on machines with few or irregular registers such as the Intel 486.

The operand stack is 32 bits wide. It is used to pass parameters to methods and receive method results, as well as to supply parameters for operations and save operation results.

For example, the `iadd` instruction adds two integers together. It expects that the integers to be added are the top two words on the operand stack, pushed there by previous instructions. Both integers are popped from the stack, added, and their sum pushed back onto the operand stack. Subcomputations may be nested on the operand stack, and result in a single operand that can be used by the nesting computation.

Each primitive data type has specialized instructions that know how to operate on operands of that type. Each operand requires a single location on the stack, except for `long` and `double`, which require two locations

Operands must be operated on by operators appropriate to their type. It is illegal, for example, to push two `int`s and then treat them as a `long`. This restriction is enforced, in the Sun implementation, by the bytecode verifier. However, a small number of operations (the dup opcodes and swap) operate on runtime data areas as raw values of a given width without regard to type.

In our description of the virtual machine instructions below, the effect of an instruction's execution on the operand stack is represented textually, with the stack growing from left to right, and each 32-bit word separately represented. Thus:

Stack: ..., *value1*, *value2* ⇒ ..., *value3*

shows an operation that begins by having *value2* on top of the stack with *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the stack and replaced by *value3*, which has been calculated by the instruction. The remainder of the stack, represented by an ellipsis, is unaffected by the instruction's execution.

The types `long` and `double` take two 32-bit words on the operand stack:

Stack: ... ⇒ ..., *value-word1*, *value-word2*

This specification does not say how the two words are selected from the 64-bit `long` or `double` value; it is only necessary that a particular implementation be internally consistent.

## 1.5 Execution Environment

The information contained in the execution environment is used to do dynamic linking, normal method returns, and exception propagation.

### 1.5.1 Dynamic Linking

The execution environment contains references to the interpreter symbol table for the current method and current class, in support of dynamic linking of the method code. The class file code for a method refers to methods to be called and variables to be accessed symbolically. Dynamic linking translates these symbolic

54

method calls into actual method calls, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.

This late binding of the methods and variables makes changes in other classes that a method uses less likely to break this code.

### 1.5.2 Normal Method Returns

If execution of the current method completes normally, then a value is returned to the calling method. This occurs when the calling method executes a return instruction appropriate to the return type.

The execution environment is used in this case to restore the registers of the caller, with the program counter of the caller appropriately incremented to skip the method call instruction. Execution then continues in the calling method's execution environment.

### 1.5.3 Exception and Error Propagation

An exceptional condition, known in Java as an `Error` or `Exception`, which are subclasses of `Throwable`, may arise in a program because of:

- a dynamic linkage failure, such as a failure to find a needed class file,
- a run-time error, such as a reference through a null pointer,
- an asynchronous event, such as is thrown by `Thread.stop`, from another thread,
- the program using a `throw` statement.

When an exception occurs:

- A list of *catch clauses* associated with the current method is examined. Each `catch` clause describes the instruction range for which it is active, describes the type of exception that it is to handle, and has the address of the code to handle it.
- An exception matches a `catch` clause if the instruction that caused the exception is in the appropriate instruction range, and the exception type is a subtype of the type of exception that the `catch` clause handles. If a matching `catch` clause is found, the system branches to the specified handler. If no handler is found, the process is repeated until all the nested catch clauses of the current method have been exhausted.
- The order of the `catch` clauses in the list is important. The virtual machine execution continues at the first matching catch clause. Because Java code is structured, it is always possible to sort all the exception handlers for one method into a single list that, for any possible program counter value, can be searched in linear order to find the proper (innermost containing applicable) exception handler for an exception occuring at that program counter value.
- If there is no matching `catch` clause then the current method is said to have as its outcome the uncaught exception. The execution state of the method that called this method is restored from the execution environment, and the propagation of the exception continues, as though the exception had just occurred in this caller.

### 1.5.4 Additional Information

The execution environment may be extended with additional implementation-specific information, such as debugging information.

## 1.6 Garbage Collected Heap

The Java heap is the runtime data area from which class instances (objects) are allocated. The Java language is designed to be garbage collected — it does not give the programmer the ability to deallocate objects explicitly. Java does not presuppose any particular kind of garbage collection; various algorithms may be used depending on system requirements.



## 1.7 Method Area

The method area is analogous to the store for compiled code in conventional languages or the text segment in a UNIX process. It stores method code (compiled Java code) and symbol tables. In the current Java implementation, method code is not part of the garbage-collected heap, although this is planned for a future release.

## 1.8 The Java Instruction Set

An instruction in the Java instruction set consists of a one-byte *opcode* specifying the operation to be performed, and zero or more *operands* supplying parameters or data that will be used by the operation. Many instructions have no operands and consist only of an opcode.

The inner loop of the virtual machine execution is effectively:

```
do {
    fetch an opcode byte
    execute an action depending on the value of the opcode
} while (there is more to do);
```

The number and size of the additional operands is determined by the opcode. If an additional operand is more than one byte in size, then it is stored in *big-endian* order — high order byte first. For example, a 16-bit parameter is stored as two bytes whose value is:

```
first_byte * 256 + second_byte
```

The bytecode instruction stream is only byte-aligned, with the exception being the `tableswitch` and `lookupswitch` instructions, which force alignment to a 4-byte boundary within their instructions.

These decisions keep the virtual machine code for a compiled Java program compact and reflect a conscious bias in favor of compactness at some possible cost in performance.

## 1.9 Limitations

The per-class constant pool has a maximum of 65535 entries. This acts as an internal limit on the total complexity of a single class.

The amount of code per method is limited to 65535 bytes by the sizes of the indices in the code in the exception table, the line number table, and the local variable table. This may be fixed for 1.0beta2.

Besides this limit, the only other limitation of note is that the number of words of arguments in a method call is limited to 255.

# 2 Class File Format

This chapter documents the Java class (`.class`) file format.

Each class file contains the compiled version of either a Java class or a Java interface. Compliant Java interpreters must be capable of dealing with all class files that conform to the following specification.

A Java class file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two or four 8-bit bytes, respectively. The bytes are joined together in network (big-endian) order, where the high bytes come first. This format is supported by the Java `java.io.DataInput` and `java.io.DataOutput` interfaces, and classes such as `java.io.DataInputStream` and `java.io.DataOutputStream`.

The class file format is described here using a structure notation. Successive fields in the structure appear in the external representation without padding or alignment. Variable size arrays, often of variable sized elements are called tables and are commonplace in these structures.

The types `u1`, `u2`, and `u4` mean an unsigned one-, two-, or four-byte quantity, respectively, which are read by method such as `readUnsignedByte`, `readUnsignedShort` and `readInt` of the java.io.`DataInput` interface.

## 2.1 Format

The following pseudo-structure gives a top-level description of the format of a class file:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count - 1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attribute_count];
}
```

**`magic`**

This field must have the value 0xCAFEBABE.

**`minor_version, major_version`**

These fields contain the version number of the Java compiler that produced this class file. An implementation of the virtual machine will normally support some range of minor version numbers 0-*n* of a particular major version number. If the minor version number is



incremented the new code won't run on the old virtual machines, but it is possible to make a new virtual machine which can run versions up to *n*+1.

A change of the major version number indicates a major incompatible change, one that requires a different virtual machine that may not support the old major version in any way.

The current major version number is 45; the current minor version number is 3.

**`constant_pool_count`**

This field indicates the number of entries in the constant pool in the class file.

**`constant_pool`**

The constant pool is an table of values. These values are the various string constants, class names, field names, and others that are referred to by the class structure or by the code.

`constant_pool[0]` is always unused by the compiler, and may be used by an implementation for any purpose.

Each of the `constant_pool` entries 1 through `constant_pool_count-1` is a variable-length entry, whose format is given by the first "tag" byte, as described in section 2.3.

**`access_flags`**

This field contains a mask of up to sixteen modifiers used with class, method, and field declarations. The same encoding is used on similar fields in `field_info` and `method_info` as described below. Here is the encoding:

| Flag Name | Value | Meaning | Used By |
|---|---|---|---|
| ACC_PUBLIC | 0x0001 | Visible to everyone | Class, Method, Variable |
| ACC_PRIVATE | 0x0002 | Visible only to the defining class | Method, Variable |
| ACC_PROTECTED | 0x0004 | Visible to subclasses | Method, Variable |
| ACC_STATIC | 0x0008 | Variable or method is static | Method, Variable |
| ACC_FINAL | 0x0010 | No further subclassing, overriding, or assignment after initialization | Class, Method, Variable |
| ACC_SYNCHRONIZED | 0x0020 | Wrap use in monitor lock | Method |
| ACC_VOLATILE | 0x0040 | Can't cache | Variable |
| ACC_TRANSIENT | 0x0080 | Not to be written or read by a persistent object manager | Variable |
| ACC_NATIVE | 0x0100 | Implemented in a language other than Java | Method |
| ACC_INTERFACE | 0x0200 | Is an interface | Class |
| ACC_ABSTRACT | 0x0400 | No body provided | Class, Method |

**`this_class`**

This field is an index into the constant pool; `constant_pool[this_class]` must be a CONSTANT_class.

**super_class**

This field is an index into the constant pool. If the value of `super_class` is nonzero, then `constant_pool[super_class]` must be a class, and gives the index of this class's superclass in the constant pool.

If the value of `super_class` is zero, then the class being defined must be `java.lang.Object`, and it has no superclass.

**interfaces_count**

This field gives the number of interfaces that this class implements.

**interfaces**

Each value in this table is an index into the constant pool. If an table value is nonzero (`interfaces[i] != 0`, where 0 <= *i* < `interfaces_count`), then `constant_pool[interfaces[i]]` must be an interface that this class implements.

Question: How could one of these entries ever be 0?

**fields_count**

This field gives the number of instance variables, both static and dynamic, defined by this class. The `fields` table includes only those variables that are defined explicitly by this class. It does not include those instance variables that are accessible from this class but are inherited from superclasses.

**fields**

Each value in this table is a more complete description of a field in the class. See section 2.4 for more information on the `field_info` structure.

**methods_count**

This field indicates the number of methods, both static and dynamic, defined by this class. This table only includes those methods that are explicitly defined by this class. It does not include inherited methods.

**methods**

Each value in this table is a more complete description of a method in the class. See section 2.5 for more information on the `method_info` structure.

**attributes_count**

This field indicates the number of additional attributes about this class.

**attributes**

A class can have any number of optional attributes associated with it. Currently, the only class attribute recognized is the "SourceFile" attribute, which indicates the name of the source file from which this class file was compiled. See section 2.6 for more information on the `attribute_info` structure.

## 2.2 Signatures

A signature is a string representing a type of a method, field or array.



The field signature represents the value of an argument to a function or the value of a variable. It is a series of bytes generated by the following grammar:

```
<field_signature>   ::= <field_type>
<field_type>        ::= <base_type>|<object_type>|<array_type>
<base_type>         ::= B|C|D|F|I|J|S|Z
<object_type>       ::= L<fullclassname>;
<array_type>        ::= [<optional_size><field_type>
<optional_size>     ::= [0-9]*
```

The meaning of the base types is as follows:

| | | |
|---|---|---|
| B | `byte` | signed byte |
| C | `char` | character |
| D | `double` | double precision IEEE float |
| F | `float` | single precision IEEE float |
| I | `int` | integer |
| J | `long` | long integer |
| L<fullclassname>; | ... | an object of the given class |
| S | `short` | signed short |
| Z | `boolean` | true or false |
| [<field sig> | ... | array |

A return-type signature represents the return value from a method. It is a series of bytes in the following grammar:

```
<return_signature>   ::= <field_type> | V
```

The character V indicates that the method returns no value. Otherwise, the signature indicates the type of the return value.

An argument signature represents an argument passed to a method:

```
<argument_signature> ::= <field_type>
```

A method signature represents the arguments that the method expects, and the value that it returns.

```
<method_signature>    ::= (<arguments_signature>) <return_signature>
<arguments_signature>::= <argument_signature>*
```

## 2.3 Constant Pool

Each item in the constant pool begins with a 1-byte tag:. The table below lists the valid tags and their values.

| Constant Type | Value |
| --- | --- |
| CONSTANT_Class | 7 |
| CONSTANT_Fieldref | 9 |
| CONSTANT_Methodref | 10 |
| CONSTANT_InterfaceMethodref | 11 |
| CONSTANT_String | 8 |
| CONSTANT_Integer | 3 |
| CONSTANT_Float | 4 |
| CONSTANT_Long | 5 |
| CONSTANT_Double | 6 |
| CONSTANT_NameAndType | 12 |
| CONSTANT_Utf8 | 1 |
| CONSTANT_Unicode | 2 |

Each tag byte is then followed by one or more bytes giving more information about the specific constant.

### 2.3.1 CONSTANT_Class

CONSTANT_Class is used to represent a class or an interface.

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

**tag**

The tag will have the value CONSTANT_Class

**name_index**

constant_pool[name_index] is a CONSTANT_Utf8 giving the string name of the class.

Because arrays are objects, the opcodes anewarray and multianewarray can reference array "classes" via CONSTANT_Class items in the constant pool. In this case, the name of the class is its signature. For example, the class name for

```
int[][]
```

is

```
[[I
```

The class name for

```
Thread[]
```

is

```
"[Ljava.lang.Thread;"
```

### 2.3.2 CONSTANT_{Fieldref,Methodref,InterfaceMethodref}

Fields, methods, and interface methods are represented by similar structures.



```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

**tag**

The tag will have the value CONSTANT_Fieldref, CONSTANT_Methodref, or CONSTANT_InterfaceMethodref.

**class_index**

constant_pool[class_index] will be an entry of type CONSTANT_Class giving the name of the class or interface containing the field or method.

For CONSTANT_Fieldref and CONSTANT_Methodref, the CONSTANT_Class item must be an actual class. For CONSTANT_InterfaceMethodref, the item must be an interface which purports to implement the given method.

**name_and_type_index**

constant_pool[name_and_type_index] will be an entry of type CONSTANT_NameAndType. This constant pool entry indicates the name and signature of the field or method.

### 2.3.3 CONSTANT_String

CONSTANT_String is used to represent constant objects of the built-in type String.

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

**tag**

The tag will have the value CONSTANT_String

**string_index**

constant_pool[string_index] is a CONSTANT_Utf8 string giving the value to which the String object is initialized.

### 2.3.4 CONSTANT_Integer and CONSTANT_Float

CONSTANT_Integer and CONSTANT_Float represent four-byte constants.

18

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}

CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

**tag**

The tag will have the value `CONSTANT_Integer` or `CONSTANT_Float`

**bytes**

For integers, the four bytes are the integer value. For floats, they are the IEEE 754 standard representation of the floating point value. These bytes are in network (high byte first) order.

### 2.3.5 CONSTANT_Long and CONSTANT_Double

`CONSTANT_Long` and `CONSTANT_Double` represent eight-byte constants.

```
CONSTANT_Long_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}

CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

All eight-byte constants take up two spots in the constant pool. If this is the $n^{th}$ item in the constant pool, then the next item will be numbered n+2.

**tag**

The tag will have the value `CONSTANT_Long` or `CONSTANT_Double`.

**high_bytes, low_bytes**

For `CONSTANT_Long`, the 64-bit value is (`high_bytes` << 32) + `low_bytes`.

For `CONSTANT_Double`, the 64-bit value, `high_bytes` and `low_bytes` together represent the standard IEEE 754 representation of the double-precision floating point number.

### 2.3.6 CONSTANT_NameAndType

`CONSTANT_NameAndType` is used to represent a field or method, without indicating which class it belongs to.

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 signature_index;
}
```

**tag**

The tag will have the value `CONSTANT_NameAndType`.



**name_index**

`constant_pool[name_index]` is a `CONSTANT_Utf8` string giving the name of the field or method.

**signature_index**

`constant_pool[signature_index]` is a `CONSTANT_Utf8` string giving the signature of the field or method.

### 2.3.7 CONSTANT_Utf8 and CONSTANT_Unicode

`CONSTANT_Utf8` and `CONSTANT_Unicode` are used to represent constant string values.

`CONSTANT_Utf8` strings are "encoded" so that strings containing only non-null ASCII characters, can be represented using only one byte per character, but characters of up to 16 bits can be represented:

All characters in the range 0x0001 to 0x007F are represented by a single byte:

```
+-+-+-+-+-+-+-+-+
|0|7bits of data|
+-+-+-+-+-+-+-+-+
```

The null character (0x0000) and characters in the range 0x0080 to 0x07FF are represented by a pair of two bytes:

```
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
|1|1|0| 5 bits  |   |1|0|  6 bits   |
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
```

Characters in the range 0x0800 to 0xFFFF are represented by three bytes:

```
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
|1|1|1|0|4 bits |   |1|0|  6 bits   |   |1|0|  6 bits   |
+-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
```

There are two differences between this format and the "standard" UTF-8 format. First, the null byte (0x00) is encoded in two-byte format rather than one-byte, so that our strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats are used. We do not recognize the longer formats.

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}

CONSTANT_Unicode_info {
    u1 tag;
    u2 length;
    u2 bytes[length];
}
```

**tag**

The tag will have the value `CONSTANT_Utf8` or `CONSTANT_Unicode`.

**length**

The number of bytes in the string. These strings are not null terminated.

**bytes**

The actual bytes of the string.

## 2.4 Fields

The information for each field immediately follows the field_count field in the class file. Each field is described by a variable length field_info structure. The format of this structure is as follows:

```
field_info {
   u2 access_flags;
   u2 name_index;
   u2 signature_index;
   u2 attributes_count;
   attribute_info attributes[attribute_count];
}
```

**access_flags**

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table "Access Flags" on page 12 which indicates the meaning of the bits in this field.

The possible fields that can be set for a field are ACC_PUBLIC, ACC_PRIVATE, ACC_PROTECTED, ACC_STATIC, ACC_FINAL, ACC_VOLATILE, and ACC_TRANSIENT.

At most one of ACC_PUBLIC, ACC_PROTECTED, and ACC_PRIVATE can be set for any method.

**name_index**

constant_pool[name_index] is a CONSTANT_Utf8 string which is the name of the field.

**signature_index**

constant_pool[signature_index] is a CONSTANT_Utf8 string which is the signature of the field. See the section "Signatures" for more information on signatures.

**attributes_count**

This value indicates the number of additional attributes about this field.

**attributes**

A field can have any number of optional attributes associated with it. Currently, the only field attribute recognized is the "ConstantValue" attribute, which indicates that this field is a static numeric constant, and indicates the constant value of that field.

Any other attributes are skipped.

## 2.5 Methods

The information for each method immediately follows the method_count field in the class file. Each method is described by a variable length method_info structure. The structure has the following format:



```
method_info {
   u2 access_flags;
   u2 name_index;
   u2 signature_index;
   u2 attributes_count;
   attribute_info attributes[attribute_count];
}
```

**access_flags**

This is a set of sixteen flags used by classes, methods, and fields to describe various properties and how they many be accessed by methods in other classes. See the table "Access Flags" on page 12 which gives the various bits in this field.

The possible fields that can be set for a method are ACC_PUBLIC, ACC_PRIVATE, ACC_PROTECTED, ACC_STATIC, ACC_FINAL, ACC_SYNCHRONIZED, ACC_NATIVE, and ACC_ABSTRACT.

At most one of ACC_PUBLIC, ACC_PROTECTED, and ACC_PRIVATE can be set for any method.

**name_index**

constant_pool[name_index] is a CONSTANT_Utf8 string giving the name of the method.

**signature_index**

constant_pool[signature_index] is a CONSTANT_Utf8 string giving the signature of the field. See the section "Signatures" for more information on signatures.

**attributes_count**

This value indicates the number of additional attributes about this field.

**attributes**

A field can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only field attributes recognized are the "Code" and "Exceptions" attributes, which describe the bytecodes that are executed to perform this method, and the Java Exceptions which are declared to result from the execution of the method, respectively.

Any other attributes are skipped.

## 2.6 Attributes

Attributes are used at several different places in the class format. All attributes have the following format:

```
GenericAttribute_info {
   u2 attribute_name;
   u4 attribute_length;
   u1 info[attribute_length];
}
```

The attribute_name is a 16-bit index into the class's constant pool; the value of constant_pool[attribute_name] is a CONSTANT_Utf8 string giving the name of the attribute. The field attribute_length indicates the length of the subsequent information in bytes. This length does not include the six bytes of the attribute_name and attribute_length.

In the following text, whenever we allow attributes, we give the name of the attributes that are currently understood. In the future, more attributes will be added. Class file readers are expected to skip over and ignore the information in any attribute they do not understand.

### 2.6.1 SourceFile

The "SourceFile" attribute has the following format:

```
SourceFile_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 sourcefile_index;
}
```

**attribute_name_index**

constant_pool[attribute_name_index] is the CONSTANT_Utf8 string "SourceFile".

**attribute_length**

The length of a SourceFile_attribute must be 2.

**sourcefile_index**

constant_pool[sourcefile_index] is a CONSTANT_Utf8 string giving the source file from which this class file was compiled.

### 2.6.2 ConstantValue

The "ConstantValue" attribute has the following format:

```
ConstantValue_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 constantvalue_index;
}
```

**attribute_name_index**

constant_pool[attribute_name_index] is the CONSTANT_Utf8 string "ConstantValue".

**attribute_length**

The length of a ConstantValue_attribute must be 2.

**constantvalue_index**

constant_pool[constantvalue_index] gives the constant value for this field.

The constant pool entry must be of a type appropriate to the field, as shown by the following table:

| | |
|---|---|
| long | CONSTANT_Long |
| float | CONSTANT_Float |
| double | CONSTANT_Double |
| int, short, char, byte, boolean | CONSTANT_Integer |

### 2.6.3 Code

The "Code" attribute has the following format:



```
Code_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 max_stack;
   u2 max_locals;
   u4 code_length;
   u1 code[code_length];
   u2 exception_table_length;
   {  u2    start_pc;
      u2    end_pc;
      u2    handler_pc;
      u2    catch_type;
   }  exception_table[exception_table_length];
   u2 attributes_count;
   attribute_info attributes[attribute_count];
}
```

**attribute_name_index**

constant_pool[attribute_name_index] is the CONSTANT_Utf8 string "Code".

**attribute_length**

This field indicates the total length of the "Code" attribute, excluding the initial six bytes.

**max_stack**

Maximum number of entries on the operand stack that will be used during execution of this method. See the other chapters in this spec for more information on the operand stack.

**max_locals**

Number of local variable slots used by this method. See the other chapters in this spec for more information on the local variables.

**code_length**

The number of bytes in the virtual machine code for this method.

**code**

These are the actual bytes of the virtual machine code that implement the method. When read into memory, if the first byte of code is aligned onto a multiple-of-four boundary the the tableswitch and tablelookup opcode entries will be aligned; see their description for more information on alignment requirements.

**exception_table_length**

The number of entries in the following exception table.

**exception_table**

Each entry in the exception table describes one exception handler in the code.

**start_pc, end_pc**

The two fields start_pc and end_pc indicate the ranges in the code at which the exception handler is active. The values of both fields are offsets from the start of the code. start_pc is inclusive. end_pc is exclusive.

**handler_pc**

This field indicates the starting address of the exception handler. The value of the field is an offset from the start of the code.

`catch_type`

If `catch_type` is nonzero, then `constant_pool[catch_type]` will be the class of exceptions that this exception handler is designated to catch. This exception handler should only be called if the thrown exception is an instance of the given class.

If catch_type is zero, this exception handler should be called for all exceptions.

`attributes_count`

This field indicates the number of additional attributes about code. The "Code" attribute can itself have attributes.

`attributes`

A "Code" attribute can have any number of optional attributes associated with it. Each attribute has a name, and other additional information. Currently, the only code attributes defined are the "LineNumberTable" and "LocalVariableTable," both of which contain debugging information.

### 2.6.4 Exceptions Table

This table is used by compilers which indicate which Exceptions a method is declared to throw:

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

`attribute_name_index`

`constant_pool[attribute_name_index]` will be the CONSTANT_Utf8 string "`Exceptions`".

`attribute_length`

This field indicates the total length of the Exceptions_attribute, excluding the initial six bytes.

`number_of_exceptions`

This field indicates the number of entries in the following exception index table.

`exception_index_table`

Each value in this table is an index into the constant pool. For each table element (`exception_index_table[`*i*`]` != 0, where 0 <= *i* < `number_of_exceptions`), then `constant_pool[exception_index+table[`*i*`]]` is a Exception that this class is declared to throw.

### 2.6.5 LineNumberTable

This attribute is used by debuggers and the exception handler to determine which part of the virtual machine code corresponds to a given location in the source. The LineNumberTable_attribute has the following format:



```
LineNumberTable_attribute {
    u2              attribute_name_index;
    u4              attribute_length;
    u2              line_number_table_length;
    {   u2              start_pc;
        u2              line_number;
    }               line_number_table[line_number_table_length];
}
```

`attribute_name_index`

`constant_pool[attribute_name_index]` will be the CONSTANT_Utf8 string "`LineNumberTable`".

`attribute_length`

This field indicates the total length of the LineNumberTable_attribute, excluding the initial six bytes.

`line_number_table_length`

This field indicates the number of entries in the following line number table

`line_number_table`

Each entry in the line number table indicates that the line number in the source file changes at a given point in the code.

`start_pc`

This field indicates the place in the code at which the code for a new line in the source begins. `source_pc` <<SHOULD THAT BE `start_pc`?>> is an offset from the beginning of the code.

`line_number`

The line number that begins at the given location in the file.

### 2.6.6 LocalVariableTable

This attribute is used by debuggers to determine the value of a given local variable during the dynamic execution of a method. The format of the LocalVariableTable_attribute is as follows:

```
LocalVariableTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    {   u2    start_pc;
        u2   length;
        u2   name_index;
        u2   signature_index;
        u2   slot;
    }   local_variable_table[local_variable_table_length];
}
```

`attribute_name_index`

`constant_pool[attribute_name_index]` will be the CONSTANT_Utf8 string "`LocalVariableTable`".

`attribute_length`

This field indicates the total length of the LineNumberTable_attribute, excluding the initial six bytes.

`local_variable_table_length`

This field indicates the number of entries in the following local variable table.

`local_variable_table`

Each entry in the local variable table indicates a code range during which a local variable has a value. It also indicates where on the stack the value of that variable can be found.

`start_pc, length`

The given local variable will have a value at the code between `start_pc` and `start_pc + length`. The two values are both offsets from the beginning of the code.

`name_index, signature_index`

`constant_pool[name_index]` and `constant_pool[signature_index]` are `CONSTANT_Utf8` strings giving the name and signature of the local variable.

`slot`

The given variable will be the *slot*th local variable in the method's frame.

# 3 The Virtual Machine Instruction Set

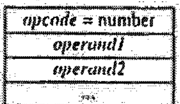
## 3.1 Format for the Instructions

Java Virtual Machine instructions are represented in this document by an entry of the following form.

### instruction name

*Short description* of the instruction

Syntax:




Stack: ...., *value1*, *value2* ⇒ ..., *value3*

A *longer description* that explains the functions of the instruction and indicates any exceptions that might be thrown during execution.

Each line in the syntax diagram represents a single 8-bit byte.

Operations of the Java Virtual Machine most often take their operands from the stack and put their results back on the stack. As a convention, the descriptions do not usually mention when the stack is the source or destination of an operation, but will always mention when it is not. For instance, the `iload` instruction has the short description "Load integer from local variable." Implicitly, the integer is loaded onto the stack. The `iadd` instruction is described as "Integer add"; both its source and destination are the stack.

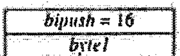
Instructions that do not affect the control flow of a computation may be assumed to always advance the virtual machine pc to the opcode of the following instruction. Only instructions that do affect control flow will explicitly mention the effect they have on pc.

## 3.2 Pushing Constants onto the Stack

### bipush

Push one-byte signed integer

Syntax:




Stack: ... => ..., *value*

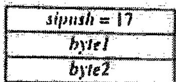
*byte1* is interpreted as a signed 8-bit *value*. This *value* is expanded to an integer and pushed onto the operand stack.



### sipush

Push two-byte signed integer

Syntax:



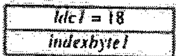

Stack: ... => ..., *item*

*byte1* and *byte2* are assembled into a signed 16-bit *value*. This *value* is expanded to an integer and pushed onto the operand stack.

### ldc1

Push item from constant pool

Syntax:




Stack: ... => ..., *item*

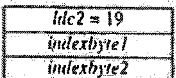
*indexbyte1* is used as an unsigned 8-bit index into the constant pool of the current class. The *item* at that index is resolved and pushed onto the stack. If a `String` is being pushed and there isn't enough memory to allocate space for it then an `OutOfMemoryError` is thrown.

Note: A String push results in a reference to an object; what other constants do, and explain this somewhere here.

### ldc2

Push item from constant pool

Syntax:




Stack: ... => ..., *item*

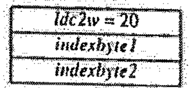
*indexbyte1* and *indexbyte2* are used to construct an unsigned 16-bit index into the constant pool of the current class. The *item* at that index is resolved and pushed onto the stack. If a `String` is being pushed and there isn't enough memory to allocate space for it then an `OutOfMemoryError` is thrown.

Note: A String push results in a reference to an object; what other constants do, and explain this somewhere here.

### ldc2w

Push long or double from constant pool

Syntax:



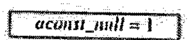

Stack: ... => ..., *constant-word1*, *constant-word2*

*indexbyte1* and *indexbyte2* are used to construct an unsigned 16-bit index into the constant pool of the current class. The two-word *constant* at that index is resolved and pushed onto the stack.

### aconst_null

Push null object reference

Syntax:



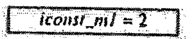

Stack: ... => ..., *null*

Push the `null` object reference onto the stack.

### iconst_m1

Push integer constant -1

Syntax:



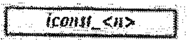

Stack: ... => ..., -1

Push the integer -1 onto the stack.

### iconst_<n>

Push integer constant

Syntax:




Stack: ... => ..., <n>

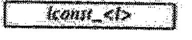
Forms: `iconst_0` = 3, `iconst_1` = 4, `iconst_2` = 5, `iconst_3` = 6, `iconst_4` = 7, `iconst_5` = 8

Push the integer <n> onto the stack.

### lconst_<l>

Push long integer constant

Syntax:




Stack: ... => ..., <l>-*word1*, <l>-*word2*

Forms: `lconst_0` = 9, `lconst_1` = 10

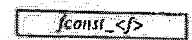
Push the long integer <l> onto the stack.



### fconst_<f>

Push single float

Syntax:




Stack: ... => ..., <f>

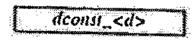
Forms: `fconst_0` = 11, `fconst_1` = 12, `fconst_2` = 13

Push the single-precision floating point number <f> onto the stack

### dconst_<d>

Push double float

Syntax:




Stack: ... => ..., <d>-*word1*, <d>-*word2*

Forms: `dconst_0` = 14, `dconst_1` = 15

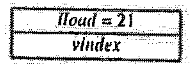
Push the double-precision floating point number <d> onto the stack.

## 3.3 Loading Local Variables Onto the Stack

### iload

Load integer from local variable

Syntax:



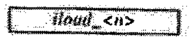

Stack: ... => ..., *value*

The *value* of the local variable at *vindex* in the current Java frame is pushed onto the operand stack.

### iload_<n>

Load integer from local variable

Syntax:




Stack: ... => ..., *value*

Forms: `iload_0` = 26, `iload_1` = 27, `iload_2` = 28, `iload_3` = 29

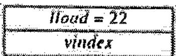
The *value* of the local variable at <n> in the current Java frame is pushed onto the operand stack.

This instruction is the same as `iload` with a *vindex* of <n>, except that the operand <n> is implicit.

## lload

Load long integer from local variable

Syntax:



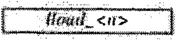

Stack: ... => ..., *value-word1*, *value-word2*

The *value* of the local variables at *vindex* and *vindex*+1 in the current Java frame is pushed onto the operand stack.

## lload_<n>

Load long integer from local variable

Syntax:




Stack: ... => ..., *value-word1*, *value-word2*

Forms: `lload_0` = 30, `lload_1` = 31, `lload_2` = 32, `lload_3` = 33

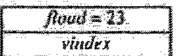
The *value* of the local variables at <*n*> and <*n*>+1 in the current Java frame is pushed onto the operand stack.

This instruction is the same as `lload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

## fload

Load single float from local variable

Syntax:



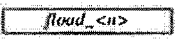

Stack: ... => ..., *value*

The *value* of the local variable at *vindex* in the current Java frame is pushed onto the operand stack.

## fload_<n>

Load single float from local variable

Syntax:




Stack: ... => ..., *value*

Forms: `fload_0` = 34, `fload_1` = 35, `fload_2` = 36, `fload_3` = 37

The *value* of the local variable at <*n*>in the current Java frame is pushed onto the operand stack.

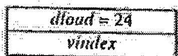
This instruction is the same as `fload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.



## dload

Load double float from local variable

Syntax:




Stack: ... => ..., *value-word1*, *value-word2*

The *value* of the local variables at *vindex* and *vindex*+1 in the current Java frame is pushed onto the operand stack.

## dload_<n>

Load double float from local variable

Syntax:

Stack: ... => ..., *value-word1*, *value-word2*

Forms: `dload_0` = 38, `dload_1` = 39, `dload_2` = 40, `dload_3` = 41

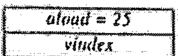
The *value* of the local variables at <*n*> and <*n*>+1 in the current Java frame is pushed onto the operand stack.

This instruction is the same as `dload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

## aload

Load object reference from local variable

Syntax:




Stack: ... => ..., *value*

The *value* of the local variable at *vindex* in the current Java frame is pushed onto the operand stack.

## aload_<n>

Load object reference from local variable

Syntax:

Stack: ... => ..., *value*

Forms: `aload_0` = 42, `aload_1` = 43, `aload_2` = 44, `aload_3` = 45

The *value* of the local variable at <*n*>in the current Java frame is pushed onto the operand stack.

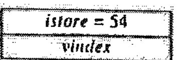
This instruction is the same as `aload` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

## 3.4 Storing Stack Values into Local Variables

### istore

Store integer into local variable

Syntax:




Stack: ..., *value* => ...

*value* must be an integer. Local variable *vindex* in the current Java frame is set to *value*.

### istore_<n>

Store integer into local variable

Syntax:

Stack: ..., *value* => ...

Forms: `istore_0` = 59, `istore_1` = 60, `istore_2` = 61, `istore_3` = 62

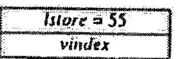
*value* must be an integer. Local variable <*n*> in the current Java frame is set to *value*.

This instruction is the same as `istore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

### lstore

Store long integer into local variable

Syntax:




Stack: ..., *value-word1*, *value-word2* => ...

*value* must be a long integer. Local variables *vindex* and *vindex*+1 in the current Java frame are set to *value*.

### lstore_<n>

Store long integer into local variable

Syntax:

Stack: ..., *value-word1*, *value-word2* => ...

Forms: `lstore_0` = 63, `lstore_1` = 64, `lstore_2` = 65, `lstore_3` = 66

*value* must be a long integer. Local variables <*n*> and <*n*>+1 in the current Java frame are set to *value*.

This instruction is the same as `lstore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.



### fstore

Store single float into local variable

Syntax:

Stack: ..., *value* => ...

*value* must be a single-precision floating point number. Local variable *vindex* in the current Java frame is set to *value*.

### fstore_<n>

Store single float into local variable

Syntax:

Stack: ..., *value* => ...

Forms: `fstore_0` = 67, `fstore_1` = 68, `fstore_2` = 69, `fstore_3` = 70

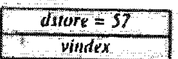
*value* must be a single-precision floating point number. Local variable <*n*> in the current Java frame is set to *value*.

This instruction is the same as `fstore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

### dstore

Store double float into local variable

Syntax:



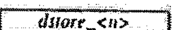

Stack: ..., *value-word1*, *value-word2* => ...

*value* must be a double-precision floating point number. Local variables *vindex* and *vindex*+1 in the current Java frame are set to *value*.

### dstore_<n>

Store double float into local variable

Syntax:




Stack: ..., *value-word1*, *value-word2* => ...

Forms: `dstore_0` = 71, `dstore_1` = 72, `dstore_2` = 73, `dstore_3` = 74

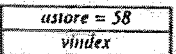
*value* must be a double-precision floating point number. Local variables <*n*> and <*n*>+1 in the current Java frame are set to *value*.

This instruction is the same as `dstore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit

### astore

Store object reference into local variable

Syntax:



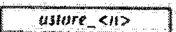

Stack: ..., *value* => ...

*value* must be a return address or a reference to an object. Local variable *vindex* in the current Java frame is set to *value*.

### astore_<n>

Store object reference into local variable

Syntax:




Stack: ..., *value* => ...

Forms: `astore_0` = 75, `astore_1` = 76, `astore_2` = 77, `astore_3` = 78

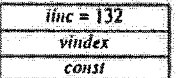
*value* must be a return address or a reference to an object. Local variable <*n*> in the current Java frame is set to *value*.

This instruction is the same as `astore` with a *vindex* of <*n*>, except that the operand <*n*> is implicit.

### iinc

Increment local variable by constant

Syntax:




Stack: no change

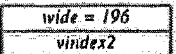
Local variable *vindex* in the current Java frame must contain an integer. Its value is incremented by the value *const*, where *const* is treated as a signed 8-bit quantity.

## 3.5 Wider index for Loading, Storing and Incrementing

### wide

Wider index for accessing local variables in load, store and increment.

Syntax:




Stack: no change

This bytecode must precede one of the following bytecodes: `iload`, `lload`, `fload`, `dload`, `aload`, `istore`, `lstore`, `fstore`, `dstore`, `astore`, `iinc`. The *vindex* of the following bytecode and *vindex2* from this bytecode are assembled into an unsigned 16-bit index to a local variable in the current Java frame. The following bytecode operates as normal except for the use of this wider index.

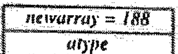


## 3.6 Managing Arrays

### newarray

Allocate new array

Syntax:




Stack: ..., *size* => *result*

*size* must be an integer. It represents the number of elements in the new array.

*atype* is an internal code that indicates the type of array to allocate. Possible values for *atype* are as follows:

| | |
|---|---|
| T_BOOLEAN | 4 |
| T_CHAR | 5 |
| T_FLOAT | 6 |
| T_DOUBLE | 7 |
| T_BYTE | 8 |
| T_SHORT | 9 |
| T_INT | 10 |
| T_LONG | 11 |

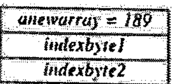
A new array of *atype*, capable of holding *size* elements, is allocated, and *result* is a reference to this new object. Allocation of an array large enough to contain *size* items of *atype* is attempted. All elements of the array are initialized to zero.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

## anewarray

Allocate new array of references to objects

Syntax:


| anewarray = 189 |
|---|
| indexbyte1 |
| indexbyte2 |

Stack: ..., *size*=> *result*

*size* must be an integer. It represents the number of elements in the new array.

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry must be a class.

A new array of the indicated class type and capable of holding *size* elements is allocated, and *result* is a reference to this new object. Allocation of an array large enough to contain *size* items of the given class type is attempted. All elements of the array are initialized to `null`.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

`anewarray` is used to create a single dimension of an array of object references. For example, to create

```
new Thread[7]
```

the following code is used:

```
bipush 7
anewarray <Class "java.lang.Thread">
```

`anewarray` can also be used to create the first dimension of a multi-dimensional array. For example, the following array declaration:

```
new int[6][]
```

is created with the following code:

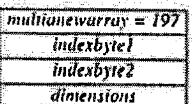
```
bipush 6
anewarray <Class "[I">
```

See `CONSTANT_Class` in the "Class File Format" chapter for information on array class names.

## multianewarray

Allocate new multi-dimensional array

Syntax:


| multianewarray = 197 |
|---|
| indexbyte1 |
| indexbyte2 |
| dimensions |

Stack: ..., *size1 size2...sizen* => *result*

Each *size* must be an integer. Each represents the number of elements in a dimension of the array.

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index is resolved. The resulting entry must be an array class of one or more dimensions.

*dimensions* has the following aspects:

- It must be an integer ≥ 1.
- It represents the number of dimensions being created. It must be ≤ the number of dimensions of the array class.



- It represents the number of elements that are popped off the stack. All must be integers greater than or equal to zero. These are used as the sizes of the dimension. For example, to create

  ```
  new int[6][3][]
  ```

  the following code is used:

  ```
  bipush 6
  bipush 3
  multianewarray <Class "[[[I"> 2
  ```

If any of the *size* arguments on the stack is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

The *result* is a reference to the new array object.

Note: More explanation needed about how this is an array of arrays.

Note: It is more efficient to use `newarray` or `anewarray` when creating a single dimension.

See `CONSTANT_Class` in the "Class File Format" chapter for information on array class names.

## arraylength

Get length of array

Syntax:

| arraylength = 190 |
|---|

Stack: ..., *objectref* => ..., *length*

*objectref* must be a reference to an array object. The length of the array is determined and replaces *objectref* on the top of the stack.

If the *objectref* is `null`, a `NullPointerException` is thrown.

## iaload

Load integer from array

Syntax:

| iaload = 46 |
|---|

Stack: ..., *arrayref, index* => ..., *value*

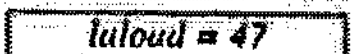
*arrayref* must be a reference to an array of integers. *index* must be an integer. The integer *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

## laload

Load long integer from array

Syntax:


| laload = 47 |
|---|

Stack: ..., *arrayref, index* => ..., *value-word1, value-word2*

*arrayref* must be a reference to an array of long integers. *index* must be an integer. The long integer *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### faload

Load single float from array

Syntax:

| *faload = 48* |
|---|

Stack: ..., *arrayref*, *index* => ..., *value*

*arrayref* must be a reference to an array of single-precision floating point numbers. *index* must be an integer. The single-precision floating point number *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### daload

Load double float from array

Syntax:

| *daload = 49* |
|---|

Stack: ..., *arrayref*, *index* => ..., *value-word1*, *value-word2*

*arrayref* must be a reference to an array of double-precision floating point numbers. *index* must be an integer. The double-precision floating point number *value* at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### aaload

Load object reference from array

Syntax:

| *aaload = 50* |
|---|

Stack: ..., *arrayref*, *index* => ..., *value*

*arrayref* must be a reference to an array of references to objects. *index* must be an integer. The object reference at position number *index* in the array is retrieved and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### baload

Load signed byte from array.

Syntax:

| *baload = 51* |
|---|

Stack: ..., *arrayref*, *index* => ..., *value*

*arrayref* must be a reference to an array of signed bytes. *index* must be an integer. The signed byte value at position number *index* in the array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.



### caload

Load character from array

Syntax:

| *caload = 52* |
|---|

Stack: ..., *arrayref*, *index* => ..., *value*

*arrayref* must be a reference to an array of characters. *index* must be an integer. The character value at position number *index* in the array is retrieved, zero-extended to an integer, and pushed onto the top of the stack.

If *arrayref* is `null` a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### saload

Load short from array

Syntax:

| *saload = 53* |
|---|

Stack: ..., *arrayref*, *index* => ..., *value*

*arrayref* must be a reference to an array of short integers. *index* must be an integer. The ;signed short integer value at position number *index* in the array is retrieved, expanded to an integer, and pushed onto the top of the stack.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### iastore

Store into integer array

Syntax:

| *iastore = 79* |
|---|

Stack: ..., *arrayref*, *index*, *value* => ...

*arrayref* must be a reference to an array of integers, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

### lastore

Store into long integer array

Syntax:

| *lastore = 80* |
|---|

Stack: ..., *arrayref*, *index*, *value-word1*, *value-word2* => ...

*arrayref* must be a reference to an array of long integers, *index* must be an integer, and *value* a long integer. The long integer *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown.

**fastore**

Store into single float array

Syntax:

| *fastore = 81* |
|---|

Stack: ..., *arrayref*, *index*, *value* => ...

*arrayref* must be an array of single-precision floating point numbers, *index* must be an integer, and *value* a single-precision floating point number. The single float *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

**dastore**

Store into double float array

Syntax:

| *dastore = 82* |
|---|

Stack: ..., *arrayref*, *index*, *value-word1*, *value-word2* => ...

*arrayref* must be a reference to an array of double-precision floating point numbers, *index* must be an integer, and *value* a double-precision floating point number. The double float *value* is stored at position *index* in the array.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

**aastore**

Store into object reference array

Syntax:

| *aastore = 83* |
|---|

Stack: ..., *arrayref*, *index*, *value* => ...

*arrayref* must be a reference to an array of references to objects, *index* must be an integer, and *value* a reference to an object. The object reference *value* is stored at position *index* in the *array*.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown.

The actual type of *value* must be conformable with the actual type of the elements of the array. For example, it is legal to store an instance of class `Thread` in an array of class `Object`, but not vice versa. (See the *Java Language Specification* for information on how to determine whether a object reference is an instance of a class.) An `ArrayStoreException` is thrown if an attempt is made to store an incompatible object reference.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.



**bastore**

Store into signed byte array

Syntax:

| *bastore = 84* |
|---|

Stack: ..., *arrayref*, *index*, *value* => ...

*arrayref* must be a reference to an array of signed bytes, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array. If *value* is too large to be a signed byte, it is truncated.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

**castore**

Store into character array

Syntax:

| *castore = 85* |
|---|

Stack: ..., *arrayref*, *index*, *value* => ...

*arrayref* must be an array of characters, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array. If *value* is too large to be a character, it is truncated.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of [the array an `ArrayIndexOutOfBoundsException` is thrown.

**sastore**

Store into short array

Syntax:

| *sastore = 86* |
|---|

Stack: ..., *array*, *index*, *value* => ...

*arrayref* must be an array of shorts, *index* must be an integer, and *value* an integer. The integer *value* is stored at position *index* in the array. If *value* is too large to be an short, it is truncated.

If *arrayref* is `null`, a `NullPointerException` is thrown. If *index* is not within the bounds of the array an `ArrayIndexOutOfBoundsException` is thrown.

## 3.7 Stack Instructions

**nop**

Do nothing

Syntax:

| *nop = 0* |
|---|

Stack: no change

Do nothing.

31

### pop

Pop top stack word

Syntax:

| *pop = 87* |
|---|

Stack: ..., *any* => ...

Pop the top word from the stack.

### pop2

Pop top two stack words

Syntax:

| *pop2 = 88* |
|---|

Stack: ..., *any2*, *any1* => ...

Pop the top two words from the stack.

### dup

Duplicate top stack word

Syntax:

| *dup = 89* |
|---|

Stack: ..., *any* => ..., *any*, *any*

Duplicate the top word on the stack.

### dup2

Duplicate top two stack words

Syntax:

| *dup2 = 92* |
|---|

Stack: ..., *any2*, *any1* => ..., *any2*, *any1*, *any2*, *any1*

Duplicate the top two words on the stack.

### dup_x1

Duplicate top stack word and put two down

Syntax:

| *dup_x1 = 90* |
|---|

Stack: ..., *any2*, *any1* => ..., *any1*, *any2*, *any1*

Duplicate the top word on the stack and insert the copy two words down in the stack.



### dup2_x1

Duplicate top two stack words and put two down

Syntax:

| *dup2_x1 = 93* |
|---|

Stack: ..., *any3*, *any2*, *any1* => ..., *any2*, *any1*, *any3*, *any2*, *any1*

Duplicate the top two words on the stack and insert the copies two words down in the stack.

### dup_x2

Duplicate top stack word and put three down

Syntax:

| *dup_x2 = 91* |
|---|

Stack: ..., *any3*, *any2*, *any1* => ..., *any1*, *any3*, *any2*, *any1*

Duplicate the top word on the stack and insert the copy three words down in the stack.

### dup2_x2

Duplicate top two stack words and put three down

Syntax:

| *dup2_x2 = 94* |
|---|

Stack: ..., *any4*, *any3*, *any2*, *any1* => ..., *any2*, *any1*, *any4*, *any3*, *any2*, *any1*

Duplicate the top two words on the stack and insert the copies three words down in the stack.

### swap

Swap top two stack words

Syntax:

| *swap = 95* |
|---|

Stack: ..., *any2*, *any1* => ..., *any2*, *any1*

Swap the top two elements on the stack.

## 3.8 Arithmetic Instructions

### iadd

Integer add

Syntax:

| *iadd = 96* |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must be integers. The values are added and are replaced on the stack by their integer sum.

72

32

## ladd

Long integer add

Syntax:

| *ladd = 97* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be long integers. The values are added and are replaced on the stack by their long integer sum.

## fadd

Single floats add

Syntax:

| *fadd = 98* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be single-precision floating point numbers. The values are added and are replaced on the stack by their single-precision floating point sum.

## dadd

Double floats add

Syntax:

| *dadd = 99* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be double-precision floating point numbers. The values are added and are replaced on the stack by their double-precision floating point sum.

## isub

Integer subtract

Syntax:

| *isub = 100* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be integers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their integer difference.

## lsub

Long integer subtract

Syntax:

| *lsub = 101* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be long integers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their long integer difference.



## fsub

Single float subtract

Syntax:

| *fsub = 102* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be single-precision floating point numbers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their single-precision floating point difference

## dsub

Double float subtract

Syntax:

| *dsub = 103* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be double-precision floating point numbers. *value2* is subtracted from *value1*, and both values are replaced on the stack by their double-precision floating point difference.

## imul

Integer multiply

Syntax:

| *imul = 104* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be integers. Both values are replaced on the stack by their integer product

## lmul

Long integer multiply

Syntax:

| *lmul = 105* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be long integers. Both values are replaced on the stack by their long integer product.

## fmul

Single float multiply

Syntax:

| *fmul = 106* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be single-precision floating point numbers. Both values are replaced on the stack by their single-precision floating point product.

73

33

## dmul

Double float multiply

Syntax:

| *dmul = 107* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be double-precision floating point numbers. Both values are replaced on the stack by their double-precision floating point product.

## idiv

Integer divide

Syntax:

| *idiv = 108* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be integers. *value1* is divided by *value2*, and both values are replaced on the stack by their integer quotient.

The result is truncated to the nearest integer that is between it and 0. An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

## ldiv

Long integer divide

Syntax:

| *ldiv = 109* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be long integers. *value1* is divided by *value2*, and both values are replaced on the stack by their long integer quotient.

The result is truncated to the nearest integer that is between it and 0. An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

## fdiv

Single float divide

Syntax:

| *fdiv = 110* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must be single-precision floating point numbers. *value1* is divided by *value2*, and both values are replaced on the stack by their single-precision floating point quotient.

Divide by zero results in the quotient being NaN.



## ddiv

Double float divide

Syntax:

| *ddiv = 111* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must be double-precision floating point numbers. *value1* is divided by *value2*, and both values are replaced on the stack by their double-precision floating point quotient.

Divide by zero results in the quotient being NaN.

## irem

Integer remainder

Syntax:

| *irem = 112* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must both be integers. *value1* is divided by *value2*, and both values are replaced on the stack by their integer remainder.

An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown

Note: need a description of the integer remainder semantics

## lrem

Long integer remainder

Syntax:

| *lrem = 113* |
|---|

Stack: ..., *value1-word1, value1-word2, value2-word1, value2-word2* => ..., *result-word1, result-word2*

*value1* and *value2* must both be long integers. *value1* is divided by *value2*, and both values are replaced on the stack by their long integer remainder.

An attempt to divide by zero results in a "/ by zero" `ArithmeticException` being thrown.

Note: need a description of the integer remainder semantics

## frem

Single float remainder

Syntax:

| *frem = 114* |
|---|

Stack: ..., *value1, value2* => ..., *result*

*value1* and *value2* must both be single-precision floating point numbers. *value1* is divided by *value2*, and the quotient is truncated to an integer, and then multiplied by *value2*. The product is subtracted from *value1*. The result, as a single-precision floating point number, replaces both values on the stack *result* = *value1* - (integral_part(*value1* / *value2*) * *value2*), where integral_part() rounds to the nearest integer, with a tie going to the even number.

An attempt to divide by zero results in NaN.

Note: gls to provide a better definition of the floating remainder semantics

74

### drem

Double float remainder

Syntax:

| *drem* = 115 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

*value1* and *value2* must both be double-precision floating point numbers. *value1* is divided by *value2*, and the quotient is truncated to an integer, and then multiplied by *value2*. The product is subtracted from *value1*. The result, as a double-precision floating point number, replaces both values on the stack. *result* = *value1* - (integral_part(*value1* / *value2*) * *value2*), where integral_part() rounds to the nearest integer, with a tie going to the even number.

An attempt to divide by zero results in NaN.

Note: gls to provide a better definition of the floating remainder semantics

### ineg

Integer negate

Syntax:

| *ineg* = 116 |
|---|

Stack: ..., *value* => ..., *result*

*value* must be an integer. It is replaced on the stack by its arithmetic negation.

### lneg

Long integer negate

Syntax:

| *lneg* = 117 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

*value* must be a long integer. It is replaced on the stack by its arithmetic negation.

### fneg

Single float negate

Syntax:

| *fneg* = 118 |
|---|

Stack: ..., *value* => ..., *result*

*value* must be a single-precision floating point number. It is replaced on the stack by its arithmetic negation.

### dneg

Double float negate

Syntax:

| *dneg* = 119 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

*value* must be a double-precision floating point number. It is replaced on the stack by its arithmetic negation.



## 3.9 Logical Instructions

### ishl

Integer shift left

Syntax:

| *ishl* = 120 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must be integers. *value1* is shifted left by the amount indicated by the low five bits of *value2*. The integer result replaces both values on the stack.

### ishr

Integer arithmetic shift right

Syntax:

| *ishr* = 122 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must be integers. *value1* is shifted right arithmetically (with sign extension) by the amount indicated by the low five bits of *value2*. The integer result replaces both values on the stack

### iushr

Integer logical shift right

Syntax:

| *iushr* = 124 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must be integers. *value1* is shifted right logically (with no sign extension) by the amount indicated by the low five bits of *value2*. The integer result replaces both values on the stack.

### lshl

Long integer shift left

Syntax:

| *lshl* = 121 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2* => ..., *result-word1*, *result-word2*

*value1* must be a long integer and *value2* must be an integer. *value1* is shifted left by the amount indicated by the low six bits of *value2*. The long integer result replaces both values on the stack.

### lshr

Long integer arithmetic shift right

Syntax:

| *lshr* = 123 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2* => ..., *result-word1*, *result-word2*

*value1* must be a long integer and *value2* must be an integer. *value1* is shifted right arithmetically (with sign extension) by the amount indicated by the low six bits of *value2*. The long integer result replaces both values on the stack.

35

### lushr

Long integer logical shift right

Syntax:

| *lushr* = 125 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

*value1* must be a long integer and *value2* must be an integer. *value1* is shifted right logically (with no sign extension) by the amount indicated by the low six bits of *value2*. The long integer result replaces both values on the stack.

### iand

Integer boolean AND

Syntax:

| *iand* = 126 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must both be integers. They are replaced on the stack by their bitwise logical and (conjunction).

### land

Long integer boolean AND

Syntax:

| *land* = 127 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

*value1* and *value2* must both be long integers. They are replaced on the stack by their bitwise logical and (conjunction).

### ior

Integer boolean OR

Syntax:

| *ior* = 128 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must both be integers. They are replaced on the stack by their bitwise logical or (disjunction).

### lor

Long integer boolean OR

Syntax:

| *lor* = 129 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

*value1* and *value2* must both be long integers. They are replaced on the stack by their bitwise logical or (disjunction).



### ixor

Integer boolean XOR

Syntax:

| *ixor* = 130 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must both be integers. They are replaced on the stack by their bitwise exclusive or (exclusive disjunction).

### lxor

Long integer boolean XOR

Syntax:

| *lxor* = 131 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word2* => ..., *result-word1*, *result-word2*

*value1* and *value2* must both be long integers. They are replaced on the stack by their bitwise exclusive or (exclusive disjunction).

## 3.10 Conversion Operations

### i2l

Integer to long integer conversion

Syntax:

| *i2l* = 133 |
|---|

Stack: ..., *value* => ..., *result-word1*, *result-word2*

*value* must be an integer. It is converted to a long integer. The result replaces *value* on the stack.

### i2f

Integer to single float

Syntax:

| *i2f* = 134 |
|---|

Stack: ..., *value* => ..., *result*

*value* must be an integer. It is converted to a single-precision floating point number. The result replaces *value* on the stack.

### i2d

Integer to double float

Syntax:

| *i2d* = 135 |
|---|

Stack: ..., *value* => ..., *result-word1*, *result-word2*

*value* must be an integer. It is converted to a double-precision floating point number. The result replaces *value* on the stack.

76

36

### l2i

Long integer to integer

Syntax:

| *l2i* = 136 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result*

*value* must be a long integer. It is converted to an integer by taking the low-order 32 bits. The result replaces *value* on the stack.

### l2f

Long integer to single float

Syntax:

| *l2f* = 137 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result*

*value* must be a long integer. It is converted to a single-precision floating point number. The result replaces *value* on the stack.

### l2d

Long integer to double float

Syntax:

| *l2d* = 138 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

*value* must be a long integer. It is converted to a double-precision floating point number. The result replaces *value* on the stack.

### f2i

Single float to integer

Syntax:

| *f2i* = 139 |
|---|

Stack: ..., *value* => ..., *result*

*value* must be a single-precision floating point number. It is converted to an integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.



### f2l

Single float to long integer

Syntax:

| *f2l* = 140 |
|---|

Stack: ..., *value* => ..., *result-word1*, *result-word2*

*value* must be a single-precision floating point number. It is converted to a long integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

### f2d

Single float to double float

Syntax:

| *f2d* = 141 |
|---|

Stack: ..., *value* => ..., *result-word1*, *result-word2*

*value* must be a single-precision floating point number. It is converted to a double-precision floating point number. The result replaces *value* on the stack.

### d2i

Double float to integer

Syntax:

| *d2i* = 142 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result*

*value* must be a double-precision floating point number. It is converted to an integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

### d2l

Double float to long integer

Syntax:

| *d2l* = 143 |
|---|

Stack: ..., *value-word1*, *value-word2* => ..., *result-word1*, *result-word2*

*value* must be a double-precision floating point number. It is converted to a long integer. The result replaces *value* on the stack. See *The Java Language Specification* for details on converting floating point numbers to integers.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

77

## d2f

Double float to single float

Syntax:

Stack: ..., *value-word1*, *value-word2* => ..., *result*

*value* must be a double-precision floating point number. It is converted to a single-precision floating point number. If overflow occurs, the result must be infinity with the same sign as *value*. The result replaces *value* on the stack.

## int2byte

Integer to signed byte

Syntax:

Stack: ..., *value* => ..., *result*

*value* must be an integer. It is truncated to a signed 8-bit result, then sign extended to an integer. The result replaces *value* on the stack.

## int2char

Integer to char

Syntax:

Stack: ..., *value* => ..., *result*

*value* must be an integer. It is truncated to an unsigned 16-bit result, then zero extended to an integer. The result replaces *value* on the stack.

## int2short

Integer to short

Syntax:

Stack: ..., *value* => ..., *result*

*value* must be an integer. It is truncated to a signed 16-bit result, then sign extended to an integer. The result replaces *value* on the stack.

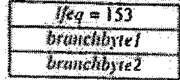


# 3.11 Control Transfer Instructions

## ifeq

Branch if equal to 0

Syntax:



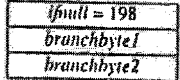

Stack: ..., *value* => ...

*value* must be an integer. It is popped from the stack. If *value* is zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifeq`.

## ifnull

Branch if null

Syntax:



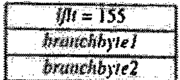

Stack: ..., *value* => ...

*value* must be a reference to an object. It is popped from the stack. If *value* is `null`, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifnull`

## iflt

Branch if less than 0

Syntax:




Stack: ..., *value* => ...

*value* must be an integer. It is popped from the stack. If *value* is less than zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `iflt`.

38

### ifle

Branch if less than or equal to 0

Syntax:

| *ifle* = 158 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value* => ...

*value* must be an integer. It is popped from the stack. If *value* is less than or equal to zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifle`.

### ifne

Branch if not equal to 0

Syntax:

| *ifne* = 154 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value* => ...

*value* must be an integer. It is popped from the stack. If *value* is not equal to zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifne`.

### ifnonnull

Branch if not null

Syntax:

| *ifnonnull* = 199 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value* => ...

*value* must be a reference to an object. It is popped from the stack. If *value* is not `null`, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifnonnull`.

### ifgt

Branch if greater than 0

Syntax:

| *ifgt* = 157 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value* => ...

*value* must be an integer. It is popped from the stack. If *value* is greater than zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifgt`.



### ifge

Branch if greater than or equal to 0

Syntax:

| *ifge* = 156 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value* => ...

*value* must be an integer. It is popped from the stack. If *value* is greater than or equal to zero, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `ifge`.

### if_icmpeq

Branch if integers equal

Syntax:

| *if_icmpeq* = 159 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be integers. They are both popped from the stack. If *value1* is equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpeq`.

### if_icmpne

Branch if integers not equal

Syntax:

| *if_icmpne* = 160 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be integers. They are both popped from the stack. If *value1* is not equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpne`.

### if_icmplt

Branch if integer less than

Syntax:

| *if_icmplt* = 161 |
| --- |
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be integers. They are both popped from the stack. If *value1* is less than *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that

51

offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmplt`.

## if_icmpgt

Branch if integer greater than

Syntax:

| *if_icmpgt* = 163 |
|---|
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be integers. They are both popped from the stack. If *value1* is greater than *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpgt`.

## if_icmple

Branch if integer less than or equal to

Syntax:

| *if_icmple* = 164 |
|---|
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be integers. They are both popped from the stack. If *value1* is less than or equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmple`.

## if_icmpge

Branch if integer greater than or equal to

Syntax:

| *if_icmpge* = 162 |
|---|
| *branchbyte1* |
| *branchbyte2* |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be integers. They are both popped from the stack. If *value1* is greater than or equal to *value2*, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_icmpge`.



## lcmp

Long integer compare

Syntax:

| *lcmp* = 148 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word1* => ..., *result*

*value1* and *value2* must be long integers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value -1 is pushed onto the stack.

## fcmpl

Single float compare (-1 on NaN)

Syntax:

| *fcmpl* = 149 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must be single-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value -1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value -1 is pushed onto the stack.

## fcmpg

Single float compare (1 on NaN)

Syntax:

| *fcmpg* = 150 |
|---|

Stack: ..., *value1*, *value2* => ..., *result*

*value1* and *value2* must be single-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value -1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value 1 is pushed onto the stack.

## dcmpl

Double float compare (-1 on NaN)

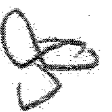
Syntax:

| *dcmpl* = 151 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word1* => ..., *result*

*value1* and *value2* must be double-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value -1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value -1 is pushed onto the stack.



## dcmpg

Double float compare (1 on NaN)

Syntax:

| dcmpg = 152 |
|---|

Stack: ..., *value1-word1*, *value1-word2*, *value2-word1*, *value2-word1* => ..., *result*

*value1* and *value2* must be double-precision floating point numbers. They are both popped from the stack and compared. If *value1* is greater than *value2*, the integer value 1 is pushed onto the stack. If *value1* is equal to *value2*, the value 0 is pushed onto the stack. If *value1* is less than *value2*, the value -1 is pushed onto the stack.

If either *value1* or *value2* is NaN, the value 1 is pushed onto the stack.

## if_acmpeq

Branch if object references are equal

Syntax:

| if_acmpeq = 165 |
|---|
| branchbyte1 |
| branchbyte2 |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be references to objects. They are both popped from the stack. If the objects referenced are not the same, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the Address of this instruction. Otherwise execution proceeds at the instruction following the `if_acmpeq`.

## if_acmpne

Branch if object references not equal

Syntax:

| if_acmpne = 166 |
|---|
| branchbyte1 |
| branchbyte2 |

Stack: ..., *value1*, *value2* => ...

*value1* and *value2* must be references to objects. They are both popped from the stack. If the objecs referenced are not the same, *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction. Otherwise execution proceeds at the instruction following the `if_acmpne`.

## goto

Branch always

Syntax:

| goto = 167 |
|---|
| branchbyte1 |
| branchbyte2 |

Stack: no change

*branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. Execution proceeds at that offset from the address of this instruction.



## goto_w

Branch always (wide index)

Syntax:

| goto_w = 200 |
|---|
| branchbyte1 |
| branchbyte2 |
| branchbyte3 |
| branchbyte4 |

Stack: no change

*branchbyte1*, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit offset. Execution proceeds at that offset from the address of this instruction.

## jsr

Jump subroutine

Syntax:

| jsr = 168 |
|---|
| branchbyte1 |
| branchbyte2 |

Stack: ... => ..., *return-address*

*branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset. The address of the instruction immediately following the `jsr` is pushed onto the stack. Execution proceeds at the offset from the address of this instruction.

Note: The `jsr` instruction is used in the implementation of Java's `finally` keyword.

## jsr_w

Jump subroutine (wide index)

Syntax:

| jsr_w = 201 |
|---|
| branchbyte1 |
| branchbyte2 |
| branchbyte3 |
| branchbyte4 |

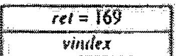
Stack: ... => ..., *return-address*

*branchbyte1*, *branchbyte2*, *branchbyte3*, and *branchbyte4* are used to construct a signed 32-bit offset. The address of the instruction immediately following the `jsr_w` is pushed onto the stack. Execution proceeds at the offset from the address of this instruction.

### ret

Return from subroutine

Syntax:




Stack: no change

Local variable *vindex* in the current Java frame must contain a return address. The contents of the local variable are written into the pc.

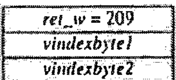
Note that `jsr` pushes the address onto the stack, and `ret` gets it out of a local variable. This asymmetry is intentional.

Note: The `ret` instruction is used in the implementation of Java's `finally` keyword.

### ret_w

Return from subroutine (wide index)

Syntax:




Stack: no change

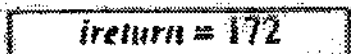
*vindexbyte1* and *vindexbyte2* are assembled into an unsigned 16-bit index to a local variable in the current Java frame. That local variable must contain a return address. The contents of the local variable are written into the pc. See the `ret` instruction for more information.

## 3.12 Function Return

### ireturn

Return integer from function

Syntax:



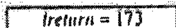

Stack: ..., *value* => [empty]

*value* must be an integer. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

### lreturn

Return long integer from function

Syntax:




Stack: ..., *value-word1*, *value-word2* => [empty]

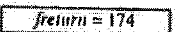
*value* must be a long integer. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.



### freturn

Return single float from function

Syntax:



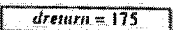

Stack: ..., *value* => [empty]

*value* must be a single-precision floating point number. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

### dreturn

Return double float from function

Syntax:



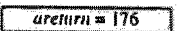

Stack: ..., *value-word1*, *value-word2* => [empty]

*value* must be a double-precision floating point number. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

### areturn

Return object reference from function

Syntax:



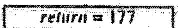

Stack: ..., *value* => [empty]

*value* must be a reference to an object. The value *value* is pushed onto the stack of the previous execution environment. Any other values on the operand stack are discarded. The interpreter then returns control to its caller.

### return

Return (void) from procedure

Syntax:



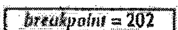

Stack: ... => [empty]

All values on the operand stack are discarded. The interpreter then returns control to its caller.

### breakpoint

Stop and pass control to breakpoint handler

Syntax:




Stack: no change

## 3.13 Table Jumping

### tableswitch

Access jump table by index and jump

Syntax:

| *tableswitch = 170* |
|---|
| *...0-3 byte pad...* |
| *default-offset1* |
| *default-offset2* |
| *default-offset3* |
| *default-offset4* |
| *low1* |
| *low2* |
| *low3* |
| *low4* |
| *high1* |
| *high2* |
| *high3* |
| *high4* |
| *...jump offsets...* |

Stack: ..., *index* => ...

`tableswitch` is a variable length instruction. Immediately after the `tableswitch` instruction, between zero and three 0's are inserted as padding so that the next byte begins at an address that is a multiple of four. After the padding follow a series of signed 4-byte quantities: *default-offset*, *low*, *high*, and then *high-low*+1 further signed 4-byte offsets. The *high-low*+1 signed 4-byte offsets are treated as a 0-based jump table.

The *index* must be an integer. If *index* is less than *low* or *index* is greater than *high*, then *default-offset* is added to the address of this instruction. Otherwise, *low* is subtracted from *index*, and the *index-low*'th element of the jump table is extracted, and added to the address of this instruction.



### lookupswitch

Access jump table by key match and jump

Syntax:

| *lookupswitch = 171* |
|---|
| *...0-3 byte pad...* |
| *default-offset1* |
| *default-offset2* |
| *default-offset3* |
| *default-offset4* |
| *npairs1* |
| *npairs2* |
| *npairs3* |
| *npairs4* |
| *..match-offset pairs..* |

Stack: ..., *key* => ...

`lookupswitch` is a variable length instruction. Immediately after the `lookupswitch` instruction, between zero and three 0's are inserted as padding so that the next byte begins at an address that is a multiple of four.

Immediately after the padding are a series of pairs of signed 4-byte quantities. The first pair is special. The first item of that pair is the default offset, and the second item of that pair gives the number of pairs that follow. Each subsequent pair consists of a *match* and an *offset*.

The *key* must be an integer. The integer *key* on the stack is compared against each of the *matches*. If it is equal to one of them, the *offset* is added to the address of this instruction. If the *key* does not match any of the *matches*, the default offset is added to the address of this instruction.

## 3.14 Manipulating Object Fields

### putfield

Set field in object

Syntax:

| *putfield = 181* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref*, *value* => ...

OR

Stack: ..., *objectref*, *value-word1*, *value-word2* => ...

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer which has both the field width (in bytes) and the field offset (in bytes).

The field at that *offset* from the start of the object referenced by *objectref* will be set to the *value* on the top of the stack.

This instruction deals with both 32-bit and 64-bit wide fields.

If *objectref* is `null`, a `NullPointerException` is generated.

If the specified field is a static field, an `IncompatibleClassChangeError` is thrown.

### getfield

Fetch field from object

Syntax:

| *getfield* = 180 |
| --- |
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref* => ..., *value*

OR

Stack: ..., *objectref* => ..., *value-word1*, *value-word2*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a class name and a field name. The item is resolved to a field block pointer which has both the field width (in bytes) and the field *offset* (in bytes).

*objectref* must be a reference to an object. The value at *offset* into the object referenced by *objectref* replaces *objectref* on the top of the stack.

This instruction deals with both 32-bit and 64-bit wide fields.

If *objectref* is `null`, a `NullPointerException` is generated.

If the specified field is a static field, an `IncompatibleClassChangeError` is thrown.

### putstatic

Set static field in class

Syntax:

| *putstatic* = 179 |
| --- |
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *value* => ...

OR

Stack: ..., *value-word1*, *value-word2* => ...

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field will be set to have the value on the top of the stack.

This instruction works for both 32-bit and 64-bit wide fields.

If the specified field is a dynamic field, an `IncompatibleClassChangeError` is thrown.



### getstatic

Get static field from class

Syntax:

| *getstatic* = 178 |
| --- |
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., => ..., *value*

OR

Stack: ..., => ..., *value-word1*, *value-word2*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class.

This instruction deals with both 32-bit and 64-bit wide fields.

If the specified field is a dynamic field, an `IncompatibleClassChangeError` is generated.

## 3.15 Method Invocation

There are four instructions that implement method invocation.

| | |
| --- | --- |
| `invokevirtual` | Invoke an instance method of an object, dispatching based on the runtime (virtual) type of the object. This is the normal method dispatch in Java. |
| `invokenonvirtual` | Invoke an instance method of an object, dispatching based on the compile-time (non-virtual) type of the object. This is used, for example, when the keyword `super` or the name of a superclass is used as a method qualifier. |
| `invokestatic` | Invoke a class (`static`) method in a named class. |
| `invokeinterface` | Invoke a method which is implemented by an interface, searching the methods implemented by the particular run-time object to find the appropriate method. |

### invokevirtual

Invoke instance method, dispatch based on run-time type

Syntax:

| *invokevirtual* = 182 |
| --- |
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain a reference to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object reference. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is an index into the method table of the named class, which is used with the object's dynamic type to look in the method table of that type, where a pointer to the method block for

the matched method is found. The method block indicates the type of method (`native`, `synchronized`, and so on) and the number of arguments expected on the operand stack.

If the method is marked `synchronized` the monitor associated with *objectref* is entered.

The *objectref* and arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If the object reference on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

### invokenonvirtual

Invoke instance method, dispatching based on compile-time type

Syntax:

| *invokenonvirtual* = 183 |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain a reference to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a complete method signature and class. The method signature is looked up in the method table of the class indicated. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, and so on) and the number of arguments (nargs) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with *objectref* is entered.

The *objectref* and arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If the object reference on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

### invokestatic

Invoke a class (static) method

Syntax:

| *invokestatic* = 184 |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature and class. The method signature is looked up in the method table of the class indicated. The method signature is guaranteed to exactly match one of the method signatures in the class's method table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, and so on) and the number of arguments (nargs) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with the class is entered.

The arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.



### invokeinterface

Invoke interface method

Syntax:

| *invokeinterface* = 185 |
|---|
| *indexbyte1* |
| *indexbyte2* |
| *nargs* |
| *reserved* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]], ... => ...

The operand stack must contain a reference to an object and *nargs*-1 arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object reference. The method signature is looked up in the method table. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, and so on) but unlike `invokevirtual` and `invokenonvirtual`, the number of available arguments (nargs) is taken from the bytecode.

If the method is marked `synchronized` the monitor associated with *objectref* is entered.

The *objectref* and arguments are popped off this method's stack and become the initial values of the local variables of the new method. Execution continues with the first instruction of the new method.

If the *objectref* on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

## 3.16 Exception Handling

### athrow

Throw exception or error

Syntax:

| *athrow* = 191 |
|---|

Stack: ..., *objectref* => [undefined]

*objectref* must be a reference to an object which is a subclass of `Throwable`, which is thrown. The current Java stack frame is searched for the most recent catch clause that catches this class or a superclass of this class. If a matching catch list entry is found, the pc is reset to the address indicated by the catch-list entry, and execution continues there.

If no appropriate catch clause is found in the current stack frame, that frame is popped and the object is rethrown. If one is found, it contains the location of the code for this exception. The pc is reset to that location and execution continues. If no appropriate catch is found in the current stack frame, that frame is popped and the *objectref* is rethrown.

If *objectref* is `null`, then a `NullPointerException` is thrown instead.

## 3.17 Miscellaneous Object Operations

### new

Create new object

Syntax:

| *new* = 187 |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ... => ..., *objectref*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index must be a class name that can be resolved to a class pointer, *class*. A new instance of that class is then created and a reference to the object is pushed on the stack.

### checkcast

Make sure object is of given type

Syntax:

| *checkcast* = 192 |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref* => ..., *objectref*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name which can be resolved to a class pointer, *class*. *objectref* must be a reference to an object.

`checkcast` determines whether *objectref* can be cast to be a reference to an object of class *class*. A `null` *objectref* can be cast to any *class*. Otherwise the referenced object must be an instance of *class* or one of its superclasses. (See the *Java Language Specification* for information on how to determine whether a *objectref* is an instance of a class.) If *objectref* can be cast to *class* execution proceeds at the next instruction, and the *objectref* remains on the stack.

If *objectref* cannot be cast to *class*, a `ClassCastException` is thrown.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.

### instanceof

Determine if an object is of given type

Syntax:

| *instanceof* = 193 |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref* => ..., *result*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The string at that index of the constant pool is presumed to be a class name which can be resolved to a class pointer, *class*. *objectref* must be a reference to an object.

*instanceof* determines whether *objectref* can be cast to be a reference to an object of the class *class*. This instruction will overwrite *objectref* with 1 if *objectref* is an instance of *class* or one of its superclasses. (See the *Java Language Specification* for information on how to determine whether a object reference is an instance of a class.) Otherwise, *objectref* is overwritten by 0. If *objectref* is `null`, it's overwritten by 0.

Note: Mustn't refer to the *Java Language Specification*; give semantics here.



## 3.18 Monitors

### monitorenter

Enter monitored region of code

Syntax:

| *monitorenter* = 194 |
|---|

Stack: ..., *objectref* => ...

*objectref* must be a reference to an object.

The interpreter attempts to obtain exclusive access via a lock mechanism to *objectref*. If another thread already has *objectref* locked, than the current thread waits until the object is unlocked. If the current thread already has the object locked, then continue execution. If the object is not locked, then obtain an exclusive lock.

If *objectref* is `null`, then a `NullPointerException` is thrown instead.

### monitorexit

Exit monitored region of code

Syntax:

| *monitorexit* = 195 |
|---|

Stack: ..., *objectref* => ...

*objectref* must be a reference to an object.

The lock on the object released. If this is the last lock that this thread has on that object (one thread is allowed to have multiple locks on a single object), then other threads that are waiting for the object to be available are allowed to proceed.

If *objectref* is `null`, then a `NullPointerException` is thrown instead.

# Appendix A: An Optimization

The following set of pseudo-instructions suffixed by `_quick` are variants of Java virtual machine instructions. They are used to improve the speed of interpreting bytecodes. They are not part of the virtual machine specification or instruction set, and are invisible outside of an Java virtual machine implementation. However, inside a virtual machine implementation they have proven to be an effective optimization.

A compiler from Java source code to the Java virtual machine instruction set emits only non-`_quick` instructions. If the `_quick` pseudo-instructions are used, each instance of a non-`_quick` instruction with a `_quick` variant is overwritten on execution by its `_quick` variant. Subsequent execution of that instruction instance will be of the `_quick` variant.

In all cases, if an instruction has an alternative version with the suffix `_quick`, the instruction references the constant pool. If the `_quick` optimization is used, each non-`_quick` instruction with a `_quick` variant performs the following:

- Resolves the specified item in the constant pool
- Signals an error if the item in the constant pool could not be resolved for some reason
- Turns itself into the `_quick` version of the instruction. The instructions `putstatic`, `getstatic`, `putfield`, and `getfield` each have two `_quick` versions.
- Performs its intended operation

This is identical to the action of the instruction without the `_quick` optimization, except for the additional step in which the instruction overwrites itself with its `_quick` variant.

The `_quick` variant of an instruction assumes that the item in the constant pool has already been resolved, and that this resolution did not generate any errors. It simply performs the intended operation on the resolved item.

Note: some of the invoke methods only support a single-byte offset into the method table of the object; for objects with 256 or more methods some invocations cannot be "quicked" with only these bytecodes. We also need to define or change existing `getfield` and `putfield` bytecodes to support more than a byte of *offset*.

This Appendix doesn't give the opcode values of the pseudo-instructions, since they are invisible and subject to change.

## A.1 Constant Pool Resolution

When the class is read in, an array `constant_pool[]` of size `nconstants` is created and assigned to a field in the class. `constant_pool[0]` is set to point to a dynamically allocated array which indicates which fields in the constant_pool have already been resolved. `constant_pool[1]` through `constant_pool[nconstants - 1]` are set to point at the "type" field that corresponds to this constant item.

When an instruction is executed that references the constant pool, an index is generated, and `constant_pool[0]` is checked to see if the index has already been resolved. If so, the value of `constant_pool[index]` is returned. If not, the value of `constant_pool[index]` is resolved to be the actual pointer or data, and overwrites whatever value was already in `constant_pool[index]`.

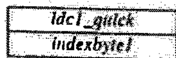


## A.2 Pushing Constants onto the Stack (_quick variants)

### ldc1_quick

Push item from constant pool onto stack

Syntax:



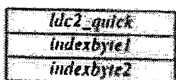

Stack: ... => ..., *item*

*indexbyte1* is used as an unsigned 8-bit index into the constant pool of the current class. The *item* at that index is pushed onto the stack.

### ldc2_quick

Push item from constant pool onto stack

Syntax:



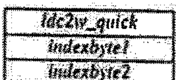

Stack: ... => ..., *item*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The *constant* at that index is resolved and the *item* at that index is pushed onto the stack.

### ldc2w_quick

Push long integer or double float from constant pool onto stack

Syntax:




Stack: ... => ..., *constant-word1*, *constant-word2*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The *constant* at that index is pushed onto the stack.

## A.3 Managing Arrays (_quick variants)

### anewarray_quick

Allocate new array of references to objects

Syntax:

| *anewarray_quick* |
| --- |
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *size* => *result*

*size* must be an integer. It represents the number of elements in the new array.

*indexbyte1* and *indexbyte2* are are used to construct an index into the constant pool of the current class. The entry must be a class.

A new array of the indicated class type and capable of holding *size* elements is allocated, and *result* is a reference to this new array. Allocation of an array large enough to contain *size* items of the given class type is attempted. All elements of the array are initialized to zero.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

### multianewarray_quick

Allocate new multi-dimensional array

Syntax:

| *multianewarray_quick* |
| --- |
| *indexbyte1* |
| *indexbyte2* |
| *dimensions* |

Stack: ..., *size1*, *size2*, ...*sizen* => *result*

Each *size* must be an integer. Each represents the number of elements in a dimension of the array.

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The resulting entry must be a class.

*dimensions* has the following aspects:

- It must be an integer ≥ 1.
- It represents the number of dimensions being created. It must be ≤ the number of dimensions of the array class.
- It represents the number of elements that are popped off the stack. All must be integers greater than or equal to zero. These are used as the sizes of the dimension.

If any of the *size* arguments on the stack is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryError` is thrown.

The *result* is a reference to the new array object.

Note: More explanation needed about how this is an array of arrays.



## A.4 Manipulating Object Fields (_quick variants)

### putfield_quick

Set field in object

Syntax:

| *putfield_quick* |
| --- |
| *offset* |
| *unused* |

Stack: ..., *objectref*, *value* => ...

*objectref* must be a reference to an object. *value* must be a value of a type appropriate for the specified field. *offset* is the offset for the field in that object. *value* is written at *offset* into the object. Both *objectref* and *value* are popped from the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

### putfield2_quick

Set long integer or double float field in object

Syntax:

| *putfield2_quick* |
| --- |
| *offset* |
| *unused* |

Stack: ..., *objectref*, *value-word1*, *value-word2*=> ...

*objectref* must be a reference to an object. *value* must be a value of a type appropriate for the specified field. *offset* is the offset for the field in that object. *value* is written at *offset* into the object . Both *objectref* and *value* are popped from the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

### getfield_quick

Fetch field from object

Syntax:

| *getfield_quick* |
| --- |
| *offset* |
| *unused* |

Stack: ..., *objectref* => ..., *value*

*objectref* must be a handle to an object. The value at *offset* into the object referenced by *objectref* replaces *objectref* on the top of the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

### getfield2_quick

Fetch field from object

Syntax:

| *getfield2_quick* |
|---|
| *offset* |
| *unused* |

Stack: ..., *objectref* => ..., *value-word1*, *value-word2*

*objectref* must be a handle to an object. The value at *offset* into the object referenced by *objectref* replaces *objectref* on the top of the stack.

If *objectref* is `null`, a `NullPointerException` is generated.

### putstatic_quick

Set static field in class

Syntax:

| *putstatic_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *value* => ...

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. *value* must be the type appropriate to that field. That field will be set to have the value *value*.

### putstatic2_quick

Set static field in class

Syntax:

| *putstatic2_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *value-word1*, *value-word2* => ...

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field must either be a long integer or a double precision floating point number. *value* must be the type appropriate to that field. That field will be set to have the value *value*.

### getstatic_quick

Get static field from class

Syntax:

| *getstatic_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., => ..., *value*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The value of that field will replace *handle* on the stack.



### getstatic2_quick

Get static field from class

Syntax:

| *getstatic2_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., => ..., *value-word1*, *value-word2*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The field must be a long integer or a double precision floating point number. The value of that field will replace *handle* on the stack

## A.5 Method Invocation (_quick variants)

### invokevirtual_quick

Invoke instance method, dispatching based on run-time type

Syntax:

| *invokevirtual_quick* |
|---|
| *offset* |
| *nargs* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object and *nargs-1* arguments. The method block at *offset* in the object's method table, as determined by the object's dynamic type, is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.).

If the method is marked `synchronized` the monitor associated with the object is entered.

The base of the local variables array for the new Java stack frame is set to point to *objectref* on the stack, making *objectref* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

### invokevirtualobject_quick

Invoke instance method of class `java.lang.Object`, specifically for benefit of arrays

Syntax:

| *invokevirtualobject_quick* |
|---|
| *offset* |
| *nargs* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object or to an array and *nargs-1* arguments. The method block at *offset* in `java.lang.Object`'s method table is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.).

If the method is marked `synchronized` the monitor associated with *handle* is entered.

The base of the local variables array for the new Java stack frame is set to point to *objectref* on the stack, making *objectref* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new

frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

### invokenonvirtual_quick

Invoke instance method, dispatching based on compile-time type

Syntax:

| *invokenonvirtual_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with the object is entered.

The base of the local variables array for the new Java stack frame is set to point to *objectref* on the stack, making *objectref* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

### invokestatic_quick

Invoke a class (static) method

Syntax:

| *invokestatic_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., [*arg1*, [*arg2* ...]] => ...

The operand stack must contain some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (`native`, `synchronized`, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked `synchronized` the monitor associated with the method's class is entered.

The base of the local variables array for the new Java stack frame is set to point to the first argument on the stack, making the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the



operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

### invokeinterface_quick

Invoke interface method

Syntax:

| *invokeinterface_quick* |
|---|
| *idbyte1* |
| *idbyte2* |
| *nargs* |
| *guess* |

Stack: ..., *objectref*, [*arg1*, [*arg2* ...]] => ...

The operand stack must contain *objectref*, a reference to an object, and *nargs*-1 arguments. *idbyte1* and *idbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object *handle*.

The method signature is searched for in the object's method table. As a short-cut, the method signature at slot *guess* is searched first. If that fails, a complete search of the method table is performed. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (`native`, `synchronized`, etc.) but the number of available arguments (*nargs*) is taken from the bytecode

If the method is marked `synchronized` the monitor associated with *handle* is entered.

The base of the local variables array for the new Java stack frame is set to point to *handle* on the stack, making *handle* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If *objectref* is `null`, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowError` is thrown.

*guess* is the last guess. Each time through, guess is set to the method offset that was used.

## A.6 Miscellaneous Object Operations (_quick variants)

### new_quick

Create new object

Syntax:

| *new_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ... => ..., *objectref*

*indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index must be a class. A new instance of that class is then created and *objectref*, a reference to that object is pushed on the stack.

## checkcast_quick

Make sure object is of given type

Syntax:

| *checkcast_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref* => ..., *objectref*

*objectref* must be a reference to an object. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The object at that index of the constant pool must have already been resolved.

`checkcast` then determines whether *objectref* can be cast to a reference to an object of class *class*. A `null` reference can be cast to any *class*, and otherwise the superclasses of *objectref*'s type are searched for *class*. If class is determined to be a superclass of *objectref*'s type, or if *objectref* is null, it can be cast to *objectref* cannot be cast to *class*, a `ClassCastException` is thrown.

Note: here (and probably in other places) we assume casts don't change the reference; this is implementation dependent.

## instanceof_quick

Determine if object is of given type

Syntax:

| *instanceof_quick* |
|---|
| *indexbyte1* |
| *indexbyte2* |

Stack: ..., *objectref* => ..., *result*

*objectref* must be a reference to an object. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item of class *class* at that index of the constant pool must have already been resolved.

`instanceof` determines whether *objectref* can be cast to an object of the class *class*. A `null` *objectref* can be cast to any *class*, and otherwise the superclasses of *objectref*'s type are searched for *class*. If *class* is determined to be a superclass of *objectref*'s type, *result* is 1 (`true`). Otherwise, *result* is 0 (`false`). If *handle* is `null`, *result* is 0 (`false`).

# Index of Instructions


aaload 39
aastore 41
aconst_null 29
aload 32
aload_<n> 32
anewarray 37
anewarray_quick 75
areturn 64
arraylength 38
astore 35
astore_<n> 35
athrow 70
baload 39
bastore 42
bipush 27
breakpoint 64
caload 40
castore 42
checkcast 71
checkcast_quick 81
d2f 55
d2i 54
d2l 54
dadd 45
daload 39
dastore 41
dcmpg 61
dcmpl 60
dconst_<d> 30
ddiv 48
dload 32
dload_<n> 32
dmul 47
dneg 49
drem 49
dreturn 64
dstore 34
dstore_<n> 34
dsub 46
dup 43
dup_x1 43
dup_x2 44
dup2 43
dup2_x1 44
dup2_x2 44
f2d 54
f2i 53
f2l 54
fadd 45
faload 39
fastore 41
fcmpg 60
fcmpl 60
fconst_<f> 30
fdiv 47
fload 31
fload_<n> 31
fmul 46
fneg 49
frem 48
freturn 64
fstore 34
fstore_<n> 34
fsub 46
getfield 67
getfield_quick 76
getfield2_quick 77
getstatic 68
getstatic_quick 77
getstatic2_quick 78
goto 61
goto_w 62
i2d 52
i2f 52
i2l 52
iadd 44
iaload 38
iand 51
iastore 40
iconst_<n> 29
iconst_m1 29
idiv 47
if_acmpeq 61
if_acmpne 61
if_icmpeq 58
if_icmpge 59
if_icmpgt 59
if_icmple 59
if_icmplt 58
if_icmpne 58
ifeq 56
ifge 58
ifgt 57
ifle 57
iflt 56
ifne 57
ifnonnull 57
ifnull 56
iinc 35
iload 30
iload_<n> 30
imul 46
ineg 49
instanceof 71
instanceof_quick 81
instruction name 27
int2byte 55
int2char 55
int2short 55
invokeinterface 70
invokeinterface_quick 80
invokenonvirtual 69
invokenonvirtual_quick 79
invokestatic 69
invokestatic_quick 79
invokevirtual 68
invokevirtual_quick 78
invokevirtualobject_quick 78
ior 51
irem 48
ireturn 63
ishl 50
ishr 50
istore 33
istore_<n> 33
isub 45
iushr 50
ixor 52
jsr 62
jsr_w 62
l2d 53
l2f 53
l2i 53
ladd 45
laload 38
land 51
lastore 40
lcmp 60
lconst_<l> 29
ldc1 28
ldc1_quick 74
ldc2 28
ldc2_quick 74
ldc2w 29
ldc2w_quick 74
ldiv 47
lload 31
lload_<n> 31
lmul 46
lneg 49
lookupswitch 66
lor 51
lrem 48
lreturn 63
lshl 50
lshr 50
lstore 33
lstore_<n> 33
lsub 45
lushr 51
lxor 52
monitorenter 72
monitorexit 72
multianewarray 37
multianewarray_quick 75
new 71
new_quick 80
newarray 36
nop 42
pop 43
pop2 43
putfield 66
putfield_quick 76
putfield2_quick 76
putstatic 67
putstatic_quick 77
putstatic2_quick 77
ret 63
ret_w 63
return 64
saload 40
sastore 42
sipush 28
swap 44
tableswitch 65
wide 35

P3

# Efficient JavaVM Just-in-Time Compilation

Andreas Krall
http://www.complang.tuwien.ac.at/andi/

## Abstract

*Conventional compilers are designed for producing highly optimized code without paying much attention to compile time. The design goals of Java just-in-time compilers are different: produce fast code at the smallest possible compile time. In this article we present a very fast algorithm for translating JavaVM byte code to high quality machine code for RISC processors. This algorithm handles combines instructions, does copy elimination and coalescing and does register allocation. It comprises three passes: basic block determination, stack analysis and register preallocation, final register allocation and machine code generation. This algorithm replaces an older one in the CACAO JavaVM implementation reducing the compile time by a factor of seven and producing slightly faster machine code. The speedup comes mainly from following simplifications: fixed assignment of registers at basic block boundaries, simple register allocator, better exception handling, better memory management and fine tuning the implementation. The CACAO system is currently faster than every JavaVM implementation for the Alpha processor and generates machine code for all used methods of the javac compiler and its libraries in 60 milliseconds on an Alpha workstation.*

## 1 Introduction

Java's [2] success as a programming language results from its role as an Internet programming language. The basis for this success is the machine-independent distribution format of programs with the Java virtual machine [12]. The standard interpretive implementation of the Java virtual machine makes execution of programs slow. This does not matter if small applications are executed in a browser, but becomes intolerable if big applications are executed. There are two solutions to solve this problem:

- specialized JavaVM processors,
- compilation of byte code to the native code of a standard processor.

SUN took both paths and is developing both Java processors and native code compilers. In our CACAO system we chose to go for native code compilation since it is more portable and gives more opportunities for improving the execution speed. Compiling to native code can be done in two different ways: compilation of the complete program in advance or compilation on demand of only the methods which are executed (just in time compiler, JIT). The CACAO system [10] uses a JIT compiler and is freely available via the world wide web.

### 1.1 Previous Work

The idea of machine independent program representations is quite old and goes back to the year 1960 [14]. An intermediate language UNCOL (UNiversal Computer Oriented Language) was proposed for use in compilers to reduce the development effort of compiling many different languages to many different architectures. The design of the JavaVM has been strongly influenced by P code, the abstract machine used by many Pascal implementations [13]. P code is well known from its use in the UCSD Pascal system. There have even been efforts to develop microprocessors which execute P code directly.

The Amsterdam compiler kit [16] [15] uses a stack oriented intermediate language. This language has been designed for fast compilers which emit efficient code. The intermediate representation of the Gardens Point compiler project is also based on a stack machine called *Dcode* [8]. *Dcode* was influenced by Pascal P code. Both *Dcode* interpreters and code generators for different architectures exist.

The problems of compiling a stack oriented abstract machine code to native code are well known from the programming language Forth. In his thesis [5] and in [7] Ertl describes RAFTS, a Forth system that generates native code

[1]Copyright 1998 IEEE. Published in the Proceedings of PACT'98, 12-18 October 1998 in Paris, France. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

at run time. Translating the stack operations to native code is done by translating the operations back to expressions represented as directed acyclic graphs as an intermediate step. In [6] he translates Forth to native code using C as an intermediate language. In this system the stack slots are translated to local variables of a function. Optimization and code generation are performed by the C compiler.

The first implementations of JIT compilers became available last year for the browsers from Netscape and Microsoft on PCs. They were followed by Symantec's development environment. Recently SUN released a JIT compiler for the Sparc and PowerPC processors. Silicon Graphics developed a JIT compiler for the MIPS processor and recently Digital released a JIT for the Alpha processor.

A public domain JIT compiler for several architectures is the kaffe system developed by Tim Wilkinson (`http://www.kaffe.org/`). For all the above mentioned systems, no publicly available description of the compilation techniques exists.

The translation scheme of the *Caffeine* system is described in [9]. It supports both a simple translation scheme which emulates the stack architecture and a more sophisticated one which eliminates the stack completely and uses registers instead. *Caffeine* is not intended as a JIT compiler. It compiles a complete program in advance. DAISY (Dynamically Architected Instruction Set from Yorktown) is a VLIW architecture developed at IBM for fast execution of PowerPC, S/390 and JavaVM code. Compatibility with different old architectures is achieved by using a JIT compilation technique. The JIT compilation scheme for the JavaVM is described in [4].

Adl-Tabatabai and others [1] describe a fast and effective code generation system for a JIT compiler. This compiler does optimizations like bound check elimination, common subexpression elimination and two kinds of register allocation, a simple one and a global priority based one. The results show that for most benchmark programs the complex register allocator and the subexpression eliminator incur to much overhead which does not pay back at run time.

## 2 Translation of stack code to register code

The JavaVM is a typed stack architecture [12]. There are different instructions for integer, long integer, floating point and address types. The main instruction set consists of arithmetic/logical and load/store/constant instructions. All these instructions either work directly on the stack or move values between the stack and local variables. There are special instructions for array access and for accessing the fields of objects (memory access), for method invocation, and for type checking.

The architecture of a RISC processor is completely different from the stack architecture of the JavaVM. RISC processors have large sets of registers. They execute arithmetic and logic operations only on values which are held in registers. Load and store instructions are provided to move data between memory and registers. Local variables of methods usually reside in registers and are saved in memory only during a method call or if there are too few registers.

### 2.1 Machine code translation examples

The example expression `a = b - c * d` would be translated by an optimizing C compiler to the following two Alpha instructions (the variables `a`, `b`, `c` and `d` reside in registers):

```
MULL c,d,tmp0    ; tmp0 = c * d
SUBL b,tmp0,a    ; a = b - tmp0
```

If JavaVM code is translated to machine code, the stack is eliminated and the stack slots are represented by temporary variables usually residing in registers. A naive translation of the previous example would result in the following Alpha instructions:

```
iload b    -->  MOVE b,t0
iload c    -->  MOVE c,t1
iload d    -->  MOVE d,t2
imul       -->  MULL t1,t2,t1
isub       -->  SUBL t0,t1,t0
istore a   -->  MOVE t0,a
```

The problems of translating JavaVM code to machine code are primarily the elimination of the unnecessary copy instructions and finding an efficient register allocation algorithm. A common but expensive technique is to do the naive translation and use an additional pass for copy elimination and coalescing.

### 2.2 The old translation scheme

The old CACAO compiler did the translation to machine code in four steps. First, basic blocks were determined. Then, the JavaVM was translated into a register oriented intermediate representation, the registers were allocated, and finally machine code was generated. The intermediate representation was oriented towards a RISC architecture target and assumed that all operands reside in registers (assuming an unlimited number of pseudo registers). The intermediate instructions contained a `MOVE` instruction for register moves, `OP1`, `OP2` and `OP3` instructions for the arithmetic/logical operations, a `MEM` instruction for accessing the fields of objects, `BRA` instructions and special instructions for method invocation (`METHOD`). Two special instructions (`ACTIVATE` and `DROP`) maintained live range information for the register allocator.

P5

The second pass of the compiler translates each JavaVM `load` or `store` instruction into a corresponding intermediate code `MOVE` instruction using a new register as the destination register in the case of a `load`. Always using a new register yields code in a similar form to static single assignment form [3], which is commonly used for compiler optimizations. A JavaVM `iadd` instruction is translated into an `OP2` instruction, again using a new destination register.

This naive translation scheme would generate many `MOVE` instructions. Therefore `MOVE` instructions are generated lazily. The translator keeps lists which track which registers should contain the same values (that are registers which are just copies of another register). Instead of generating a `MOVE` instruction, the translator enters the register into a copy list. If the translator should later generate a `DROP` instruction, it deletes the register from the list.

When at control flow joins the register lists did not match, the corresponding `MOVE` instruction had to be generated. But for most joins the stack, and therefore the register lists, are empty or else the registers are compatible. Furthermore the register allocator tries to assign the same hardware register to the same stack slots so that `MOVE` instructions can be eliminated.

### 2.3 Old register allocation

For a just-in-time compiler expensive register allocation algorithms, like graph coloring, cannot be used. We therefore designed a simple and fast scheme. There are two different sets of registers: registers for stack slots and registers for local variables. First, registers for stack slots are assigned. Afterwards, the remaining registers are assigned to the local variables which are active in the whole method.

All registers are assigned to a CPU register at the beginning of a basic block. An existing allocation is left unchanged. The allocator scans the instructions and, for each instruction which activates a register and to which no CPU register has been assigned, a new CPU register is selected. If the allocator has run out of CPU registers, the register is spilled to memory. There exist some conventions for the assignment of registers when calling methods. To prevent unnecessary copy instructions at a method call prior to the allocation pass, pseudo registers which are method parameters or return values are assigned the correct register (precoloring).

### 2.4 Problems of the old scheme

The old compiler used a lot of doubly linked lists and allocated every object explicitly. So a large amount of memory was used and a large percentage of the compile time was spent in object allocation. It had to do four passes over the code and there were examples in our applications where the compiler took up to fifty percent of the total run time. So we searched for improvements and designed a new translation algorithm.

## 3 The new translation algorithm

The new translation algorithm can get by with three passes. The first pass determines basic blocks and builds a representation of the JavaVM instructions which is faster to decode. The second pass analyses the stack and generates a static stack structure. During stack analysis variable dependencies are tracked and register requirements are computed. In the final pass register allocation of temporary registers is combined with machine code generation.

The new compiler computes the exact number of objects needed or computes an upper bound and allocates the memory for the necessary temporary data structures in three big blocks (the basic block array, the instruction array and the stack array). Eliminating all the double linked lists also reduced the memory requirements by a factor of five.

### 3.1 Basic block determination

The first pass scans the JavaVM instructions, determines the basic blocks and generates an array of instructions which has fixed size and is easier to decode in the following passes. Each instruction contains the opcode, two operands and a pointer to the static stack structure after the instruction (see next sections). The different opcodes of JavaVM instructions which fold operands into the opcode are represented by just one opcode in the instruction array.

### 3.2 Basic block interfacing convention

The handling of control flow joins was quite complicated in the old compiler. We therefore introduced a fixed interface at basic block boundaries. Every stack slot at a basic block boundary is assigned a fixed interface register. The stack analysis pass determines the type of the register and if it has to be saved across method invocations. To enlarge the size of basic blocks method invocations do not end basic blocks. To guide our compiler design we did some static analysis on a large application written in Java: the javac compiler and the libraries it uses. Table 1 shows that in more than 93% of the cases the stack is empty at basic block boundaries and that the maximal stack depth is 6. Using this data it becomes clear that the old join handling did not improve the quality of the machine code.

### 3.3 Copy elimination

To eliminate unnecessary copies loading of values is delayed until the instruction is reached which consumes the

| stack depth | 0 | 1 | 2 | 3 | 4 | 5 | 6 | >6 |
|---|---|---|---|---|---|---|---|---|
| occurrences | 7930 | 258 | 136 | 112 | 36 | 8 | 3 | 0 |

**Table 1. distribution of stack depth at block boundary**

value. To compute the information the run time stack is simulated at compile time. Instead of values the compile time stack contains the type of the value, if a local variable was loaded to a stack location and similar information. Adl-Tabatabai [1] used a dynamic stack which is changed at every instruction. A dynamic stack only gives the possibility to move information from earlier instructions to later instructions. We use a static stack structure which enables information flow in both directions.

Fig. 1 shows our instruction and stack representation. An instruction has a reference to the stack before the instruction and the stack after the instruction. The stack is represented as a linked list. The two stacks can be seen as the source and destination operands of an instruction. In the implementation only the destination stack is stored, the source stack is the destination of stack of the previous instruction.

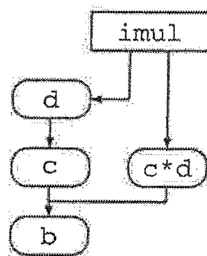



**Figure 1. instruction and stack representation**

This representation can easily be used for copy elimination. Each stack element not only contains the type of the stack slot but also the local variable number of which it is a copy, the argument number if it is an argument, the interface register number if it is an interface. Load (push the content of a variable onto the stack) and store instructions do no generate a copy machine instruction if the stack slot contains the same local variable. Generated machine instructions for arithmetic operations directly use the local variables as their operands.

There are some pitfalls with this scheme. Take the example of fig. 2. The stack bottom contains the local variable `a`. The instruction `istore a` will write a new value for `a` and will make a later use of this variable invalid. To avoid this we have to copy the local variable to a stack variable. An important decision is at which position the copy instruction should be inserted. Since there is a high number of `dup` instructions in Java programs (around 4%) and it is possible that a local variable resides in memory, the copy should be done with the `load` instruction. Since the stack is represented as a linked list only the destination stack has to be checked for occurrences of the offending variable and these occurrences are replaced by a stack variable.

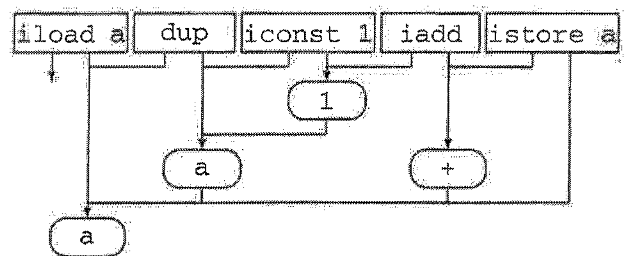



**Figure 2. anti dependence**

To answer the question of how often this could happen and how expensive the stack search is, we analyzed again the javac compiler. In more than 98% of the cases the stack is empty (see table 2). In only 0.2% of the cases the stack depth is higher than 1 and the biggest stack depth is 3.

| stack depth | 0 | 1 | 2 | 3 | >3 |
|---|---|---|---|---|---|
| occurrences | 2167 | 31 | 1 | 3 | 0 |

**Table 2. distribution of `store` stack depth**

To avoid copy instructions when executing a `store` it is necessary to connect the creation of a value with the store which consumes it. In that case a `store` not only can conflict with copies of a local variable which result from `load` instructions before the creator of the value, but also with `load` and `store` instructions which exist between the creation of value and the `store`. In fig. 3 the `iload a` instruction conflicts with the `istore a` instruction.

The anti dependences are detected by checking the stack locations of the previous instructions for conflicts. Since the stack locations are allocated as one big array just the stack elements which have a higher index than the current stack element have to be checked. Table 3 gives the distribution of the distance between the creation of the value and the corresponding store. In 86% of the cases the distance is one.

The output dependences are checked by storing the instruction number of the last store in each local variable. If

| chain length | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | >9 |
|---|---|---|---|---|---|---|---|---|---|---|
| occurrences | 1892 | 62 | 23 | 62 | 30 | 11 | 41 | 9 | 7 | 65 |

**Table 3. distribution of creator-store distances**

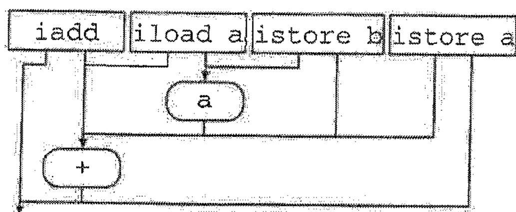



**Figure 3. anti dependence**

a store conflicts due to dependences the creator places the value in a stack register. Additional dependences arise because of exceptions. The exception mechanism in Java is precise. Therefore `store` instructions are not allowed to be executed before an exception raising instruction. This is checked easily by remembering the last instruction which could raise an exception. In methods which contain no exception handler this conflict can be safely ignored because no exception handler can have access to these variables.

### 3.4 Register allocation

Expensive register allocation algorithms are neither suitable nor necessary. The javac compiler does a coloring of the local variables and assigns the same number to variables which are not active at the same time. The stack variables have implicitly encoded their live ranges. When a value is pushed, the live range start. When a value is popped, the live range ends.

Complications arise only with stack manipulation instructions like `dup` and `swap`. We flag therefore the first creation of a stack variable and mark a duplicated one as a copy. The register used for this variable can be reused only after the last copy is popped.

During stack analysis stack variables are marked which have to survive a method invocation. These stack variables and local variables are assigned callee saved registers. If there are not enough registers available, these variables are allocated in memory.

Efficient implementation of method invocation is crucial to the performance of Java. Therefore, we preallocate the argument registers and the return value in a similar way as we handle store instructions. Input arguments (in Java input arguments are the first variables) for leaf procedures (and input arguments for processors with register windows) are preassigned, too.

### 3.5 Instruction combining

Together with stack analysis we combine constant loading instructions with selected instructions which are following immediately. In the class of combinable instructions are add, subtract, multiply and divide instructions, logical and shift instructions and compare/branch instructions. During code generation the constant is checked if it lies in the range for immediate operands of the target architecture and appropriate code is generated.

The old translator expanded some complex instructions into multiple instructions to avoid complex instructions in the later passes. One of such instructions was the expansion of the `lookup` instruction in a series of load constant and compare and branch instructions. Since the constants are usually quite small this unnecessarily increased the size of the intermediate representation and the final code. The new compiler delays the expansion into multiple instructions to the code generation pass which reduces all representations and speeds up the compilation.

### 3.6 Example

Fig. 4 shows the intermediate representation and stack information as produced by the compiler for debugging purposes. The `Local Table` gives the types and register assignment for the local variables. The Java compiler reuses the same local variable slot for different local variables if there life ranges do not overlap. In this example the variable slot 3 is even used for local variables of different types (integer and address). The JIT-compiler assigned the saved register 12 to this variable.

One interface register is used in this example entering the basic block with label `L004`. At the entry of the basic block the interface register has to be copied to the argument register `A00`. This is one of the rare cases where a more sophisticated coalescing algorithm could have allocated an argument register for the interface.

The combining of a constant with an arithmetic instruction happens at instruction 2 and 3. Since the instructions are allocated in an array the empty slot has to be filled with a `NOP` instruction. The `ADDCONSTANT` instruction already has the local variable `L02` as destination, an information which comes from the later `ISTORE` at number 4. Similarly the `INVOKESTATIC` at number 31 has marked all its operands as arguments. In this example all copies (beside the one to the interface register) have been eliminated.

98

| | sieve | JavaLex | javac | espresso | Toba | java_cup |
|---|---|---|---|---|---|---|
| run time on 21164A 600MHz (in seconds) | | | | | | |
| CACAO old total | 1.120 | 0.720 | 1.336 | 0.858 | 1.208 | 0.398 |
| load | 0.040 | 0.067 | 0.224 | 0.141 | 0.068 | 0.077 |
| compile | 0.022 | 0.116 | 0.343 | 0.235 | 0.139 | 0.196 |
| run | 1.058 | 0.537 | 0.769 | 0.481 | 1.000 | 0.125 |
| CACAO new total | 0.902 | 0.522 | 0.925 | 0.614 | 0.982 | 0.218 |
| load | 0.040 | 0.067 | 0.223 | 0.141 | 0.068 | 0.077 |
| compile | 0.004 | 0.018 | 0.060 | 0.050 | 0.019 | 0.026 |
| run | 0.858 | 0.437 | 0.642 | 0.423 | 0.895 | 0.115 |
| speedup | | | | | | |
| speedup total old/new | 1.24 | 1.38 | 1.44 | 1.40 | 1.23 | 1.82 |
| speedup compile old/new | 7.33 | 6.44 | 5.62 | 4.70 | 7.31 | 7.53 |
| number of compiled JavaVM instructions | | | | | | |
| | 2514 | 13412 | 34759 | 27281 | 14430 | 17489 |
| number of cycles per compiled JavaVM instruction | | | | | | |
| | 955 | 805 | 1035 | 1099 | 790 | 891 |

**Table 4. comparison between old and new compiler**

### 3.7 Complexity of the algorithm

The complexity of the algorithm is mostly linear with respect to the number of instructions and the number of local variables plus the number of stack slots. There are only a small number of spots where it is not linear.

- At the begin of a basic block the stack has to be copied to separate the stacks of different basic blocks. Table 1 shows that the stack at the boundary of a basic block is in most cases zero. Therefore, this copying does not influence the linear performance of the algorithm.
- A store has to check for a later use of the same variable. Table 2 shows that this is not a problem, too.
- A store additionally has to check for the previous use of the same variable between creation of the value and the store. The distances between the creation and the use are small (in most case only 1) as shown by table 3.

Compiling javac 29% of the compile time are spent in parsing and basic block determination, 18% in stack analysis, 16% in register allocation and 37% in machine code generation.

## 4 Results

To evaluate the differences between the old and the new compiler we used six different programs: sieve is a Java implementation of the well known prime number generation program, JavaLex is a scanner generator, javac is the Java compiler from sun, espresso is a compiler for an enhanced Java dialect, Toba is a system which translates Java class files to C and java_cup is a parser generator. As input data for javac, espresso and Toba we used all source files of Toba (18 files).

Table 4 shows the total run time, the load time, the compile time and the run time for the old and the new system on an Alpha workstation with a 600Mhz 21164a processor. The new compiler is between 5 and 7 times faster than the old compiler. The new system also has some improvements in the code generation and uses a hardware null pointer check ([11]). Both improvements together speed up the new sytem between 23% and 82%. On average only 800 to 1100 cycles are needed to compile one JavaVM instruction. A profiler which assumes that all memory accesses go to the first level cache computed 423 cycles per compiled JavaVM instruction.

To evaluate the performance of CACAO we compared it with Sun's JDK and with kaffe version 0.8 (see section 1.1). We also got access to a beta version of Digitals JIT compiler. Due to problems with the monitor implementation ([11]) this compiler gives very bad results for javac and similar programs (three times slower than the JDK), but produced effiecient code for the sieve benchmark.

Table 5 gives the run times for all these systems on an ALPHA workstation with a 300MHz 21064a processor. The CACAO system is between 3 and 5 times faster than the kaffe system and twice as fast as the Digital JIT compiler.

|  | sieve | JavaLex | javac | espresso | Toba | java_cup |
|---|---|---|---|---|---|---|
| run time on 21064A 300MHz (in seconds) | | | | | | |
| JDK | 83.2 | 29.8 | 18.5 | 8.7 | 32.1 | 3.5 |
| Digital JIT | 6.27 | 84.4 | 47.6 | 14.1 | - | 9.8 |
| kaffe | 9.14 | 9.9 | 17.8 | 12.5 | - | 2.98 |
| CACAO old | 4.80 | 2.65 | 4.74 | 3.17 | 4.58 | 1.52 |
| CACAO new | 3.87 | 1.92 | 3.29 | 2.26 | 3.72 | 0.83 |
| speedup with respect to interpreter | | | | | | |
| speedup JDK/DEC-JIT | 13.3 | 0.35 | 0.38 | 0.48 | - | 0.36 |
| speedup JDK/kaffe | 9.10 | 3.01 | 1.04 | 0.7 | - | 1.17 |
| speedup JDK/CACAO old | 17.3 | 11.24 | 3.90 | 2.74 | 7.01 | 2.30 |
| speedup JDK/CACAO new | 21.5 | 15.52 | 5.62 | 3.85 | 8.62 | 4.22 |

**Table 5. comparison between JDK, Digital JIT, kaffe and CACAO**

## 5 Conclusion and further work

We presented an efficient algorithm for translating the JavaVM to efficient native code for RISC processors. This new algorithm is about seven times faster than the compiler used before. CACAO executes Java programs up to 5 times faster than other JIT compilers. CACAO can be obtained via the world wide web at `http://www.complang.tuwien.ac.at/java/cacao/`. Currently additional code generators for the Sparc, MIPS and PowerPC processors are being developed. We are working to integrate bound check removal, instruction scheduling and method inlining.

## Acknowledgement

We express our thanks to Manfred Brockhaus, David Gregg and Anton Ertl for their comments on earlier drafts of this paper.

## References

[1] A.-R. Adl-Tabatabai, M. Ciernak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Conference on Programming Language Design and Implementation*, volume 33(6) of *SIGPLAN*, page to appear, Montreal, 1998. ACM.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] K. Ebcioğlu, E. Altman, and E. Hokenek. A Java ILP machine based on fast dynamic compilation. In *MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.

[5] M. A. Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, April 1996.

[6] M. A. Ertl and M. Maierhofer. Translating Forth to native C. In *EuroForth '95*, 1995.

[7] M. A. Ertl and C. Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, 1997.

[8] K. J. Gough. Multi-language, multi-target compiler development: Evolution of the Gardens Point compiler project. In H. Mössenböck, editor, *JMLC'97 –Joint Modular Languages Conference*, Linz, 1997. LNCS 1204.

[9] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'29)*, 1996.

[10] A. Krall and R. Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[11] A. Krall and M. Probst. Monitors and exceptions: How to implement Java efficiently. In S. Hassanzadeh and K. Schauser, editors, *ACM 1998 Workshop on Java for High-Performance Computing*, pages 15–24, Palo Alto, March 1998. ACM.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[13] S. Pemberton and M. C. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.

[14] T. B. Steel. A first version of UNCOL. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*, pages 371 – 377, 1961.

[15] A. S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen, and C. J. H. Jacobs. The design of very fast portable compilers. *ACM SIGPLAN Notices*, 24(11):125–131, Nov. 1989.

[16] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 16(9):654–660, September 1983.

100

```
java.io.ByteArrayOutputStream.write (int)void

Local Table:
     0:                 (addr) S15
     1:     (int) S14
     2:     (int) S13
     3:     (int) S12   (addr) S12

Interface Table:
     0:     (int) T24

[                  L00]        0  ALOAD           0
[                  T23]        1  GETFIELD        16
[                  L02]        2  IADDCONST       1
[                  L02]        3  NOP
[                     ]        4  ISTORE          2
[                  L02]        5  ILOAD           2
[              L00 L02]        6  ALOAD           0
[              T23 L02]        7  GETFIELD        8
[              T23 L02]        8  ARRAYLENGTH
[                     ]        9  IF_ICMPLE       L005
                                 ..............

[                     ]       18  IF_ICMPLT       L003
[                     ] L002:
[                  I00]       19  ILOAD           3
[                  I00]       20  GOTO            L004
[                     ] L003:
[                  I00]       21  ILOAD           2
[                  A00] L004:
[                  L03]       22  BUILTIN1        newarray_byte
[                     ]       23  ASTORE          3
[                  L00]       24  ALOAD           0
[                  A00]       25  GETFIELD        8
[              A01 A00]       26  ICONST          0
[          A02 A01 A00]       27  ALOAD           3
[      A03 A02 A01 A00]       28  ICONST          0
[ L00 A03 A02 A01 A00]        29  ALOAD           0
[ A04 A03 A02 A01 A00]        30  GETFIELD        16
[                     ]       31  INVOKESTATIC    java/lang/System.arraycopy
[                  L00]       32  ALOAD           0
[              L03 L00]       33  ALOAD           3
[                     ]       34  PUTFIELD        8
[                     ] L005:
                                 ..............

[                     ]       45  RETURN
```

**Figure 4. Example: intermediate instructions and stack contents**

101

# Technical Overview of the Common Language Runtime

Erik Meijer
Microsoft
Redmond WA
emeijer@microsoft.com

John Gough
QUT
Brisbane, Australia
j.gough@qut.edu.au

## Abstract

*The functionality of the recently announced Microsoft .NET system is founded on the capabilities of the Common Language Infrastructure (CLI). Unlike some other recent systems based on virtual machines, the CLI was designed from the start to support a wide range of programming languages. It is also expected that ECMA standardization will make the CLI available on a wide range of computing platforms. This combination of multi-language capability and multiplatform implementation make the CLI an important target for future language compilers.*

*In this paper, the technical details of the CLI are briefly described. To motivate some of the discussion a comparison is made with the Java$^{TM}$ virtual machine (JVM). The JVM was designed under rather different constraints, making it a much more difficult target for languages other than Java$^{TM}$. We also briefly discuss the issues involved in mapping various language constructs to the primitives of the CLI.*

## 1 Introduction

The ideas of virtual machines, intermediate languages and language independent execution platforms have fascinated language researchers for a long time. Well known examples include UNCOL [6], UCSD P-code [23], ANDF [20], AS-400 [25], hardware emulators such as VMWare, Transmeta Crusoe$^{TM}$ [30], binary translation [26], the JVM [19], and most recently Microsoft's *Common Language Infrastructure* (CLI) [2].

There are several reasons why people are looking at alternative implementation paths for native compilers:

**Portability** By using an intermediate language, you need only $n + m$ translators instead of $n * m$ translators, to implement $n$ languages on $m$ platforms.

**Compactness** Intermediate code is often much more compact than the original source. This was an important property back in the days when memory was a limited resource, and has recently regained importance in the context of dynamically downloaded code.

**Efficiency** By delaying the commitment to a specific native platform as much as possible, the execution platform can make optimal use of the knowledge of the underlying machine, or even adapt to the dynamic behavior of the program.

**Security** High-level intermediate code is more amenable to deployment and runtime enforcement of security and typing constraints than low level binaries.

**Interoperability** By sharing a common type system and high-level execution environment (that provides services such as a common garbage collected heap, threading, security, etc), interoperability between different languages becomes easier than binary interoperability. Easy interoperability is a prerequisite for multi-language library design and software component reuse.

**Flexibility** Combining high level intermediate code with metadata enables the construction of (typesafe) metaprogramming concepts such as reflection, dynamic code generation, serialization, type browsing etc.

Attracted by the high-level runtime support and the wide availability of the JVM, and the rich set of libraries on the Java$^{TM}$ platform, quite a number of language implementers have recently turned to the JVM as the execution environment for their language [29, 7].

The JVM is a great target for Java$^{TM}$, but even though the JVM designers hope to attract implementers of other languages [19, Chapter 1.2], we will argue that the JVM is essentially a suboptimal multi-language platform.

For a start, the JVM provides no way of encoding type-unsafe features of typical programming languages, such as pointers, immediate descriptors (tagged pointers), and unsafe type conversions. Furthermore, in many cases the JVM lacks the primitives to implement language features that are not found in Java$^{TM}$, but are present in other languages. Examples of such features include unboxed structures and unions (records and variant records), reference parameters, varargs, multiple return values, function pointers, overflow sensitive arithmetic, lexical closures, tail calls, fully dynamic dispatch, generics, structural type equivalence etc [17, 18, 14, 9, 12, 11, 24].

The CLI has been designed from the ground up as a target for multiple languages, and explicitly addresses many of the issues mentioned above that are needed to efficiently compile a wide variety of languages. To ensure this, from early on in the development process of the CLI, Microsoft has worked closely with a large number of language implementers (both commercial and academic, for an up to date list see `www.gotdotnet.com`). For instance, the tail call instruction was added as a direct result of feedback from language researchers; tail calls are a necessary condition for efficiency in many declarative languages that use recursion as their sole way of expressing repetition.

It would be unfair to state that the CLI as it is now, is already the *perfect* multi-language platform. It currently has good support for imperative (COBOL, C, Pascal, Fortran) and statically typed OO languages (such as C♯, Eiffel, Oberon, Component Pascal). Microsoft continues to work with language implementers and researchers to improve support for languages in non-standard paradigms [16].

In the remainder of this paper, we give a quick overview of the architecture, instruction set and type system of the CLI and point out specific points where we think the CLI is a better multi-language execution environment than the JVM. The treatment is necessarily brief. For a more detailed and tutorial overview of the CLI, see the recent book [10].

## 2 Architecture of the Common Language Infrastructure (CLI)

The CLI manages multiple concurrent threads of control (which are not necessarily native OS threads). A thread can be viewed as a singly linked list of *activation records*[13, 3], where a activation record is created and linked back to the current record by a method call instruction, and removed when the method call completes (either by a normal return, a tailcall, or by an exception). It is usual, but not necessary, that the activation records of a single thread are allocated on a *runtime stack*. However, since the management of activation records is abstracted away in the CLI, and to avoid confusion, we shall use the term "stack" here exclusively to refer to the *evaluation stack* of the virtual machine.

**An instruction pointer (IP)** which points to the next CLI instruction to be executed by the CLI in the present method.

**An evaluation stack** which contains intermediate values of the computation performed by the executing method (the *operand stack* in JVM terminology).

**A (zero-based) array of local variables** A local variable may hold any data type. However, a particular variable must be used in a type-consistent way (in the JVM, a local variable can contain an integer at one point in time and a float at another).

**A (zero-based) array of incoming arguments** Unlike the JVM the argument array and the local variable array are not the same.

**A methodInfohandle** which contains information about the method, such as its signature, the types of its local variables, and data about its exception handlers.

**A local memory pool** The CLI includes instructions for dynamic allocation of objects from the local memory pool (e.g. [3, Chapter 7.3, page 408].

**A return state handle** which is used to restore the method state on return from the current method. This corresponds to what in conventional compiler terminology would be the *dynamic link*.

**A security descriptor** which is used by the CLI security system to record security overrides (assert, permit-only, and deny). This descriptor is not directly accessible to managed code. Although extremely important and interesting, the security mechanism of the CLI is outside the scope of this paper.

In contrast to the JVM where all storage locations (local variables, stack slots, arguments) are 4 bytes wide, storage locations in the CLI are polymorphic, in the sense that they might be 4 bytes (such as a 32 bit integer) or hundreds of bytes (such as a user-defined value type), but their type is fixed for lifetime of the frame.

## 3 Assemblies

Every execution environment has a notion of "software component" [28]. An *assembly* is a set of files (modules) containing Common Intermediate Language (CIL) code and metadata, that serves as the primary unit of a software component in the CLI. Security, versioning, type resolution, processes (application domains) all work on a per assembly basis. In JVM terms an assembly could roughly be compared to a JAR file.

An assembly *manifest* describes information about the assembly itself, such as its version, which files make up the assembly, which types are exported from this assembly, and optionally a digital signature and public key of the manifest itself. Here is an example manifest for an assembly using ILASM syntax [2]:

```
.assembly HelloWorld {}

.assembly extern mscorlib {
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 1:0:2411:0
}
```

Inside an assembly or module we can define *reference* types such as classes, interfaces, arrays, delegates) (see section 7) and *value types* such as structs, enums (see section 6), and nested types. In contrast to the JVM, the CLI allows top-level methods and fields. All these declarations are included in the assembly's *metadata*. A unique feature of the CLI is that it's metadata is user extensible via the notion of custom attributes.

For a more detailed and tutorial overview of the role of assemblies as software components, see [21].

## 4 Type System

In this section we give an informal overview of the CLI type system, a more formal introduction is given by Gordon and Syme [8].

In addition to user defined types (section 6 and section 7), the CLI supports the following set of *primitive types*:

- `object`, shorthand for `System.Object`, `string`, shorthand for `System.String`, `void`, void return type.
- `bool`, 8-bit 2's complement signed value, `char`, 16-bit Unicode character.
- `int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`, `int64`, `unsigned int64`, unsigned and 2's complement signed integers of respective width; `native int`, `unsigned native int`, machine dependent unsigned and 2's complement signed value.
- `float32`, `float64`, IEEE-754 floating point value of respective width; `native float`, machine dependent floating point number (not user visible).
- `typed reference`, an opaque descriptor of a pair of a pointer and a type, used for type safe varargs.

Primitive types can be combined into composite types using the following set of *type constructors*:

- `valuetype` **`typeref`**, `class` **`typeref`**, reference to value or reference type.
- **`type`** `pinned`, prevents the object at which local variable points from being moved by GC. This is outside the scope of this paper.
- **`type`**`[`**`bounds`**`]`, (multi-dimensional) array. This is outside the scope of this paper, suffice to note that in constrast to the JVM, the CLI *does* support true multi-dimensional arrays.
- `method` **`callConv`** **`type`**`*(`**`parameters`**`)`, function pointer. This is outside the scope of this paper.

- **type**&, managed pointer to **type**.
- **type***, unmanaged pointer to **type**.

The natural-size, or *generic*, types (primitive types `native int`, `unsigned native int`, `object`, and the two type constructors &, *) are a mechanism in the CLI for deferring the choice of a value's size. The CLI maps each to the natural size for a specific processor at JIT- or run-time. For example, a native int would map to `int32` on a Pentium processor, but to `int64` on an IA64 processor.

The `object` type represents an object reference that is managed by the CLI. A *managed pointer* & is similar to the `object` type, but points to the interior of an object. Managed pointers are not interchangeable with object references. *Unmanaged pointers* * or `native int` are the traditional pointers of other runtime systems, that is, the addresses of data. Unmanaged pointers are an essential element for the interoperation of CLI programs with native code components. Such pointers may not point into the managed heap since such heap values are under the control of a garbage collector that is free to move and compact objects. Conversely, values of managed pointer type may safely point outside the managed heap, since the garbage collector knows the heap limits.

Natural sized types offer a significant advantage over the JVM which prematurely commits all storage locations to be 32 bits wide. This implies for example that values of type `long` or `double` occupy two locations, which makes things unnecessarily hard for compiler writers.

A more important weakness of the JVM as a target for multiple language is the fact that its type system lumps together all pointers into one `reference` type, closing the door for languages or compilers that do need a more fine-grained level of detail. We will expand on the usefulness of the CLI pointer types in more detail in section 9.

## 5 Base Instruction set

The CLI has about 220 instructions, so obviously we do not have space to cover all of them in this paper, instead we will highlight a few representative instructions from each group below[1].

When comparing to JVM instructions, you will notice that unlike the JVM where most instructions have the types of their arguments hard-coded in the instruction (which makes it easier to *interpret* JVM byte code, but puts a burden on every compiler that generates JVM byte codes), the CLI instruction set is much more polymorphic and usually only requires explicit type information for the result of an instruction (which makes it easier for compilers to generate CIL code, but requires more work from the JIT).

### 5.1 Constants, arguments, local variables, and pointers

The CLI provides a number of instructions for transferring values to and from the evaluation stack. Instructions that push values on the evaluation stack are called "loads", and instructions that pop elements from the stack into local variables are called "stores".

The simplest load instruction is `ldc.t` **v**, that pushes the value **v** of type **T**[2] on the evaluation stack. The `ldnull` pushes a null reference (of type `object`) on the stack.

The `ldarg` **n** instruction pushes the contents of the **n**-th argument on the evaluation stack. The `ldarga` **n** instruction pushes the *address* (as a managed pointer of type **T**&) of the **n** argument on the evaluation stack. The `starg` **n** instruction pops a value from the stack and stores it in the **n**-th argument. In each case, the JIT knows the type of the value from the signature of the method.

The `ldloc` **n** instruction pushes the contents of the **n**-th local variable onto the evaluation stack, and `ldloca` **n** pushes the *address* of the **n**-th local variable on the evaluation stack as a managed pointer. The `stloc` **n** instruction pops a value from the stack and stores it in the **n**-th argument. Again, the JIT can figure out the types of these values from the context.

[1] Many of the CLI instruction also have short forms, that allow more compact representation in certain special cases. We will not discuss these variants here

[2] Here **T** ∈ {`int32`, `int64`, `float32`, `float64`} and **t** is the short form of **T**. The short form of types is used in all instructions that have a type index.

The `ldind.t` instruction expects an address (which can be a native int, or a unmanaged or managed pointer) on the stack, dereferences that pointer and puts the value on the stack. The `stind.t v` instruction stores a value **v** of type **T** at address found at the top of the stack. In both cases, the type **t** is needed because the JIT cannot always infer what the type of the resulting value is.

The other load and store instructions include `ldfld`, `ldsfld`, `stfld`, `stsfld`, and `ldflda` and `ldsflda` to manipulate instance and static fields, and a similar family of instructions for arrays.

**Example: reference arguments** The ability to load the address of local variables, and to dereference pointers to indirectly get the value they point at allows compiler writers to efficiently implement languages that support passing arguments by reference. For example, here is the CIL version of the `Swap` function that swaps the values of two variables:

```
.method static void Swap(int32& xa, int32& ya) {
  .maxstack  2
  .locals (int32 z)
    ldarg xa; ldind.i4; stloc z
    ldarg xa; ldarg ya; ldind.i4
    stind.i4; ldarg ya; ldloc z
    stind.i4; ret // return
}
```

To call this function (see section 8), we just pass the addresses of the local variables as arguments to function `Swap`:

```
.locals (int32 x, int32 y)
// initialize x and y
  ldloca   x
  ldloca   y
  call     void Swap(int32&, int32&)
```

In the JVM there is a separate load (and store) instruction for each type, i.e. `iload_n` pushes the integer content of the **n**-th local variable on the stack, and similarly for `aload_n` (reference), `dload_n` (double, so it will moved as two 32 bit values), `fload_n` (float), and `lload_n` (long, again, moves two items will be moved).

The JVM does not allow compilers to take the address of local variables, hence it is impossible to implement byref arguments directly. Instead compiler writers have to resort to tricks such as passing one-element arrays, or by introducing explicit box classes (the JVM does not support boxing and unboxing either). Gough [12] gives a detailed overview of the intricate design space of implementing reference arguments on the JVM.

### 5.2 Arithmetic

The `add` instruction adds the two topmost values on the stack together (and similarly for other arithmetic instructions). Overflow is not normally detected for integral operations unless you specify `.ovf` (signed) or `ovf.un` (unsigned); floating-point overflow returns $+\infty$ or $-\infty$.

The JVM *never* indicates overflow during operations on integer data types, which means that the time penalty may be significant for procedures which perform intensive arithmetic in languages (such as Ada95 [1] or SML [22]) that require overflow detection. A minor issue in this context, is that there is a separate `add` instruction for each type (and similar for other arithmetic instructions), just as is the case for load and store.

### 5.3 Simple control flow

The CLI supports the usual variety of (conditional) branch instructions (such as `br`, `beq`, `bge` etc.). There is no analog of the JVM "jump subroutine" instruction. Also the CLI does not limit the length of branches to 64K as the JVM does (which might not be a big deal for humans programming in Java, but it is a real problem for compilers generating JVM byte code).

# 6 Value Types

A *value type* is similar to a struct in C or record in Pascal, i.e. a sequence of named fields of various types. In contrast to reference types, which are always allocated on the GC heap, value types are allocated "in place". In the CLI, value types can also contain (static, virtual, or instance) methods [2], the details of which are outside the scope of this paper.

## 6.1 Structures

Here is the definition of a simple `Point` structure that contains two fields `x` and `y` (which the CLI may store in any order):

```
.class value Point {
  .field public int x
  .field public int y
}
```

## 6.2 Unions

The CLI also supports sequential and explicit layout control of fields. The latter is needed to implement C-style *union types* (or variant records in Pascal), a structure where the fields may overlap. For example the following value class defines a union that may hold either a float or an int:

```
.class value explicit FloatOrInt {
  .field [0] public float32 f
  .field [0] public int32   n
}
```

## 6.3 Enums

Besides structures, there is another kind of value type, *enumerations*, which correspond to C-style enums. Enumerations provide a type safe way to associate names with integer values. For example the following enum defines a new value type `Shape` with two constants `RECTANGLE` and `CIRCLE`:

```
.class enum Shape {
  .field public static valuetype Shape RECTANGLE = int32(0)
  .field public static valuetype Shape CIRCLE    = int32(1)
}
```

The CLI also allows you to specify enum details such as the internal storage type or indicating that the enumeration is a collection of bits, for more details see [2].

## 6.4 Initializing valuetypes

Except for boxing and the `.locals` directive, the CLI does not have special mechanisms or instructions to explicitly allocate memory for a valuetype. The `initobj T` instruction expects the address of a valuetype `T` on the stack, and initializes all the fields of the valuetype to either `null` or a 0 of the appropriate primitive type (this is a nice example of a *polytypic* instruction). For example to initialize the example `Point` struct that we introduced in section 6.1, we would load the address of the local variable `p` of type `Point` on the stack and call `initobj Point`:

```
.locals (valuetype Point p)
  ldloca   p
  initobj  Point
```

It should be obvious that having value types is essential for compiling Pascal or C-like languages that have enums, record and union types. Compiling such languages to the JVM is inefficient to start with, as you need to represent enums and structs by classes and unions by class hierarchies [4, Chapter5]. A much more serious consequence is that it is impossible to support the full semantics of such languages, as it is impossible to implement the common (type unsafe) trick where you store a a float in an `FloatOrInt` union type, and read it as an int:

```
.locals (valuetype FloatOrInt fi, int32 n)
   // fi.f = 3.14
   ldloca     fi
   ldc.r4     3.14
   stfld      float32 FloatOrInt::f
   // n = fi.n
   ldloca     fi
   ldfld      int32 FloatOrInt::n
```

## 7 Reference types

The CLI supports types such as classes, interfaces, arrays, delegates. Because of lack of space, we will restrict our attention to classes. Classes can contain methods and fields; but yet again, to support as many languages as possible, besides virtual and static methods (as in Eiffel, and Java$^{TM}$), the CLI also support instance methods (as in C++).

For example, here are two classes `Foo` and `Bar` that both define an instance method `f`, and a virtual method `g`:

```
.class public Foo {
  .method public virtual void f() {...}
  .method public instance void g() {...}
  .method public static void h() {...}
  .method public specialname void .ctor() {...}
  }

.class public Bar extends Foo {
  .method public virtual void f() {...}
  .method public instance void g() {...}
  .method public static void h() {...}
  .method public specialname void .ctor() {...}
}
```

Constructors always are names `.ctor` and have to be marked as `specialname`.

### 7.1 Instantiating Reference types

The `newobj c` instruction allocates a new instance of the class associated with constructor `c` and initializes all the fields in the new instance. It then calls the constructor with the given arguments along with the newly created instance.

For example, we can create an instance `f` with static type `Foo` of our class `Foo`, and an instance `b` with static type `Foo` of our class `Bar` using the following instruction sequence:

```
.locals (class Foo f, class Foo b)
    newobj void Foo::.ctor(); stloc f
    newobj void Bar::.ctor(); stloc b
```

To create an instance of a class **c** in the JVM, you always have to use the sequence new **c**; dup; invokespecial **c**.<init>()V (and similarly for using a constructor that takes arguments) and the Java$^{TM}$ verifier must do a complex

dataflow analysis to ensure that no object is used before it is properly initialized or that it is initialized more than once [19, Chapter 4.9.4]. It seems much simpler to avoid all the complexity to start with and just do allocation and initialization in a single instruction.

## 8 Invoking methods

The CLI has two call instructions for directly invoking methods and interfaces. A third call instruction `calli` allows indirect calls on a function pointer, but this is outside the scope of this paper.

The `call m` instruction is normally used to call a static method **m** (i.e. it is comparable to the `callstatic` instruction in the JVM). For example, to call method `Foo::h()`, we just write:

```
call void Foo::h()
```

It is legal to call a virtual or instance method using `call instance` (rather than `callvirt`); in which case method lookup is done statically, in other words, you will get an early bound call (i.e. the effect is comparable to a `invokespecial` on the JVM). Assuming that `bar` is a local variable that contains an instance of class `Bar`, the following call would actually execute method `Foo::f()`:

```
ldloc bar;
call instance void Foo::f()
```

The `instance` calling convention indicates that `Foo::f()` expects an additional "this" parameter.

The `callvirt m` instruction makes a late bound call to a virtual method **m**, in other words, the actual method that is invoked depends on the dynamic type of the "this" parameter (the JVM has two separate instructions, `invokevirtual` and `invokeinterface` for this purpose, which once again makes life harder for compiler writers). So in the example below, the method that will be invoked is `Bar::f()` since the this parameter passed to the call has static type `class Foo`, but dynamic type `class Bar`:

```
ldloc bar;
callvirt void Foo::f()
```

For instance methods, `callvirt` will still result in an early bound call.

### 8.1 Tailcalls

Some people find it hard to believe, but there are programming languages where recursion is the only way of expressing repetition (examples include Haskell, Scheme, Mercury). For these languages, it is essential that the underlying execution environment supports tailcalls. The `tail.` prefix instructs the JIT compiler to discard the caller's stack frame prior to making the call, which means that the following method will indeed loop forever instead of throwing a stack overflow exception:

```
.method public static void Bottom() {
  .maxstack  8
  tail. call void Bottom(); ret
}
```

If the call is from untrusted code to trusted code the frame cannot be fully discarded for security reasons.

Since the JVM does not support tailcalls, compiler writers are forced to use tricks like trampolines to artificially force the JVM to discard stack frames [5, 27, 15, 18].

## 9 Interaction between value and reference types

If you have both valuetypes and reference types, programmers will want to use valuetypes in contexts where reference types are required (for instance to store a `Point` in a collection). The same problem occurs in dynamic languages like Scheme and statically typed polymorphic functional languages like Haskell and SML where polymorphic functions expect a uniform argument representation.

To support these scenarios, it is essential to have efficient support from the execution environment to move between the worlds of value- and reference types. Having to create an instance of a class every time you want to pass a valuetype as a reference type has too much performance overhead. Moreover, this would also force you to define a new class for every valuetype, or introduce many unnecessary casts.

The CLR provides built-in support for boxing and unboxing. A valuetype `T` can be turned into reference type `object` using the `box T` instruction, and back into a valuetype using the `unbox T` instruction.

## 10 Various exotica

As a consequence of its multi-language focus, the CLR provides a number of special facilities that are otherwise difficult to synthesize. The case of tail calls has already been mentioned, but there are others as well.

The manipulation of function pointers as values is critical to the implementation of OO languages with arbitrary mechanisms of method dispatch. Support for virtual dispatch in the case of single implementation inheritance with multiple interface implementation is built in. All other cases must rely on explicitly constructed dispatch tables. The `ldftn` instruction loads a function pointer on the stack, and the `calli` instruction invokes the function pointer on the top of the evaluation stack. Another handy instruction for the implementation of multiple inheritance is the `jmp` instruction. This takes a method reference as an argument, and transfers control to the entry point of the nominated method. The instruction provides the functionality required for constructing "trampoline" stubs that are often the preferred way of performing the "this adjustment" in the dispatch of virtual methods with multiple inheritance.

Languages that pass conformant arrays by value must allocate space for the array copy as part of the procedure call. In this case the use of the `localloc` instruction expands the current activation record. The use of this instruction is much preferable to the dynamic allocation of space for the copy on the heap, as in necessary on the JVM. This instruction thus provides the semantics of the *C* `alloca` function.

The final example that will be mentioned here is the `ldtoken` instruction. This instruction loads the runtime type handle of the type reference in the instruction argument. This operation is a basic building block in the reflection mechanisms. It is used when the type reference is known, but no instance of the type is conveniently available. The same functionality may be gained on the JVM by use of the `Class.forName()` function, but in that case the name is bound at runtime.

## 11 Verified and unverified code

The CLR provides a rich set of primitives for the implementation of both typesafe and non-typesafe features. In cases where memory safety is an important factor, the infrastructure allows for a rich subset of the primitives to be used in ways that allow for verification of safety. As is the case with the JVM, the analysis is necessarily conservative, but provides strong guarantees of freedom from certain classes of runtime errors. Verification may take place either at component deployment time, or at load time. As might be expected, verification is based on analysis of the component, and does not rely on trust of the component producer.

In general terms the guarantees that verification provides are similar to those given by the more strict of contemporary statically typed languages. The verifier guarantees that locations holding object references can only reference objects of types that fulfill the contracts of the statically declared type, and that field selection can only access fields valid for the known type. There are some guarantees that cannot be statically verified. In such cases the verifier checks that all usages that cannot be statically checked are protected by runtime tests. For example, it is seldom possible to check that all array indices are within

the known bounds of the array, so the verifier must check that all array accesses are protected by a bounds check. A similar principle applies to field or method accesses that depend on the success of a narrowing type cast.

Apart from the obvious type guarantees that verification must provide, there are also a number of checks that depend on well-formedness of the control flow. For example, the evaluation stack must have the same height and type-compatible content along all paths which join at control flow merge points. Furthermore, in the case of object references the statically known bound on the type of an evaluation stack element is the least common ancestor of the set of bounds on the types incident on the merge point.

If a compiler wishes to generate intermediate code that can be verified, certain constraints must be met. Some instructions, such as the block copy instruction, are inherently unverifiable while operations such as addition are unverifiable when used for address arithmetic. Apart from avoiding certain instructions, it is also necessary to avoid type-unsafe assignments, and some uses of undiscriminated unions[3].

Languages that are statically type-safe should always be able to be compiled down to verifiable CIL, although the mapping of data types may require some inventiveness. In unverified contexts, as an example, languages that have value arrays of statically declared size would normally declare a value class of the required runtime size. In this case array elements would be accessed by indexing into the *memory blob* that represents the value object at runtime, using address arithmetic in the usual way. If the compiler performs its own index bounds checks then such usage will be completely type safe. However, since the verifier does not permit address arithmetic, and cannot recognize all possible explicit bounds checks, an alternative mapping must be found to achieve verifiability. In this example, the solution is to tranparently allocate a reference array of the required size, and use the built in array support of the CLI. This mechanism, using a reference type to represent a value object, is a common idiom for verifiable code. We call such representation objects *reference surrogates*. The mapping of the value semantics to such surrogates is treated in detail in [10].

The important point to be emphasized is that most of the innovations of the CLR are preserved in a verified environment. Thus the use of value classes, reference parameters and even type-safe unions are permitted.

For the compiler writer, verification provides an unexpected and welcome bonus. The offline verifier, `peverify`, detects (and diagnoses) most of the common errors that are made when coding a CIL emitter. During the early stages of testing, submitting output to `peverify` allows most such errors to be detected. In the case of one of the project 7 compilers, Gardent Point Component Pascal, the compiler successfully bootstrapped itself on the first attempt, once `peverify` certified the CIL as being verifiable.

## 12 Conclusions and future work

In the previous sections we have argued that the CLI is already strictly more powerful than the JVM as a multi-language platform. Microsoft Research and the .NET product group continue to work with language inplementors to improve support a wide variety of language paradigms.

We explictly solicit language implementors (including those who now target the JVM) to try to target the CLI and provide us with feedback on how we can make the CLI even better than it is today.

## Acknowledgements

We would like to thank all *Project 7* participants, and Jim Miller, Patrick Dussud, Jim Hogg, Clemens Szysperski, Don Syme, Andrew Kennedy, and Nick Benton, for many discussions on the topics discussed in this paper. Nicks's notes on his previous experiences with compiling SMLj to the JVM were especially helpful.

## References

1. Ada 95 Reference Manual, 1995. ANSI/ISO/IEC-8652:1995.

[3] Strangely, some unions are tolerable to the verifier, provided that references are not overlapped with other types.

111


2. CLI Partition II: Metadata. http://msdn.microsoft.com/net/ecma/, 2001. ECMA TG3.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1986.
4. J. Bloch. *Effective Java Programming Language Guide.* Addison Wesley, 2001.
5. P. Bothner. Kawa —Compiling Dynamic Languages to the Java VM. In *USENIX'98 Technical Conference*, 1998.
6. M. E. Conway. Proposal for an UNCOL. *CACM*, 1(10):5–8, 1958.
7. J. Engel. *Programming for the Java Virtual Machine.* Addison Wesley, 1999.
8. A. Gordon and D. Syme. Typing a Multi-Language Intermediate Code. In *Proceedings POPL'01*, pages 248–260, 2001.
9. J. Gough. Parameter Passing for the Java Virtual Machine. In *Proceedings of the Australasian Computer Science Conference*, 1998.
10. J. Gough. *Compiling for the .NET Common Language Runtime.* Prentice-Hall, Upper Saddle River, NJ, 2001.
11. J. Gough. Stacking them up: A Comparison of Virtual Machines. In *Proceedings ACSAC-2001*, 2001.
12. J. Gough and D. Corney. Evaluating the Java Virtual Machine as a Target for Languages other than Java. In *Proceedings Joint Modula Languages Conference*, 2000.
13. D. Grune, H. Bal, C. Jacobs, and K. Langendoen. *Modern Compiler Design.* Wiley, 2001.
14. J. C. Hardwick and J. Sipelstein. Java as an Intermediate Language. Technical Report CMU-CS-96-161, Carnegie Mellon University, August 1996.
15. S. P. Jones, N. Ramsey, and F. Reig. C–: a Portable Assembly Language that Supports Garbage Collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999.
16. A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings PLDI'01*, 2001.
17. A. Krall and J. Vitek. On Extending Java. In H. Mössenböck, editor, *Joint Modular Languages Conference (JMLC'97)*, pages 321–335, Linz, 1997. Springer.
18. C. League, Z. Shao, and V. Trifonov. Representing Java Classes in a Typed Intermediate Language. In *International Conference on Functional Programming*, pages 183–196, 1999.
19. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2e).* Addison Wesley, 1999.
20. S. Macrakis. From UNCOL to ANDF: Progress In standard Intermediate Languages. Technical report, Open Software Foundation Research Institute, 1993.
21. E. Meijer and C. Szyperski. What's in a name: .NEt as a Component Framework. In *1st OOPSLA Workshop on Language Machanisms for Programming Software Components*, pages 22–28, 2001.
22. R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML.* MIT Press, 1990.
23. P. A. Nelson. A Comparison of PASCAL Intermediate Languages. *ACM SIGPLAN Notices*, 14(8):208–213, 1979.
24. M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.
25. D. L. Schleicher and R. L. Taylor. System Overview of the Application System/400. *IBM Systems Journal*, 38(2/3):398–413, 1999.
26. R. Sites, A. Chernoff, M. Kirk, and M. Marks. Binary Translation. *CACM*, 36(2):69–81, 1993.
27. G. L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, 1978.
28. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.
29. R. Tolksdorf. Programming Languages for the Java Virtual Machine. http://grunge.cs.tu-berlin.de/ tolk/vmlanguages.html.
30. TRANSMETA. The Technology behind Crusoe Processors. http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf, 2000.

112

# Syntaxgesteuerte Editoren und Baummaschinen

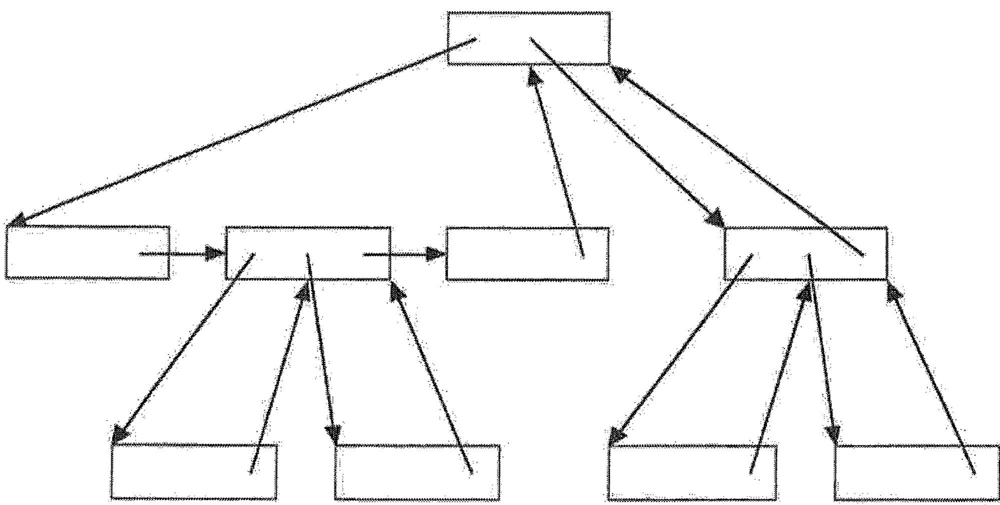
Texteingabe wird durch Schablonen gesteuert

```
PROGRAM <name>  (<namelist> );
    <declaration>
BEGIN
    <variable>  :=  <expression>
END.
```

# Anforderungen an den Zwischencode

- bijektive Abbildung zwischen Quellprogramm und Zwischencode
- speichereffizient
- laufzeiteffizient
- Rückwertsausführung muß möglich sein
- Semantik leicht erkennbar
- einfaches Einfügen und Löschen von Code

# hierarchischer Zwischencode (abstrakter Syntaxbaum)



# Besonderheiten des Zwischencodes

- Zeiger auf nächsten Knoten oder zurück zum Vaterknoten
- ermöglicht Abarbeitung des Baumes ohne Stapel
- ein Offsetfeld gibt die Position im Vaterknoten an
- Vorgängerknoten wird über Vaterknoten erreicht
- Zeiger können Sprungbefehle sein; Feld vor dem Sprungziel enthält dann den Offset zum Knotenanfang

# Karel der Roboter

Computerspiel zum Erlernen des Programmierens

- Roboter bewegt sich in einem rechtwinkeligen Straßennetz
- besitzt einen Sack mit beeper
- er kann beeper wahrnehmen, auslegen und einsammeln
- er kann sich vorwärtsbewegen und nach links drehen
- er wird in einer einfachen Programmiersprache programmiert

# Die Roboterprogrammiersprache

- keine Daten
- rekursive Unterprogramme

Sprachelemente:

- Unterprogrammdefinition
- Block-Anweisung
- WHILE-Schleife
- Zählschleife (ITERATE)
- IF-THEN-ELSE -Anweisung
- fixer Satz an Bedingungen
- Unterprogrammaufruf
- move, turnleft, pickbeeper, putbeeper, turnoff

# Knotendefinition

```
abstract class Node {
    int offset;
    Node next;

    abstract void print(int level);
    abstract int exec_step();
    abstract int back_step();
}
```

# Knotenarten

| | |
|---|---|
| ProgramNode | {DefineNode define; ExecutionNode execution}; |
| DefineNode | {Name name; Node instruction}; |
| ExecutionNode | {Node instruction}; |
| BlockNode | {Node instruction}; |
| WhileNode | {int test; Node instruction}; |
| IterateNode | {int count; Node instruction}; |
| IfThenElseNode | {int test; Node then_stmt; Node else_stmt}; |
| CallNode | {Name name}; |
| BasicInstrNode | {int instruction}; |
| Name | {String name; DefineNode define; Name next}; |

# Implementierung des Interpreters

- Befehlszähler besteht aus Knotenzeiger und Offset
- Stapel für Rücksprungadressen und ITERATE-Zähler

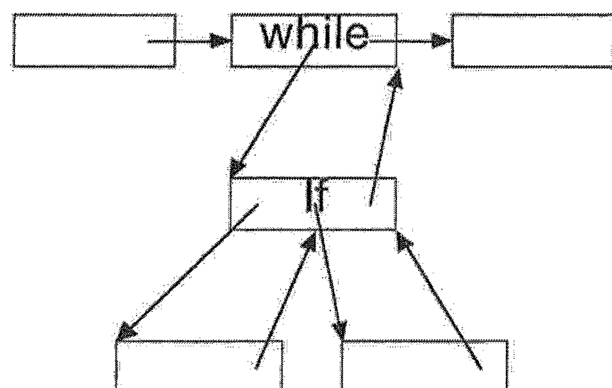
Rückwärtsausführung:

- einfache Befehle sind umkehrbar
- Vorgängerknoten wird über Vaterknoten erreicht

# Rückwärtsausführung von Schleifen und Verzweigungen

Bedingungen werden nach der Befehlsausführung in einem Stapel gespeichert

vorwärts ⟶
FTTFT
rückwärts ⟵



# Programmablauf des Editors

- Initialisierung
  - Eingabeaufforderung
  - Einlesen des Befehls
  - Analyse und Auswahl des Befehls
  - Ausführen des Befehls
  - Ausgabe des Programmtextes
  - solange Name oder Nummer
    - Einlesen von Name oder Nummer
    - Ausgabe des Programmtextes
- bis Befehl gleich Quit

## Analyse der Befehle erfolgt tabellengesteuert

Indizes: Zustand des Programmbumes, eingegebene Zeichen

Inhalt: Unterprogramm mit optionalen Parameter

Ausgabe des Programmtextes:

- Schablonen stehen in Tabelle (mit Zeileninformation)
- Einrücktiefe ist die Tiefe des Syntaxbaums

```
/* Karel the Robot, a computer language learning game after Pattis    */
/* Author:       Andreas Krall                                        */
/* Last Change: 96/03/28                                              */

interface Globals {

    // error and return codes

    static final int simple_instr_finished      = -1;  // must be < 0
    static final int no_error                    =  0;  // must be 0
    static final int robot_made_turnoff          =  1;
    static final int double_definition_error     =  2;
    static final int incomplete_program_error    =  3;
    static final int blocked_robot_error         =  4;
    static final int end_of_world_error          =  5;
    static final int no_beeper_error             =  6;
    static final int empty_bag_error             =  7;
    static final int missing_turnoff_error       =  8;
    static final int back_end_reached_error      =  9;
    static final int stack_overflow_error        = 10;
    static final int internal_program_error      = 11;

    // test codes and names

    static final int undef_test              =  0;       // must be 0
    static final int front_is_clear          =  1;       // must be previous + 1
    static final int left_is_clear           =  2;
    static final int right_is_clear          =  3;
    static final int next_to_a_beeper        =  4;
    static final int facing_north            =  5;       // north must be east - 1
    static final int facing_east             =  6;       // east must be south - 1
    static final int facing_south            =  7;       // south must be west - 1
    static final int facing_west             =  8;       // west must be south + 1
    static final int any_beepers_in_bag      =  9;
    static final int front_is_blocked        = 10;
    static final int left_is_blocked         = 11;
    static final int right_is_blocked        = 12;
    static final int not_next_to_a_beeper    = 13;
    static final int not_facing_north        = 14;
    static final int not_facing_east         = 15;
    static final int not_facing_south        = 16;
    static final int not_facing_west         = 17;
    static final int no_beepers_in_bag       = 18;

    static final String test_names[] = {
        "<test>",
        "front_is_clear",
        "left_is_clear",
        "right_is_clear",
        "next_to_a_beeper",
        "facing_north",
        "facing_east",
        "facing_south",
        "facing_west",
        "any_beepers_in_bag",
        "front_is_blocked",
        "left_is_blocked",
        "right_is_blocked",
        "not_next_to_a_beeper",
        "not_facing_north",
        "not_facing_east",
        "not_facing_south",
        "not_facing_west",
        "no_beepers_in_bag"
    };

    // basic instruction codes and names

    static final int undef_instr     = 0;     // must be 0
```



```
    static final int move_instr         = 1;     // must be previous + 1
    static final int turnleft_instr     = 2;
    static final int putbeeper_instr    = 3;
    static final int pickbeeper_instr   = 4;
    static final int turnoff_instr      = 5;

    static final String instruction_names[] = {
        "<instr>", "move", "turnleft", "putbeeper", "pickbeeper", "turnoff"
    };

    // node codes and node element description codes

    static final int is_undef           =  0;   // must be less than is_number
    static final int program_node       =  1;   // must be less than is_number
    static final int define_node        =  2;   // must be less than is_number
    static final int execution_node     =  3;   // must be less than is_number
    static final int block_node         =  4;   // must be less than is_number
    static final int while_node         =  5;   // must be less than is_number
    static final int iterate_node       =  6;   // must be less than is_number
    static final int if_then_else_node  =  7;   // must be less than is_number
    static final int call_node          =  8;   // must be less than is_number
    static final int basic_instr_node   =  9;   // must be less than is_number
    static final int is_program         = 10;   // must be less than is_number
    static final int is_execution       = 11;   // must be less than is_number
    static final int is_def_list        = 12;   // must be less than is_number
    static final int is_stmt            = 13;   // must be less than is_number
    static final int is_stmt_list       = 14;   // must be less than is_number
    static final int is_number          = 15;
    static final int is_test            = 16;   // must be greater than is_number
    static final int is_name            = 17;   // must be greater than is_number
    static final int is_instr           = 18;   // must be greater than is_number
}


final class KarelTheRobot implements Globals{

    static int offset = 0;                          // offset of current instruction
    static Node instruction = null;                 // current instruction

    // left searches to the next left position

    static void left() {
        Node instr = instruction;

        if (offset == 0) {                              // find previous node in list
            while (instruction.offset == 0)             // find last node in list
                instruction = instruction.next;
            offset = instruction.offset - 1;            // set parent offset
            instruction = instruction.next;             // set parent node
            if (instruction.get_node_at_pos(offset) == instr)
                offset--;                               // original node is list head
            else {
                instruction = instruction.get_node_at_pos(offset);
                while (instruction.next != instr) // find previous node in list
                    instruction = instruction.next;
                offset = instruction.length();    // offset of last element
            }
        } else
            offset--;
        if (offset == 0)
            return;
        // go down the tree to the rightmost node of the tree
        while ((instruction.description(offset) < is_number) &&
               (instruction.get_node_at_pos(offset) != null)) {
            // go down the tree
            instruction = instruction.get_node_at_pos(offset);
            while (instruction.offset == 0)             // find last node in list
                instruction = instruction.next;
            offset = instruction.length();              // offset of last element
```

```java
    }
}

//right searches to next right position

static void right() {
    offset++;
    while (offset >= instruction.length()) {  // while behind last element
        offset = instruction.offset;          // set parents offset and
        instruction = instruction.next;       // go up to parent node
    }
    if ((offset != 0) &&                      // go down to child node
        (instruction.description(offset) < is_number) &&
        (instruction.get_node_at_pos(offset) != null)) {
        instruction = instruction.get_node_at_pos(offset);
        offset = 0;                           // first position in child node
    }
}

// insert_node insert a node at the current position

static void insert_node(int node_code) {
    Node node, help;

    node = null;
    switch (node_code) {
        case program_node:
            offset = 0;
            instruction = new ProgramNode();
            return;
        case define_node:
            node = new DefineNode();
            break;
        case execution_node:
            node = new ExecutionNode();
            break;
        case block_node:
            node = new BlockNode();
            break;
        case while_node:
            node = new WhileNode();
            break;
        case iterate_node:
            node = new IterateNode();
            break;
        case if_then_else_node:
            node = new IfThenElseNode();
            break;
        case call_node:
            node = new CallNode();
            break;
        case basic_instr_node:
            node = new BasicInstrNode();
            break;
    }
    if (instruction.description(offset) == is_undef) {
        node.offset = instruction.offset;                    // middle of list
        node.next = instruction.next;
        instruction.offset = 0;
        instruction.next = node;
    } else if ((help = instruction.get_node_at_pos(offset)) == null) {
        instruction.put_object_at_pos(offset, node);         // empty
        node.offset = offset + 1;
        node.next = instruction;
    } else {
        instruction.put_object_at_pos(offset, node);         // head of list
        node.offset = help.offset;
        node.next = help.next;
    }
```





```java
        offset = 0;
        instruction = node;
    }

    static public void print(int level, String text) {
        while (--level >= 0)
            System.out.print("   ");
        System.out.print(text);
    }

    static public void println(int level, String text) {
        while (--level >= 0)
            System.out.print("   ");
        System.out.println(text);
    }

    static public void print_instr(int level, Node instr) {

        if (instr == null)
            KarelTheRobot.println(level + 1, "<instruction>");
        else
            while (instr != null) {
                instr.print(level + 1);
                if (instr.offset != 0)
                    return;
                instr = instr.next;
            }
    }

    static public void main(String args[])
    throws java.io.IOException {
        ProgramNode program;
        Node n1, n2, n3;
        IfThenElseNode if_stmt;
        int error;
        int ch;

        n1 = new BasicInstrNode(turnleft_instr);
        n1.next = new BasicInstrNode(turnleft_instr);
        n3 = n1.next.next = new BasicInstrNode(turnleft_instr);
        n2 = new BlockNode();
        ((BlockNode) n2).instruction = n1;
        n3.next = n2;
        n3.offset = 2;
        n1 = new DefineNode();
        ((DefineNode) n1).name =
                    Name.enter_name("turnright", (DefineNode) n1);
        ((DefineNode) n1).instruction = n2;
        program = new ProgramNode();
        program.define = (DefineNode) n1;
        n2.offset = 3;
        n2.next = program.execution;
        program.execution.instruction = if_stmt = new IfThenElseNode();
        if_stmt.test = 3;
        if_stmt.next = new BasicInstrNode(turnoff_instr);
        if_stmt.next.offset = 2;
        if_stmt.next.next = program.execution;
        n1 = if_stmt.then_stmt = new CallNode();
        n1.offset = 3;
        n1.next = if_stmt;
        ((CallNode) n1).definition = program.define.name;
        n1 = if_stmt.else_stmt = new IterateNode();
        n1.offset = 4;
        n1.next = if_stmt;
        ((IterateNode) n1).count = 3;

        program.print(0);

        offset = 0;
```

65

120

```
        instruction = program;

        BackBuffer.reset();
        DefineNode.reset();
        IterateNode.reset();

        ch = 'c';
        while ((error = instruction.exec_step()) <= 0) {
            //ch = System.in.read();
            if (ch != 'c')
                break;
        }

        System.out.println("Ergebnis des Programmes: " + error);
    }
}


final class KarelsWorld implements Globals{

    // world's bit 0 = north wall, bit 1 = east wall, bit 2 .. 15 = beeper count

    private static short world[][] = new short[100][100];
    private static final int north_wall = 1;
    private static final int east_wall = 2;

    private static int x = 0;                      // Karels x position
    private static int y = 0;                      // Karels y position
    private static int facing = facing_north;      // Karels facing
    private static int bag = 0;                    // number of beepers in Karels bag

    static void set_karel(int x, int y, int facing) {
        if (x < 0)
            x = 0;
        if (x >= world.length)
            x = world.length - 1;
        KarelsWorld.x = x;
        if (y < 0)
            y = 0;
        if (y >= world[0].length)
            y = world[0].length - 1;
        KarelsWorld.y = y;
        KarelsWorld.facing = facing_north;
        if ((facing == facing_north) ||
            (facing == facing_east) ||
            (facing == facing_south) ||
            (facing == facing_west))
            KarelsWorld.facing = facing;
    }

    static void fill_bag(int count) {
        if (count < 0)
            count = 0;
        bag = count;
    }

    private static boolean position_out_of_bounds(int x, int y) {
        if ((x < 0) || (x >= world.length) || (y < 0) || (y >= world[0].length))
            return true;
        return false;
    }

    static void set_north_wall(int x, int y) {
        if (position_out_of_bounds(x, y))
            return;
        world[x][y] |= north_wall;
    }

    static void set_east_wall(int x, int y) {
```



```
        if (position_out_of_bounds(x, y))
            return;
        world[x][y] |= east_wall;
    }

    static void clear_north_wall(int x, int y) {
        if (position_out_of_bounds(x, y))
            return;
        world[x][y] &= ~north_wall;
    }

    static void clear_east_wall(int x, int y) {
        if (position_out_of_bounds(x, y))
            return;
        world[x][y] &= ~east_wall;
    }

    static boolean test(int test) {
        switch (test) {
            case front_is_clear:
                switch (facing){
                    case facing_north:
                        return ((world[x][y] & north_wall) == 0);
                    case facing_east:
                        return ((world[x][y] & east_wall) == 0);
                    case facing_south:
                        if (y == 0)
                            return false;
                        return ((world[x][y-1] & north_wall) == 0);
                    case facing_west:
                        if (x == 0)
                            return false;
                        return ((world[x-1][y] & east_wall) == 0);
                }
                break;
            case left_is_clear:
                switch (facing){
                    case facing_north:
                        if (x == 0)
                            return false;
                        return ((world[x-1][y] & east_wall) == 0);
                    case facing_east:
                        return ((world[x][y] & north_wall) == 0);
                    case facing_south:
                        return ((world[x][y] & east_wall) == 0);
                    case facing_west:
                        if (y == 0)
                            return false;
                        return ((world[x][y-1] & north_wall) == 0);
                }
                break;
            case right_is_clear:
                switch (facing){
                    case facing_north:
                        return ((world[x][y] & east_wall) == 0);
                    case facing_east:
                        if (y == 0)
                            return false;
                        return ((world[x][y-1] & north_wall) == 0);
                    case facing_south:
                        if (x == 0)
                            return false;
                        return ((world[x-1][y ] & east_wall) == 0);
                    case facing_west:
                        return ((world[x][y] & north_wall) == 0);
                }
                break;
            case next_to_a_beeper:
                return ((world[x][y] >>> 2) > 0);
```

```
            case facing_north:
                return (facing == facing_north);
            case facing_east:
                return (facing == facing_east);
            case facing_south:
                return (facing == facing_south);
            case facing_west:
                return (facing == facing_west);
            case any_beepers_in_bag:
                return (bag > 0);
            case front_is_blocked:
                return ! test(front_is_clear);
            case left_is_blocked:
                return ! test(left_is_clear);
            case right_is_blocked:
                return ! test(right_is_clear);
            case not_next_to_a_beeper:
                return ((world[x][y] >>> 2) <= 0);
            case not_facing_north:
                return (facing != facing_north);
            case not_facing_east:
                return (facing != facing_east);
            case not_facing_south:
                return (facing != facing_south);
            case not_facing_west:
                return (facing != facing_west);
            case no_beepers_in_bag:
                return (bag <= 0);
        }
        return false;
    }

    static int exec_instr(int instr) {
        switch (instr) {
            case undef_instr:
                return incomplete_program_error;
            case move_instr:
                System.out.println("move");
                if (test(front_is_clear))
                    switch (facing) {
                        case facing_north:
                            if (y >= world[0].length)
                                return end_of_world_error;
                            y++;
                            return 0;
                        case facing_south:
                            if (y <= 0)
                                return blocked_robot_error;
                            y--;
                            return 0;
                        case facing_east:
                            if (x >= world.length)
                                return end_of_world_error;
                            x++;
                            return 0;
                        case facing_west:
                            if (x <= 0)
                                return blocked_robot_error;
                            x--;
                            return 0;
                        default:
                            return internal_program_error;
                    }
                else
                    return blocked_robot_error;
            case turnleft_instr:
                System.out.println("turnleft");
                if (--facing < facing_north)
                    facing = facing_west;
```



```
                return 0;
            case putbeeper_instr:
                System.out.println("putbeeper");
                if (bag <= 0)
                    return empty_bag_error;
                bag--;
                world[x][y] += 4;
                return 0;
            case pickbeeper_instr:
                System.out.println("pickbeeper");
                if ((world[x][y] >>> 2) <= 0)
                    return no_beeper_error;
                world[x][y] -= 4;
                bag++;
                return 0;
            case turnoff_instr:
                System.out.println("turnoff");
                return robot_made_turnoff;
        }
        return 0;
    }
}


class BackBuffer {
    static long buffer = 0;
    static int elements = 0;

    static void reset() {
        elements = 0;
    }

    static void push(boolean val) {
        if (elements < 64)
            elements++;
        buffer = buffer << 1;
        if (val)
            buffer += 1;
    }

    static boolean pop() {
        boolean retval;

        retval = (buffer & 1) != 0;
        if (elements > 0)
            elements--;
        buffer = buffer >>> 1;
        return retval;
    }
}


abstract class Node implements Globals{
    int  offset;
    Node next;

    Node() {
        offset = 0;
        next = null;
    }

    int length() {
        return 1;
    }

    abstract int description(int pos);

    int get_int_at_pos(int pos) {
        return 0;
```

122

```
    }

    Name get_name_at_pos(int pos){
        return null;
    }

    Node get_node_at_pos(int pos){
        return null;
    }

    void put_object_at_pos(int pos, int val) {
    }

    void put_object_at_pos(int pos, Name name) {
    }

    void put_object_at_pos(int pos, Node node) {
    }

    abstract void print(int level);
    abstract int exec_step();
}

final class ProgramNode extends Node implements Globals {

    static final int description[] =
        {program_node, is_def_list, is_execution, is_undef, is_undef};

    DefineNode    define;
    ExecutionNode execution;

    ProgramNode() {
        define = null;
        execution = new ExecutionNode();
        execution.next = this;
        execution.offset = 3;
    }

    int length() {
        return 2;
    }

    int description(int pos) {
        return description[pos];
    }

    Node get_node_at_pos(int pos){
        if (pos == 1)
            return define;
        else if (pos == 2)
            return execution;
        return null;
    }

    void put_object_at_pos(int pos, Node node) {
        if (pos == 1)
            define = (DefineNode) node;
        else if (pos == 2)
            execution = (ExecutionNode) node;
    }

    void print(int level) {
        KarelTheRobot.println(level, "BEGINNING-OF-PROGRAM");
        if (define == null)
            KarelTheRobot.println(level + 1, "<definition>");
        else
            define.print(level + 1);
        execution.print(level + 1);
```





```
        KarelTheRobot.println(level, "END-OF-PROGRAM");
    }

    int exec_step() {
        if (KarelTheRobot.offset != 0)
            return missing_turnoff_error;
        if (execution == null)
            return incomplete_program_error;
        KarelTheRobot.instruction = execution;
        return 0;
    }
}

final class DefineNode extends Node implements Globals {

    static final int description[] =
        {define_node, is_name, is_stmt, is_undef, is_undef};

    Name name;
    Node instruction;

    static CallNode stack[] = new CallNode[1024];
    static int top = 0;

    DefineNode() {
        name = null;
        instruction = null;
    }

    static void reset() {
        top = 0;
    }

    int length() {
        return 2; }
    }

    int description(int pos) {
        return description[pos];
    }

    Name get_name_at_pos(int pos){
        if (pos == 1)
            return name;
        return null;
    }

    void put_object_at_pos(int pos, Name name) {
        if (pos == 1)
            this.name = name;
    }

    Node get_node_at_pos(int pos){
        if (pos == 2)
            return instruction;
        return null;
    }

    void put_object_at_pos(int pos, Node node) {
        if (pos == 2)
            instruction = node;
    }

    void print(int level) {
        KarelTheRobot.print(level, "DEFINE-NEW-INSTRUCTION ");
        if (name == null)
            KarelTheRobot.print(0, "<name>");
        else
```

A23

```
            KarelTheRobot.print(0, name.get_name());
        KarelTheRobot.println(0, " AS");
        instruction.print(level + 1);
    }

    static int push(CallNode caller) {
        top++;
        if (top >= stack.length)
            return stack_overflow_error;
        stack[top] = caller;
        KarelTheRobot.instruction = caller.definition.define.instruction;
        return 0;
    }

    int exec_step() {
        if (KarelTheRobot.offset == 0)
            return internal_program_error;
        KarelTheRobot.offset = stack[top].offset;
        KarelTheRobot.instruction = stack[top].next;
        top--;
        return 0;
    }
}

final class Name implements Globals{

    private static Name name_list = null;

    String      name;
    DefineNode  define;
    private Name next;

    Name(String name, DefineNode define) {
        this.name = name;
        this.define = define;
        this.next = name_list;
        name_list = this;
    }

    public String get_name() {
        return this.name;
    }

    public static Name enter_name(String name, DefineNode definition) {
        Name nlist = name_list;
        while (nlist != null) {
            if (nlist.name.equals(name))
                if (nlist.define == null) {
                    nlist.define = definition;
                    return nlist;
                } else
                    return null;
            nlist = nlist.next;
        }
        return new Name(name, definition);
    }

    public static DefineNode find_name(String name) {
        Name nlist = name_list;
        while (nlist != null) {
            if (nlist.name.equals(name))
                return nlist.define;
            nlist = nlist.next;
        }
        new Name(name, null);
        return null;
    }
}
```



```
final class ExecutionNode extends Node implements Globals {

    static final int description[] =
        {execution_node, is_stmt_list, is_undef, is_undef, is_undef};

    Node instruction;

    ExecutionNode() {
        instruction = null;
    }

    int description(int pos) {
        return description[pos];
    }

    Node get_node_at_pos(int pos){
        if (pos == 1)
            return instruction;
        return null;
    }

    void put_object_at_pos(int pos, Node node) {
        if (pos == 1)
            instruction = node;
    }

    void print(int level) {
        KarelTheRobot.println(level, "BEGINNING-OF-EXECUTION");
        KarelTheRobot.print_instr(level, instruction);
        KarelTheRobot.println(level, "END-OF-EXECUTION");
    }

    int exec_step() {
        if (KarelTheRobot.offset == 0) {
            if (instruction == null)
                return incomplete_program_error;
            KarelTheRobot.instruction = instruction;
        } else {
            KarelTheRobot.offset = 0;
            KarelTheRobot.instruction = instruction;
        }
        return 0;
    }
}

final class BlockNode extends Node implements Globals {

    static final int description[] =
        {block_node, is_stmt_list, is_undef, is_undef, is_undef};

    Node instruction;

    BlockNode() {
        instruction = null;
    }

    int description(int pos) {
        return description[pos];
    }

    Node get_node_at_pos(int pos){
        if (pos == 1)
            return instruction;
        return null;
    }
```

```
    void put_object_at_pos(int pos, Node node) {
        if (pos == 1)
            instruction = node;
    }

    void print(int level) {
        KarelTheRobot.println(level, "BEGIN");
        KarelTheRobot.print_instr(level, instruction);
        KarelTheRobot.println(level, "END");
    }

    int exec_step() {
        if (KarelTheRobot.offset == 0) {
            if (instruction == null)
                return incomplete_program_error;
            KarelTheRobot.instruction = instruction;
        } else {
            KarelTheRobot.offset = 0;
            KarelTheRobot.instruction = instruction;
        }
        return 0;
    }
}


final class WhileNode extends Node implements Globals {

    static final int description[] =
        {while_node, is_test, is_stmt, is_undef, is_undef};

    int  test;
    Node instruction;

    WhileNode() {
        test = undef_test;
        instruction = null;
    }

    int length() {
        return 2;
    }

    int description(int pos) {
        return description[pos];
    }

    int get_int_at_pos(int pos){
        if (pos == 1)
            return test;
        return 0;
    }

    void put_object_at_pos(int pos, int val) {
        if (pos == 1)
            test = val;
    }

    Node get_node_at_pos(int pos){
        if (pos == 2)
            return instruction;
        return null;
    }

    void put_object_at_pos(int pos, Node node) {
        if (pos == 2)
            instruction = node;
    }

    void print(int level) {
```



```
        KarelTheRobot.println(level, "WHILE " + test_names[test] + " DO");
        KarelTheRobot.print_instr(level, instruction);
    }

    int exec_step() {
        if (test == 0)
            return incomplete_program_error;
        if (KarelTheRobot.offset == 0)
            BackBuffer.push(true);
        else
            BackBuffer.push(false);
        KarelTheRobot.offset = 0;
        if (KarelsWorld.test(test)) {
            if (instruction == null)
                return incomplete_program_error;
            KarelTheRobot.instruction = instruction;
        } else {
            KarelTheRobot.offset = offset;
            KarelTheRobot.instruction = next;
        }
        return 0;
    }
}


final class IterateNode extends Node implements Globals {

    static final int description[] =
        {iterate_node, is_number, is_stmt, is_undef, is_undef};

    int  count;
    Node instruction;

    static int stack[] = new int[1024];
    static int top = 0;

    IterateNode() {
        count = 0;
        instruction = null;
    }

    static void reset() {
        top = 0;
    }

    int length() {
        return 2;
    }

    int description(int pos) {
        return description[pos];
    }

    int get_int_at_pos(int pos){
        if (pos == 1)
            return count;
        return 0;
    }

    void put_object_at_pos(int pos, int val) {
        if (pos == 1)
            count = val;
    }

    Node get_node_at_pos(int pos){
        if (pos == 2)
            return instruction;
        return null;
    }
```

```java
    void put_object_at_pos(int pos, Node node) {
        if (pos == 2)
            instruction = node;
    }

    void print(int level) {
        if (count == 0)
            KarelTheRobot.println(level, "ITERATE <number> TIMES");
        else
            KarelTheRobot.println(level, "ITERATE " + count + " TIMES");
        KarelTheRobot.print_instr(level, instruction);
    }

    int exec_step() {
        if (KarelTheRobot.offset == 0) {
            if ((count == 0) || (instruction == null))
                return incomplete_program_error;
            top++;
            if (top >= stack.length)
                return stack_overflow_error;
            stack[top] = count;
            KarelTheRobot.instruction = instruction;
        } else {
            if (--stack[top] > 0) {
                KarelTheRobot.offset = 0;
                KarelTheRobot.instruction = instruction;
            } else {
                top--;
                KarelTheRobot.offset = offset;
                KarelTheRobot.instruction = next;
            }
        }
        return 0;
    }
}


class IfThenElseNode extends Node implements Globals {

    static final int description[] =
        {if_then_else_node, is_test, is_stmt, is_stmt, is_undef};

    int  test;
    Node then_stmt;
    Node else_stmt;

    IfThenElseNode() {
        test = undef_test;
        then_stmt = null;
        else_stmt = null;
    }

    int length() {
        return 3;
    }

    int description(int pos) {
        return description[pos];
    }

    int get_int_at_pos(int pos){
        if (pos == 1)
            return test;
        return 0;
    }

    void put_object_at_pos(int pos, int val) {
        if (pos == 1)
```



```java
            test = val;
    }

    Node get_node_at_pos(int pos){
        if (pos == 2)
            return then_stmt;
        else if (pos == 3)
            return else_stmt;
        return null;
    }

    void put_object_at_pos(int pos, Node node) {
        if (pos == 2)
            then_stmt = node;
        else if (pos == 3)
            else_stmt = node;
    }

    void print(int level) {
        KarelTheRobot.println(level, "IF " + test_names[test]);
        KarelTheRobot.println(level, "THEN");
        KarelTheRobot.print_instr(level, then_stmt);
        KarelTheRobot.println(level, "ELSE");
        KarelTheRobot.print_instr(level, else_stmt);
    }

    int exec_step() {
        if (test == 0)
            return incomplete_program_error;
        if (KarelTheRobot.offset == 0) {
            if (KarelsWorld.test(test)) {
                if (then_stmt == null)
                    return incomplete_program_error;
                KarelTheRobot.instruction = then_stmt;
            } else {
                if (else_stmt == null)
                    return incomplete_program_error;
                KarelTheRobot.instruction = else_stmt;
            }
        } else {
            if (KarelTheRobot.offset == 4)
                BackBuffer.push(true);
            else
                BackBuffer.push(false);
            KarelTheRobot.offset = offset;
            KarelTheRobot.instruction = next;
        }
        return 0;
    }
}


final class CallNode extends Node implements Globals {

    static final int description[] =
        {call_node, is_name, is_undef, is_undef, is_undef};

    Name definition;

    CallNode() {
        definition = null;
    }

    int description(int pos) {
        return description[pos];
    }

    Name get_name_at_pos(int pos){
        if (pos == 1)
```

126

# Syntaxgesteuerte Editoren, Interpreter und Compiler

Mentor: Arbeiten an diesem Projekt seit 1975, Struktureditor, Mental = Editor-definitionssprache, Softwareentwicklungs-umgebung, INRIA, Frankreich

Gandalf (ALOE): A Language Oriented Editor Editorgenerator S-E-Umgebung, CMU

COPS: Cornel Program Synthesizer, Editorgenerator mit Semantiküberprüfung (attributierte Grammatik), Interpreter und Debugger für Syntaxbaum, Cornell Univ.

PSG: Programm System Generator, TH Darmstadt

```
            return definition;
        return null;
    }

    void put_object_at_pos(int pos, Name name) {
        if (pos == 1)
            this.definition = name;
    }

    void print(int level) {
        KarelTheRobot.println(level, definition.get_name());
    }

    int exec_step() {
        if (KarelTheRobot.offset != 0)
            return internal_program_error;
        if ((definition == null) || (definition.define.instruction == null))
            return incomplete_program_error;
        return definition.define.push(this);
    }
}


final class BasicInstrNode extends Node implements Globals {

    static final int description[] =
        {basic_instr_node, is_instr, is_undef, is_undef, is_undef};

    int instruction;

    BasicInstrNode() {
        instruction = undef_instr;
    }

    BasicInstrNode(int instr) {
        instruction = instr;
    }

    int description(int pos) {
        return description[pos];
    }

    int get_int_at_pos(int pos){
        if (pos == 1)
            return instruction;
        return 0;
    }

    void put_object_at_pos(int pos, int val) {
        if (pos == 1)
            instruction = val;
    }

    void print(int level) {
        KarelTheRobot.println(level, instruction_names[instruction]);
    }

    int exec_step() {
        int retval;

        retval = KarelsWorld.exec_instr(instruction);
        KarelTheRobot.offset = offset;
        KarelTheRobot.instruction = next;
        return retval;
    }
}
```



427

power but retain the disciplined viewpoint of the rest of the system.

A final example illustrates an awkwardness arising not from the structural constraints of the Synthesizer, but from the textual constraints of a language whose concrete syntax was defined to be unambiguous for parsers. Inserting the template

```
IF ( condition )
   THEN statement
```

into

```
IF ( condition )
   THEN [S]tatement
   ELSE PUT LIST ( 'whose else am i?' );
```

leads to an inconsistency between the explicitly derived structure (an IF–THEN within an IF–THEN–ELSE) and the structure implied by the parser-oriented concrete syntax (an IF–THEN–ELSE within an IF–THEN). Although tempted to adopt the derived interpretation (because prettyprinting easily distinguishes one interpretation from the other), we elected, instead, to maintain compatibility with PL/I. Therefore, we prevent such an insertion and require that the user provide a compound statement explicitly.

There are many possible alternative designs, among them the following four: a) the compound statement could be inserted automatically when necessary; b) a compound statement could be displayed automatically when necessary; c) the IF–THEN–ELSE template could be defined as

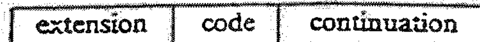
```
IF ( condition )
   THEN DO; {statement} END;
   ELSE DO; {statement} END;
```

d) the IF–THEN template could be eliminated thereby requiring that every conditional statement have an ELSE-clause. In this final case, the display of an empty ELSE clause could be suppressed unless necessary for disambiguation.

## VI. Implementation

### A. File Trees

Synthesizer files are represented internally as executable derivation trees. Each template or phrase is represented in this tree by a separate node. The pointers connecting nodes are, in fact, goto instructions for the interpreter; the null pointer is a halt instruction. Nodes are variable length; each is composed of three sections:

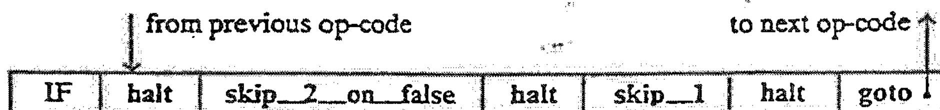



The *extension* identifies the node type and contains any other information needed to generate the display of the node but not necessary to execute it. The *code* section contains interpretable op-codes for executing the node. The *entry point* of the node is the first byte of the code section. The *continuation* contains a goto linking this node to the next op-code to be executed. The target of this goto is either the entry point of a sibling node or an interior op-code of a parent node.

For example, the template

```
IF ( condition )
   THEN statement
   ELSE statement
```

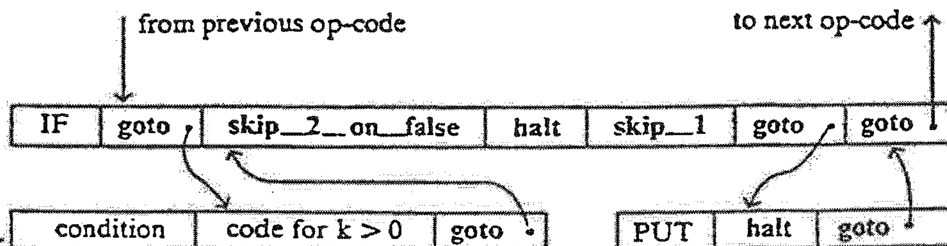
has the internal representation given below.




This node is tagged in the extension as an IF-node. It contains op-codes that implement the proper control flow and three halt instructions that represent the unexpanded placeholders. When the template has been expanded to

```
IF ( k > 0 )
   THEN statement
   ELSE PUT LIST ( list-of-expressions );
```

a link to Polish postfix code for the phrase $k > 0$ replaces the first halt op-code, and a link to the node for the PUT-statement replaces the third halt op-code. A halt instruction remains for the other *statement* placeholder:



The interpreter is classical: it executes straight line code and goto instructions. It is completely blind to the structure of the tree and requires neither recursion nor a stack to execute a file tree. Access to variables and procedure definitions is through a symbol table.

The editor walks the tree using the same goto pointers as the interpreter. Each cursor position designates one of the nodes of the tree. Cursor motion is defined with respect to a preorder traversal. There are no backward pointers; thus, backward cursor motion is implemented internally by going all the way around.

### B. Declarations

As demonstrated in Sec. II.C, declarations present a special problem: modifying a declaration can simultaneously introduce errors and correct errors at other locations in the program. Internally, information about identifiers is stored in a symbol table. When a declaration is modified, the Synthesizer discards the old symbol table and traverses the tree in preorder reparsing and redoing the semantics of every phrase. Phrases with errors are marked as invalid and are printed in the highlighted font when the screen is redrawn. Because the allocation of variables within an activation record is recomputed in the process of reconstructing the symbol table, access to the variables of a suspended activation record is lost in the process. Therefore, execution cannot be resumed after such modifications.

### C. Displaying the Tree

The print representation of a file is generated from the tree; a text representation is not saved. The external representation of each kind of template is stored in a table. The entries of this table alternate between terminal strings and placeholder-descriptors. For example, the IF-template is encoded as:

"IF ("
*condition-descriptor*
") \{\nTHEN"
*statement-1-descriptor*
"ELSE"
*statement-2-descriptor*
"\}\r"

The placeholder-descriptors identify the placeholders and their positions within the code section of an internal node. The terminal strings contain key words, punctuation marks, and formatting control characters that are interpreted on output. For example,

\{ means move left-margin right one unit,
\n means line-feed, carriage-return to current left-margin,
\} means move left-margin left one unit,
\r means carriage-return to current left-margin.

The print routine traverses the tree in preorder, simultaneously keeping track of position within the external representation of the appropriate template. Each terminal string encountered is printed and its formatting commands obeyed. Each phrase is translated from postfix to infix for display. (The parentheses of a phrase are saved in the extension of the node encoded one bit per operator.)

As the tree is traversed for display, a table mapping internal node addresses to external screen coordinates is updated. This table is used both for cursor motion in the editor, and at runtime for the trace feature.

### D. Implementation of Debugging Features

The tracing, pacing, and single-step features are implemented by taking appropriate action on the interpretation of each goto leading to a new node.

When tracing, each goto uses the map from internal node addresses to screen coordinates to determine the new cursor position. If the map is not defined for a given target node, then the cursor lies outside the window and the program is redrawn with the new cursor position centered in the window. Traced programs are never permitted to run any faster than one cursor update per refresh of the video screen in order to avoid stroboscopic effects such as loops that appear to run backwards. When pacing, the interpreter waits appropriately at each goto before continuing execution. When stepping, the interpreter waits for a resume command before continuing.

The variable-monitoring feature is implemented in a straightforward manner: a table mapping identifiers to screen positions is maintained. Assignment to a monitored variable is detected by the interpreter whereupon the appropriate position is updated on the screen.

Reverse execution also has a straightforward implementation: the forward execution interpreter maintains a history file of the flow of control and the values destroyed by assignments to variables. The reverse execution interpreter restores values and updates the screen to give the illusion of the program executing backwards.

## VII. The Synthesizer Generator

Continuing research and development of the Synthesizer will increase its power, versatility, and range of application complementing the unique syntax-directed mechanisms the environment already provides. For example, global data flow analysis techniques will be used to answer queries about static program structure, as in [18]. The video display can be used to express static relationships between components of a program; the multiple fonts of a terminal can be exploited to highlight regions of interest. For example, the programmer might request the highlighting of all uses or all assignments to a variable $X$. Alternatively, the analysis can be keyed to the present location of the editing cursor. For example, the programmer might request the highlighting of all assignments to $X$ that can account for its value at the present cursor location, or all possible uses of $X$ that can

be reached from the present cursor location.

To facilitate such further development, we are implementing a language-independent system for generating Synthesizer-like systems from a grammatical specification of a given programming language. An attribute grammar will be used to define the syntax, display format, and semantics of each template and phrase. In our application, where program units are inserted and deleted in arbitrary order, semantic analysis must be both incremental and reversible. For this purpose, attribute grammars have the advantage of expressing semantics and context-sensitive constraints applicatively and on a modular basis; the arguments to each semantic function are imported explicitly from neighboring nodes in the derivation tree.

Because propagation of semantic information through the tree is implicit in the formalism, an incremental attribute evaluator can update the appropriate attribute values in conjunction with each editing operation. In particular, because the attribute dependencies are known, the evaluator can delete semantic information automatically when program units are deleted; a separate mechanism to undo semantics is not needed. We have described one such incremental attribute evaluator in [8]; more recently, we have developed an optimal-time incremental evaluator that runs in time proportional to the number of attribute values that actually must be changed [21].

*Acknowledgments.* Many people have participated in the development of the Synthesizer. We are deeply indebted to A. Demers for many stimulating discussions and for writing the LSI-11 operating system kernel; his insights and assistance have been invaluable. We are also extremely grateful for the generous help of J. Archer, R. Conway, M. Fingerhut, D. Gries, C. Hauser, S. Horwitz, D. Jacobs, R. Johnson, D. Krafft, S. Mahaney, and R. Olsson.

Received 5/80; revised and accepted 4/81

References

1. Alberga, C.N., Brown, A.L., Leeman, G.B., Mikelsons, M., and Wegman, M.N. A program development tool. Conference Record of the 8th Ann. Symp. on Principles of Programming Languages, Williamsburg, VA, Jan., 1981, 92–104.
2. Archer, J., Conway, R., Shore, A., and Silver, L. The CORE user interface. Tech. Report No. TR80-437, Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY, Sept. 1980.
3. Balzer, R.M., EXDAMS–EXtendable Debugging and Monitoring System, AFIPS Proc. V. 34 (SJCC 1969), 567–580.
4. Constable, R., and O'Donnell, M.J. *A Programming Logic.* Winthrop, Cambridge, MA, 1978.
5. Conway, R. and Constable, R. PL/CS–A disciplined subset of PL/I. Tech. Rept No. 76-293, Dept. of Comptr. Sci., Cornell 1976.
6. Conway, R. *Primer on Disciplined Programming Using PL/CS.* Winthrop, Cambridge, MA, 1978.
7. Conway, R. and Gries, D. *An introduction to programming—a structured approach using* PL/I *and* PL/C. Winthrop, Cambridge, MA, 1979, 135–137.
8. Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. Conference Record of the 8th Ann. Symp. on Principles of Programming Languages, Williamsburg, VA, Jan. 1981.
9. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J.J. A structure-oriented program editor. Tech. Rept, IRIA–LABORIA, France 1975.
10. Engelbart, D.C. and English, W.K. A research center for augmenting human intellect. AFIPS Proc. V. 33 (FJCC, 1968).
11. Feiler, P.H. and Medina-Mora, R., An incremental programming environment. Dept. of Comptr. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, April 1980.
12. Hansen, W. Creation of hierarchic text with a computer display. Ph.D. Thesis, Comptr. Sci. Dept, Stanford University, Stanford, CA, June 1971.
13. Habermann, A.N. An overview of the Gandalf project. Comptr. Sci. Res. Rev. 1978–79, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.
14. Hodgson, L.I., and Porter, M. BIDOPS: A bi-directional programming system. Dept. of Comptr. Sci., Univ. of New England, Armidale, N.S.W., Australia, 1980.
15. Joy, B. Ex Reference manual. Dept. of Electrical Eng. and Comptr. Sci., Univ. California, Berkeley, CA, 1977.
16. Kurtz, T.E. BASIC. SIGPLAN Notices, Aug. 1978.
17. Lewis, J.W. and Porges, D.F. ALBE/P: a language-based editor for Pascal. Dept. of Comptr. Sci., Yale Univ., New Haven, CT.
18. Masinter, L.M. Global program analysis in an interactive environment. Xerox PARC Report SSL-80-1, Jan. 1980.
19. Mikelsons, M. and Wegman, M.N. PDEIL: The PL1L program development environment principles of operation. Res. Rept RC8513, IBM, Thomas J. Watson Research Center, Yorktown Heights, NY, Nov. 1980.
20. Pine, J.H. and Schweppe, E.J. A Fortran language anticipation and prompting system. Proc. ACM Nat. Conf., Atlanta, Georgia, 1973.
21. Reps, T. Optimal-time incremental semantic analysis for syntax-directed editors. Tech. Report No. 81-453, Dept. of Comptr. Sci., Cornell University, Ithaca, NY, March 1981.
22. Skinner, G. Ged user documentation. Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY,
23. Teitelbaum, T. A formal syntax for PL/CS. Tech Rept 76-281, Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY, 1976.
24. Teitelbaum, T. The Cornell Program Synthesizer: a microcomputer implementation of PL/CS. Tech. Report No. TR79-370, Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY, June 1979.
25. Teitelbaum, T. The Cornell program synthesizer: A tutorial introduction. Tech. Report No. TR79-381, Dept. Comptr. Sci., Cornell Univ., Ithaca, NY, July 1979, Revised Jan. 1980.
26. Teitelman, W. INTERLISP reference manual. Xerox PARC, 1974.
27. Teitelman, W. A display-oriented programmer's assistant. Xerox PARC, March 1977.
28. Wilcox, T.R., Davis, A.M., and Tindall, M.H. The design and implementation of a table driven, interactive diagnostic programming system. *Comm. ACM 19*, 11 (Nov. 1976), 609–616.
29. Zelkowitz, M. Reversible execution as a diagnostic tool. Ph.D. Thesis, Dept. of Comptr. Sci, Cornell Univ., Ithaca, N.Y., Jan. 1971.

# Experiences with the PSG - Programming System Generator

G. Snelting*

Institut für praktische Informatik
Technische Hochschule Darmstadt

Abstract The programming system generator developed at the Technical University of Darmstadt generates sophisticated interactive programming environments from formal language definitions. From a formal, entirely nonprocedural definition of the language's syntax, context conditions and denotational semantics, it produces a hybrid editor, an interpreter and a library system. The editor allows both structure editing and text editing, guaranteeing immediate recognition of syntax and semantic errors. The generator has been used to generate environments for PASCAL, MODULA-2 and the formal language definition language itself. A brief description of the generated environments and the definition language is given, and our experiences with formal language definitions are discussed from the language definer's point of view as well as from the programmer's point of view using the generated environments.

## 1. Introduction

The Programming System Generator PSG developed at the Technical University of Darmstadt generates language-dependent interactive programming environments from formal language definitions. From a formal definition of a language's syntax, context conditions, denotational semantics and additional information it produces an integrated software development environment. One of the major components of a PSG environment is a powerful hybrid editor which allows structure oriented editing as well as text editing. In structure mode, the editor guarantees prevention of both, syntactic and semantic errors, whereas in textual mode it guarantees their immediate recognition. The editor is generated from the language's syntax and context conditions. Furthermore, a PSG environment includes an interpreter which is generated from the language's denotational semantics. A language-independent library system is part of a PSG environment.

The basic units for editing and interpreting are called fragments. A fragment is an arbitrary part of a program, for example a statement, a procedure declaration or a whole program. Fragments are internally stored as abstract syntax trees. Fragments may be incomplete, that is, subcomponents may be missing. Missing subcomponents are called templates. Bottom-up system development is provided by combining fragments, while the fragments themselves are constructed top-down.

The editor supports two input modes, which may be mixed freely by the user. In textual mode, the editor behaves like a normal screen-oriented text editor with the usual capabilities to enter, modify, delete, search etc. text. By keystroke, incremental syntactic and semantic analysis are invoked. If the input was error-free, the text will be pretty-printed and editing may proceed. If any syntactic or semantic errors are detected, an error message will be displayed by a menu-driven error recovery routine. Earliest possible detection of both syntactic and semantic errors is guaranteed: As soon as a fragment cannot be embedded into a syntactically and semantically correct program, it will be classified as erroneous. For semantic errors, this works even if declarations of e.g. variable types are still missing.

In structured mode, programs are developed in menu-driven refinement or modification steps. The menus are generated according to the abstract syntax of the language. The usual structure oriented commands are offered to the user, such as refinement of a structure, selection from alternatives of a syntactic class, modification, insertion, and deletion of substructures, zooming of substructures, copying of substructures etc. However, the menus are filtered dynamically by the context analysis, such that only those menu-items producing syntactically <u>and</u> semantically correct refinements after selection will be offered to the user. Thus, in structural input mode, neither syntactic nor semantic errors can occur. In addition the user may retreive the context information which has been derived so far. For example, he might ask the system which variables are already declared, which variables are still undeclared, what possible types the undeclared variables may possess etc*.

Like the other system components, the interpreter is able to handle arbitrary incomplete fragments. As long as control flow in the interpreted fragment does not touch any syntactically incomplete structure, the fragment can be interpreted without difficulties. If flow of control encounters a template,

* Work of this author was supported by the "Deutsche Forschungsgemein-

* According to our philosophy, declaration before use is not required. An undeclared variable is considered a semantic error as soon as the last template offering the possibility of declaring that variable has been deleted

the editor will be invoked asking the user to enter the missing parts of the fragment. Alternatively, the language definer may force the interpreter to ask the user for e.g. values of uninitialized variables or missing expressions.

A language-independent fragment library sytem where fragments are stored as abstract syntax trees is also part of a generated environment. Reading, writing and rewriting of fragments is automatically performed by the editor if required. Deletion of fragments requires an explicit user command. PSG environments offer the facility of redirecting input to external text files. Furthermore, fragments may be written in pretty-printed style onto external files.

## 2. What the language definer has to do

One of the most important goals during the development of PSG has been the definition of a formal language definition language covering the whole spectrum of a language's syntax, context conditions and dynamic semantics as well as all of the additional information required by an interactive environment e.g. menu texts or pretty-printing information. Thus, the language definer working with PSG is offered a formal, nonprocedural definition language. This is in striking contrast to most existing environ-ment generators, which frequently support only the formal definition of the syntactic aspects of a language. For example, the language definer working with GANDALF [Hab82a] has to write so-called action routines in an ordinary programming language; these action routines will perform tasks such as type checking, code generation etc. Using the Cornell Program Synthesizer (CPS) [Rep81], which is based on attributed grammars, the language definer has to code certain attribute functions in the language C.

A PSG language definition consists of three major parts: the definition of the syntax, the context conditions, and the denotational semantics of the language. The first part is mandatory, the others are optional. Syntax and semantics definition rely on well-known concepts. However, new concepts based on AI technology had to be developed for defining and checking context conditions, due to the specific requirements of interactive environments where programs are usually incomplete containing e.g. pending variable declarations.

### Definition of the syntax

The syntax definition part starts with the definition of the lexical structure of the language, which is used to generate a scanner. The language definer has to specify all reserved words and all delimiters (special symbols). Each lexical entity is given a name. For PASCAL, this looks as follows*:

```
if -> 'IF';
then -> 'THEN';
else -> 'ELSE';
becomes ->' := ';
equal ->' = ';
sem ->';';
```

etc.

The abstract syntax, which forms the second part of the syntax definition, is the core of any language definition. All other parts of a language definition refer to the abstract syntax. Abstract syntax rules look like this:

```
CLASS statement = assignment, forstatement, compound, ifstatement, call, ... ;
NODE assignment :: variable expression;
NODE forstatement :: Id expression to_or_downto expression statement;
NODE compound :: statementlist;
LIST statementlist = statement+;
NODE ifstatement :: expression statement [statement];
NODE call :: Id [parameterlist];
LIST parameterlist = expression+;
CLASS variable = Id, record_ref, array_ref, pointer_ref;
CLASS expression = variable, constant, addition, subtraction, ... ;
NODE addition :: expression expression;
```

etc.

CLASS rules describe syntactic alternatives. NODE rules define substructures of a syntactic entity. Substructures which are optional are enclosed in square brackets. The number of a node's substructures is fixed, although they may be of different syntactic type. LIST rules define syntactic entities with a variable number of substructures of the same syntactic type.

In a PSG environment, fragments are internally represented by abstract syntax trees. Missing substructures of a node are represented by tree templates; they serve as placeholders for pending refinements. Missing sublists of a list are called list templates, they may be moved, deleted and inserted freely within a list.

* In the following, all examples refer to PASCAL

immediately that 'i' has type integer (or a subrange thereof), that 'a' is a one-dimensional array with index and component type integer, and that the still missing right-hand side of the assignment must also be compatible with integer. If a user types 'TRUE' as the right side, a semantic error must immediately be reported. In addition, the menu for the right-hand side template should be filtered in such a way that the menu item for the constant 'TRUE' will not be displayed, as well as all other non-integer expression items.

The classical methods follow the scheme: first inspect the declarations and collect information about e.g. types of variables, then use this information to check type incompatibilities in expressions etc. This scheme does not work in the above example.

The concept of context relations [Hen84] has been developed to overcome these difficulties with the classical methods. The basic idea is to compute a set of still possible attributes for each node of an incomplete fragment. A collection of still possible attribute assignments to the nodes of a fragment is called a context relation. If such a relation consists of exactly one tuple, the context information is unambiguous. If a relation is empty, a semantic error has been detected. It can be shown that the context relation of a composite fragment is just the natural join of the relations of its subfragments. Therefore context conditions may be computed incrementally during editing. As context relations are in general of infinite size, they are represented in a finite way using so-called term form relations with variables. The basic idea is to describe the set of possible attributes by a grammar, the so-called data attribute grammar. Infinite sets of attributes are then represented by incomplete derivation trees according to the data attribute grammar; in addition these derivation trees may contain arbitrary functional dependencies between (sub)trees.

To specify context conditions, the language definer first has to define the scope and visibility rules of the language. This information is used to determine whether all the different occurences of an identifier in a fragment actually denote the same "abstract" identifier. If so, their corresponding sets of still possible attribute values may be intersected.

The second part of the context conditions definition is the specification of the data attribute grammar. Here, the structure of the attributes of the language is defined. Typical rules look like this:

```
NODE attribute :: type class;
CLASS type = simple_type, array_type, set_type, ... ;
CLASS simple_type = arithmetic, ordinal;
CLASS ordinal = integer, boolean, subrange, enumeration, ... ;
NODE settype :: ordinal;
NODE arraytype :: index_types type;
LIST index_types = ordinal+;
CLASS class = variable, ctype, constant, procedure, function, ... ;
```

-etc.

The attribute format definition forms the third part of the context conditions definition. Similar to the format definition of the context-free part of the language definition, it specifies how attributes shall be displayed to the user if he looks at the symbol table.

The last and most important part of the context condition definition is the specification of the so-called basic relations, which must be specified for all terminals and each node rule of the abstract syntax. As the context relation of a fragment is the join of the relations of its components, specification of the basic relations provides enough information to analyse each fragment incrementally. A basic relation consists of a set of tuples which define a (possibly infinite) set of attribute assignments to the components of a node rule resp. a terminal. For instance, the basic relation of a syntactic integer number consisting of a single tuple might be:

```
Int: MK-attribute(integer, constant);
```

which specifies that an integer number has type integer and is a constant. More sophisticated specifications can be obtained by using variables, which specify that certain subattributes must be identical. The basic relation for an assignment

```
assignment :: variable expression
```

contains three tupels, which use the variable TYPE:

```
assignment: NIL MK-attribute(TYPE, variable) MK-attribute(TYPE, computational)
          | NIL MK-attribute(real, Variable) MK-attribute(integer, computational)
          | NIL Mk-attribute(TYPE, function) MK-attribute(TYPE, computational);
```

which says that in an assignment either

- the left-hand side is a variable of a certain TYPE, and the right-hand side is an expression of the same TYPE, or
- the left-hand side is a real variable, and the right-hand side is an integer expression, or
- the left-hand side is a function identifier with a certain result TYPE,

The structure oriented commands and menus offered to the user are generated according to the abstract syntax. For example, each template is associated with a menu of refinement possibilities. However, this menu is dynamically filtered with respect to context conditions (see below).
The concrete syntax, which is the third part of the syntax definition, is used to generate an incremental parser. The concrete syntax is restricted to full LL(1) grammars. It includes transformation rules which specify how to build abstract trees from textual input. Thus the concrete syntax is actually a string-to-tree transformation grammar. Concrete syntax rules look like this:

```
statement ::= ...
            | NODE for, Id, becomes, expression, to_or_downto, expression,
                   do, statement => forstatement
            | NODE begin, statementlist, end => compound
            | NODE Id, optparameterlist => call;
statementlist ::= LIST statement+-sem;
optparameterlist ::= [lp,parameterlist,rp];
to_or_downto ::= TERMINAL to | TERMINAL downto;
```

etc.
The NODE, LIST and TERMINAL keywords and the '[', ']', and '=>' delimiters specify how to build the abstract tree during the parsing process. However, the situation is not always that simple. Frequently, a concrete syntax does not merely reflect the rules of the abstract syntax, due to operator precedences or left-factorization used to avoid LL(1)-conflicts. For example,

```
expression ::= simple_expression, simplexpr_tail;
simple_expression ::= factor, ...;
simpleexpr_tail ::= UPDATENODE equal, simple_expression => equal_expr
                  | EMPTY;
```

Here, the UPDATENODE and EMPTY rules will construct a correct equal_expr node, although the rules reflect operator precedence and are left-factorized.
The parser will parse any input entered in textual mode. It accepts arbitrary valid prefixes of any input conforming to the syntactical category of a given template. If any syntax errors are detected, a recovery routine will compute a menu comprising all local correction possibilities, which is presented to the user. The user may then correct his input either in textual mode or by selection among the menu items.

Being the fourth part of the syntax definition, the format definition is a tree-to-string transformation grammar which is used to construct the external textual representation of an abstract tree. Prettyprinting information is part of the format definition:

```
forstatement => | for Id becomes expression to expression do statement [2];
ifstatement => | if expression then statement[2] (statement[2] -> | else,);
```

In the example, '|' means start of a new line, and indentation factors may be specified inside square brackets. Parentheses are used to specify conditional formatting: the keyword 'ELSE' will be displayed only if the optional else-part of an 'ifstatement' is indeed present. Conditional formatting is used also to re-insert parentheses into expressions if neccessary due to operator precedence (note that parentheses are discarded during parsing and that operator precedences are reflected by the abstract tree's structure). A string-to-tree-to-string transformation which is performed by parsing textual input, building the abstract tree and pretty-printing the abstract tree must yield the original input text exactly except for spaces, newlines and redundant parentheses.
In the last part of the syntax definition, headers and menu texts have to be specified which are used to generate the textual representation of templates and menus. For each name occuring in the abstract syntax an external name has to be specified:

```
statement -> 'Anweisung';
ifstatement ->'Bedingte Anweisung';
```

For each syntactic class, menu texts have to be specified:

```
statement -> 'Zuweisung', 'FOR-Anweisung', 'Verbundanweisung', ... ;
```

For purposes of generality, syntactic entities may posses different external names, depending on their occurence in templates or in menus.

<u>The definition of context conditions</u>
The context analysis of PSG has been of special interest, since the classical methods like attributed grammars [Knu68] turned out to be inadequate even if attribute evaluation is performed incrementally [Rep83]. Consider the following situation: In a PASCAL program-fragment, the variables 'a' and 'i' have not yet been declared or used, and a declaration-template is still present. Now the user enters an incomplete assignment:

```
a[a[i+1]]:=
```

Although 'a' and 'i' are still undeclared, the context analysis must derive

and the right-hand side is an expression of the same TYPE.*
During editing, an inference engine is used to derive context information from the basic relations as demonstrated in the above example. Note the similarity to the AI-paradigma of inference-rule-based deduction systems.

### The definition of semantics

Within the PSG system, the dynamic semantics of a language is defined in denotational style [Gor79]. The denotational semantics is used to generate an interpreter. The semantic functions are defined in a META-IV-like [Bjo78] extension of type-free lambda calculus. This metalanguage supports high--level conceps like lists and maps and allows the definition of higher--order-functionals of arbitrary rank. The terms of the metalanguage are used as an universal intermediate language. If a fragment is to be executed, it will be translated into a term of the metalanguage, using the definitions of the semantic functions. This term will be interpreted, that is, reduced to normal form. The resulting term is the result of program execution.
In contrast to systems like SIS [Mos79] our interpreter allows interaction with the user during program execution in order to supply input data, to enter values of uninitialised variables etc.
The definition of the semantics consists of three parts. First of all, a set of auxiliary functions to be used elsewhere in the semantics definition may be defined. For example, the definition of a "distributed concatenation" function for a list of lists (which is supposed to be used in several distinct semantic functions for different types of lists) looks as follows:

```
disconc = LAM list_of_lists. IF NULL list_of_lists THEN <>
              ELSE CONC HEAD list_of_lists, (disconc TAIL list_of_lists);
```

Here, LAM denotes functional abstraction, parentheses denote functional application. NULL is a test for the empty list, CONC, HEAD and TAIL have their usual meanings, and '<>' denotes the empty list.
The main part of the semantics definition comprises the semantic functions for each syntactic entity. In a PASCAL-subset without GOTOs and side effects of functions, the meaning of a statement may be defined as a functional which maps environments onto functions which map states to states. The meaning of an expression is a functional which maps environments onto functions from states to values. An environment is a map which maps identifiers to ‹location, descriptor› pairs. A state is a map which maps locations to values. Thus, the semantic function for a conditional statement might look as follows:

```
ifstatement: LAM env. LAM state. IF (( |[ expression ]| env) state)
                 THEN (( |[ statement 1 ]| env) state)
                 ELSE (( |statement 2:LAM env.LAM state.state| env) state);
```

The '|[' and ']|' brackets are the "meta-brackets" which denote the meaning functions of the subcomponents of a node. The special form '|' ... '|' is used for subcomponents which are optional (as the ELSE-part in our example). If the optional subcomponent is missing, the function following the colon will be used.
The third part of the semantics definition describes the meanings of the executable fragments. Typical examples are

```
procedure_declaration: '', ERROR 'Procedure declaration is not executable';
statement: 'Result of statement execution with no variables declared
            or initialized:', (( |[ statement ]| []) []);
```

where '[]' denotes the empty map. Note the difference between the result of a 'statement' execution specified here and the semantic function for the syntactic class 'statement', to which the above definition refers.

## 3. Experiences with the generator and the generated environments

Until now, environments have been generated for Algol60, PASCAL, MODULA-2, the language definition language itself, and some experimental specification languages. The language definition environment has been used intensively not only by the members of the project team, but also by lots of students, as most of our environments have been generated as part of diploma theses. At Kaiserslautern, PSG has been used along with other systems (GAG [Kas8?] and GANDALF) in student projects for the implementation of a PASCAL-subset. The PASCAL-environment was used to implement other parts of the PSG system. Thus, we feel that by now we have gathered substantial experience and that we are able to compare our approach to others, especially GANDALF and CPS.

### The benefits of a formal language definition language

In [Hab84], Habermann states that "the state of the art has not reached the point where all of the task-specific (i.e. language specific) part (of an environment) can be formally described and automatically generated". However, our experience with PSG indicates that this point has been reached by now, at least for languages of a complexity not greater than that of e.g.

* For the sake of readability, this specification does not exactly reflect the ISO-standard "assignment compatibility"

PASCAL.

The use of a formal language definition language has many advantages:

- In view of the power and complexity of the generated environments, PSG language definitions are very short. Typically, they vary in size between 240 lines for an Algol60 environment without context conditions and semantics and 3600 lines for a MODULA-2 environment including full specification of context conditions and denotational semantics.
- The expressive power of the language definition language allows concentration on the relevant aspects of a language definition. The language definer does not have to concern himself with minor details such as the organization of symbol tables etc.
- PSG language definitions are safe, since all inconsistencies in a definition are detected at generation time.*
- The modular design of the language definition language improves readability and reliability. It allows the independent definition of the syntactic, context dependent, and semantic aspects of a language, once the abstract syntax has been defined.
- a formal language definition language is an ideal tool during the development of new languages. In a "language design lab", language definitions are easily modified and tested.

As a consequence, the amount of manpower to generate an environment is small: A moderately awake graduate student with some background in programming languages and some initial knowlegde of the PSG user interface will specify and debug an Algol60 definition without context conditions and semantics within ten days. The MODULA-2 environment including full specification of context conditions and denotational semantics was defined as part of a diploma thesis within eight months. Thus, the use of a formal language definition language allows the quick generation of correct, reliable and powerful programming environments.

The benefits of the hybrid editor approach

In [Fei84], Kaiser and Feiler state for structure oriented editors that "in order to modify an expression, ... , the user must understand the underlying tree representation and enter a tedious serious of tree oriented clip, delete and insert commands. Unfortunately, complete parsing of all expressions is also nonoptimal". This is true not only for expressions, but also for arbitrary structured statements as well as for any syntactic entity including complete programs. In [Rep81], Teitelbaum and Reps state that "(the change of a while loop into a repeat loop) must be accomplished by moving the constituents of the existing WHILE-template into a newly inserted UNTIL-template. Although such modifications can be made rapidly ..., they are admittedly awkward". Within a PSG environment, problems of this kind do not exist, since users may switch freely between textual mode and structure mode. Furthermore, our experience indicates that experienced programmers prefer textual mode not only for modifications, but also to enter e.g. a sequence of statements or even a whole procedure. Since the parser accepts arbitrary incomplete input and, in case of syntax errors, generates a menu of all possible local recovery actions, textual input mode seems to be quite attractive for users who know the concrete syntax of their language. Furthermore, arbitrary parts of a fragment may be read in from an external textfile. On the other hand, unexperienced users tend to prefer structured mode. By simply selecting menu items, they need not bother about syntactic details which they do not know. Thus, the possibility to mix textual mode and structure mode freely seems to be the most flexible, general, and user friendly solution to the dichotomy of viewing programs either as text or as structure.

The benefits of dynamic context sensitive menu filtering

In [Hab82b] Habermann states that "we believe that preventing mistakes is far superior to making the user fix them. ... (however) as to semantic errors it is difficult to see how to avoid them". Within a PSG environment, structured mode prevents syntactical and semantical errors due to the dynamic context-sensitive menu filtering. This feature is not provided by any other environment known to us. In textual mode, the user may always type arbitrary nonsense, but syntactical and semantical errors will be detected immediately. This guarantees that programs are correct at every stage of their development.

We believe that our environments support syntactic and semantic error prevention in the best possible way. There is, however, one problem in connection with certain modifications: if a user modifies e.g. a procedure declaration by adding an extra parameter, context incompatibilities will occur at each place where the procedure is called. If the calls are modified first, they will become incompatible with the procedure declaration. Although it might be considered bad programming style to modify the types of objects in an uncontrolled manner, the user can circumvent such situations

* At the moment, this is not true for the semantics definition, as it is based on type free lambda calculus. However, the implementation of a type inference algorithm allowing handling of polymorphism, overloading and coercions is about to be completed (see [Let84]).

by temporarily deactivating the context analysis. It is planned to modify the context analysis in a way that enables it to tolerate faulty subtrees temporarely.

Drawbacks in generality and performance

PSG is not the ultimate system, as there remain several unsatisfying points. A formal language definition language enables language definers to generate environments in a rapid and reliable way. However, the current implementation of the definition language imposes some restrictions on the class of languages which may be defined with PSG.
First of all, if the concrete syntax of a language cannot be made LL(1), the language cannot be defined within PSG. It is, however, difficult to see how to incorporate a more powerful parsing technique. Bottom-up techniques such as incremental LR parsing ([Cel78]) do not fit in our framework. LL(k) with k > 1 is problematic in view of the requirement that arbitrary valid prefixes of sentential forms must be parseable.
Certain languages have context conditions which are not definable within the current definition language. The scope and visibility analysis cannot handle features like elliptical record references in PL/1 or FORWARD procedure declarations in PASCAL (which will lead to a 'double declaration' error). Within our framework - no declarations required before use - FORWARD declarations do not make sense anyway. The context analysis phase is unable to handle user-defined polymorphic or overloaded objects such as overloaded functions in ADA. We are currently working on a more powerful specification language for context conditions which will overcome these shortcomings. Finally, the semantics definition language is unable to handle any form of parallelism.
The performance of PSG environments has not yet reached production quality, as far as context analysis and program execution are concerned. For the context analysis, this is primarily a problem of the current implementation, which is merely a prototype. We expect that a more sophisticated implementation will result in a time speedup factor of at least five. However, the intrinsic complexity of the method is greater than that of e.g. attributed grammars: For an abstract syntax tree containing n nodes the Reps/Teitelbaum algorithm will perform with O(n), wharas our method requires O(n*ln(n)).
The performance difficulties concerning program execution are of a slightly different nature, as we have difficulties to see how to speed up the interpreter simply by improving its implementation. The interpreter is much faster than that of SIS. However, it is not fast enough for production programs, as is also noted by Pleban for PSP ([Ple84]). We think that these shortcomings may be overcome by compilation of the metalanguage terms [Bah84b], utilizing techniques like data flow analysis, elimination of unneccesary call-by-name and delayed evaluation, and elimination of tail recursion and linear recursion.

## 4. Conclusion

We presented the PSG programming system generator, which generates powerful interactive programming environments from formal language definitions. The pros and cons of using a formal, entirely nonprocedural language definition language have been discussed. It turned out that use of a formal definition language allows very simple and rapid generation of reliable and powerful environments. On the other hand, certain strange and complicated features of certain languages are not definable with the currently implemented definition language, and the performance of the generated environments has not yet reached production quality. Nevertheless, we believe that the use of formal language definitions is an appropriate tool, and that the shortcomings in performance will be captured by more sophisticated implementations, which are still under way.

## 5. Acknowledgements

I thank the other members of the project team, namely R. Bahlke, W. Henhapl, M. Hunkel, M. Jäger and T. Letschert for their valuable comments during the development of this paper.

## 6. References

[Bah84a] Bahlke, R. and Snelting, G.: Programmiersystemgenerator. Arbeitsbericht 1984. Bericht PU2R2/84, Fachgebiet Programmiersprachen und Übersetzer II, Technische Hochschule Darmstadt, Februar 1984.
[Bah84b] Bahlke, R. and Letschert, T.: Ausführbare denotationale Semantik. Proc. 4. GI-Fachgespräch Implementierung von Programmiersprachen, Zürich, März 1984.
[Bjo78] Bjørner, D. and Jones, C.B. (eds.): The Vienna Development Method: The metalanguage. LNCS 61, Springer Verlag 1978.
[Cel78] Celentano, A.: Incremental LR parsers. Acta Informatica 10 (1978), 307-321.
[Fei84] Kaiser, G.E. and Feiler, P.: Generation of language-oriented editors. Proc. Programmierumgebungen und Compiler, Berichte des German Chapter

# Virtual Machine Showdown: Stack Versus Registers

YUNHE SHI[1] and KEVIN CASEY
Trinity College Dublin
M. ANTON ERTL
Technische Universität Wien
and
DAVID GREGG
Trinity College Dublin

Virtual machines (VMs) enable the distribution of programs in an architecture-neutral format, which can easily be interpreted or compiled. A long-running question in the design of VMs is whether a stack architecture or register architecture can be implemented more efficiently with an interpreter. We extend existing work on comparing virtual stack and virtual register architectures in three ways. First, our translation from stack to register code and optimization are much more sophisticated. The result is that we eliminate an average of more than 46% of executed VM instructions, with the bytecode size of the register machine being only 26% larger than that of the corresponding stack one. Second, we present a fully functional virtual-register implementation of the Java virtual machine (JVM), which supports Intel, AMD64, PowerPC and Alpha processors. This register VM supports inline-threaded, direct-threaded, token-threaded, and switch dispatch. Third, we present experimental results on a range of additional optimizations such as register allocation and elimination of redundant heap loads. On the AMD64 architecture the register machine using switch dispatch achieves an average speedup of 1.48 over the corresponding stack machine. Even using the more efficient inline-threaded dispatch, the register VM achieves a speedup of 1.15 over the equivalent stack-based VM.

Categories and Subject Descriptors: D.3.4 [**Programming Language**]: Processor—*Interpreter*

---

[1]**Extension of Conference Paper**: An earlier version of the work in this paper appeared in the First ACM/Usenix Conference on Virtual Execution Environments (VEE'05) [Shi et al. 2005]. The new material in this paper consists of (1) a complete reimplementation of the register VM using the Cacao 0.95 JVM (2) SSA form intermediate representation (3) redundant load elimination using SSA (4) virtual register allocation to minimize the size of Java stack frame, (5) support of two additional VM instruction dispatch methods: direct threaded and inline threaded, (6) support additional architectures such as AMD64, Alpha, and PowerPC, (7) additional optimizations investigated, such as preliminary work on assessing the impact of redundant field and array load elimination
Corresponding author's address: David Gregg, Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland. `David.Gregg@cd.tcd.ie`.
Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2008 ACM 1544-3566/2008/01-ART21 $5.00 DOI 10.1145/1328195.1328197 http://doi.acm.org/10.1145/1328195.1328197

13866



General Terms: Performance, Language

Additional Key Words and Phrases: Interpreter, virtual machine, register architecture, stack architecture

**ACM Reference Format:**

Shi, Y. Casey, K., Ertl, M. A., and Gregg, D. 2008. Virtual machine showdown: stack versus registers. ACM Trans. Architec. Code Optim. 4, 4, Article 21 (January 2008), 36 pages. DOI = 10.1145/1328195.1328197 http://doi.acm.org/10.1145/1328195.1328197

## 1. MOTIVATION

Virtual machines (VMs) enable the distribution of programs in an architecture-neutral format, which can easily be interpreted or compiled. The most popular VMs, such as the Java virtual machine (JVM) and Microsoft .NET's common language runtime (CLR), use a virtual stack architecture, rather than the register architecture that dominates in real processors.

Interpreters are frequently used to implement virtual machines because they have several practical advantages over native code compilers. Interpreters are much slower than the native code produced by just-in-time compilers (even the fastest interpreters are currently about 5–10 times slower), but they are nonetheless widely used for lightweight language implementations. If written in a high-level language, interpreters are portable; they can simply be recompiled for a new architecture, whereas, generally, a just-in-time (JIT) compiler requires considerable porting effort. Interpreters also require little memory: the interpreter itself is typically much smaller than a JIT compiler [Radhakrishnan et al. 2000], and the interpreted bytecode is usually a fraction of the size of the corresponding executable native code. For this reason, interpreters are commonly found in embedded systems. Furthermore, interpreters avoid the compilation overhead in JIT compilers. For rarely executed code, interpreting is typically much faster than JIT compilation. The Hotspot JVMs [Sun-Microsystems 2001] take advantage of this by using a hybrid interpreter/JIT system. Code is initially interpreted, saving the time and space of JIT compilation. Sections of code are then JIT compiled only if they are found to be executed frequently. Interpreters are also dramatically simpler than compilers; they are easy to construct, and easy to debug. Finally, it is easy to provide tools such as debuggers and profilers when using an interpreter because it is easy to insert additional code into an interpreter loop. Providing such tools for native code is much more complex. Interpreters provide a range of attractive features for language implementation. In particular, most scripting languages are implemented using interpreters.

### 1.1 Previous Work

A long-running question in the design of VMs is whether a stack architecture or a register architecture can be implemented more efficiently with an interpreter. Stack architectures allow smaller VM code so less code must be fetched per VM instruction executed. However, stack machines require more VM instructions for a given computation, each of which requires an expensive (usually

1380c



unpredictable) indirect branch per VM instruction dispatch. Several authors have discussed the issue [Myers 1977; Schulthess and Mumprecht 1977; McGlashan and Bower 1999; Winterbottom and Pike 1997] and presented small examples where each architecture performs better, but no general conclusions can be drawn without a larger study.

The first large-scale quantitative results on this question were presented by Davis et al. [Davis et al. 2003; Gregg et al. 2005] who translated JVM stack code to a corresponding register machine code. A straightforward translation strategy was used with simple compiler optimizations to eliminate instructions that become unnecessary in register format. Of the resulting register code, around 35% fewer VM instructions were needed to perform the same computation than the stack code. However, the resulting register VM code was around 45% larger than the original stack code and resulted in a similar increase in bytecodes fetched. Given the high cost of unpredictable indirect branches, these results strongly suggest that register VMs can be implemented more efficiently than stack VMs with an interpreter. However, this work did not include an implementation of the virtual register architecture, so no real running times were presented.

### 1.2 Contribution

The present paper extends the work of Davis et al. in two respects. First, our translation from stack code to register code and subsequent optimization are much more sophisticated. We use a more aggressive copy propagation approach to eliminate almost all of the stack load and store VM instructions. We also optimize redundant constant load and other common subexpressions and move loop invariants out of loops. The result is that an average of more than 46% of executed VM instructions are eliminated. The resulting register VM code is roughly 26% larger than the original stack code, compared with the 45% for Davis et al. We find that the increased cost of fetching more VM code requires an average of only 1 extra CPU load per executed VM instruction eliminated. Given that VM dispatches are much more expensive than CPU loads, this indicates strongly that register VM code is likely to be much more time-efficient when implemented with an interpreter. The cost of this gain is the slightly increased VM code size.

The second contribution of our work is measurements of running times and code behaviour for a fully functional, interpreter-based implementation of a register JVM. We present comparative experimental results for four different VM instruction dispatch mechanisms on twelve different benchmark programs from the SPECjvm98 and Java Grande benchmark suites. Measurements are included from hardware performance counters that allow us to investigate the effect of using a register rather than stack VM on the microarchitectural behaviour of the interpreter.

### 1.3 Other Factors

There are other factors to consider in the choice of code format. Compiling source code to stack-based bytecode is usually simpler than compiling to register code,

138od



one of the reasons being that there is no need for a register allocator. If the compilation has been simple, stack code is also usually relatively simple to decompile. Similarly, stack-based bytecode may be better suited than register code as a source language for JIT compilation, at least partly because there is no assumption about the number of available registers. Apart from execution speed and suitability for JIT compilation, there are other issues in the choice of code format:

*Code Size.* One of the attractions of a stack VM is that the code is quite compact, due to the absence of explicit register operands. Later in this paper, we present work that shows that the bytecode for a register VM is only 26% larger than stack bytecode. In the case of Java, however, the bytecode only accounts for about 18% of a class file [Antonioli and Pilz 1998], the rest being occupied primarily by the constant-pool. This constant-pool is a table, used to store interface, class and field names and various constants used by the the class. Various techniques such as those employed by JAX [Tip et al. 2002] can be employed to reduce the constant-pool size. As a result bytecode can occupy as much as 75% of the memory footprint in some embedded systems [Clausen et al. 2000].

There are other options for the code format. For example, compressed syntax-tree based representations [Kistler and Franz 1999] are around twice as compact as stack-based bytecode, and are often considered a better source language for JIT compilation because they retain most of the high-level information from the source code. However, such tree based encodings are difficult to interpret efficiently, so they are most suitable when the VM will be implemented using only a JIT compiler.

*Compressed Code Size.* Code size is not only important because of the memory consumed, but also because programs may need to be sent over networks. In the case of Java, the contents tend to be easily compressed, repeating text, highly suitable for the jar file format commonly used for classfile transport. Typically classfiles are compressed to about 50% of their original size, and schemes have been proposed that compress classfiles even further (up to 10% to 25% of their original size [Pugh 1999]).

*Preparation Time.* Much work has been done in the JVM in the area of bytecode verification, a task which is greatly simplified by the simpler stack IR. One area which a VM designer may wish to consider, but which we do not examine in this paper, is the issue of how much more difficult bytecode verification becomes when dealing with a register IR.

*Portability.* We do not envisage a huge difference between a stack-based IR and a register-based one, as long as neither make assumptions about the underlying hardware.

*Complexity of Implementation.* As we note elsewhere, a stack IR can be an easier compilation target (the complexity of the compiler being the issue here). From a VM interpreter point of view, a stack IR and register IR in terms of complexity of implementation seem, from our experience, to be roughly equivalent. It is a different issue if one is choosing an IR with a view to the complexity of the JIT in a VM (if present). For a naive inlining JIT, a register IR is clearly preferable while for a more sophisticated JIT, a stack IR may be preferred.

1.3.ae



The choice of IR for the work presented in this paper was driven primarily by a different concern to those discussed above. To allow a meaningful comparison between a stack and register VMs, it was decided to keep the instruction sets as similar as possible. Where experimental issues are not the driving force, the choice of IR is likely to be made on the basis of a combination of these issues.

### 1.4 Paper Overview

The rest of this paper is organized as follows. In section 2, we describe the main differences between virtual stack machine and virtual register machines from the perspective of an interpreter. In section 3, we show how stack Java bytecode is translated into register bytecode and the optimizations applied on the new code. In section 4, we analyze the static and dynamic code behaviour before and after optimization, and we show the performance improvement in our register JVM when compared to the original stack JVM. In section 5, we examine other possible optimizations. In section 6, we discuss how our results can be extended to other VMs. Research related to our work is examined in section 7. Finally, in section 8, we conclude with a summary of the work presented in this paper.

## 2. STACK VERSUS REGISTER

The cost of executing a VM instruction in an interpreter consists of three components: dispatching the instruction, accessing the operands and performing the computation. In this section we consider the influence of these three components on the running time of VM interpreters.

### 2.1 Dispatching the Instruction

In instruction dispatch, the interpreter fetches the next VM instruction from memory and jumps to the corresponding segment of its code that implements the fetched instruction. A given task can often be expressed using fewer register machine instructions than stack ones. For example, the local variable assignment `a = b + c` might be translated to stack JVM code as `iload c, iload b, iadd, istore a`. In a virtual register machine, the same code would be a single instruction `iadd a, b, c`. Thus, virtual register machines have the potential to significantly reduce the number of instruction dispatches.

Instruction dispatch is typically implemented in C with a large `switch` statement, with one case for each opcode in the VM instruction set. Switch dispatch is simple to implement, but is rather inefficient. Most compilers produce a range check and an additional unconditional branch in the generated code for the `switch`. In processors using a branch target buffer (BTB) for indirect branch prediction, there is only one entry in the BTB for all indirect branch targets. Thus, the indirect branch generated by most compilers is highly unpredictable (around 95% [Ertl and Gregg 2003]) on architectures using a BTB for indirect branch prediction. The main advantages of switch dispatch are that the bytecode executed by the VM is compact, and it can be implemented using any ANSI C compiler.

An alternative to the `switch` statement is *token-threaded dispatch* [Klint 1981]. Threaded dispatch takes advantage of languages with labels as first

138 of



class values (such as GNU C and assembly language) to optimize the dispatch process, at the expense of the portability of the interpreter source code. Token-threaded dispatch uses the opcodes to lookup the target address of their implementation in a dispatch target address table. This enables the range check and additional unconditional branches to be eliminated, and permits the code to be restructured to improve the predictability of the indirect branch dispatch (to around 45% [Ertl and Gregg 2003]). On architectures with BTBs for indirect branch prediction, each instruction implementation has its own indirect branch instruction and thus, multiple entries of indirect branch targets can exist in the BTB.

Another alternative is *direct-threaded dispatch* [Bell 1973]. Direct-threaded code directly encodes the jump addresses as the opcodes of instructions and thus further reduces the cost of dispatch. The code to be interpreted is translated from bytecode into threaded code. In threaded code, VM opcodes are no longer bytes, but are instead addresses of the executable native code within the interpreter that performs the computation that corresponds to the original VM opcode. Thus the table lookup from token-threaded code can be eliminated, further reducing the cost of dispatch. Direct-threaded dispatch requires first class labels, a translation step, and the VM code size increases by up to a factor of four on a 32 bit machine or eight on a 64 bit machine.

An even more sophisticated approach is *inline-threaded dispatch* [Piumarta and Riccardi 1998] which copies executable machine code from the interpreter and relocates it to remove the dispatch code entirely. This requires an even more complicated translation from bytecode, much greater memory requirements, and is even less portable than the other forms of threaded dispatch. It is, however, the fastest VM instruction dispatch mechanism, and we present results for it in this paper.

Another alternative is context threading [Berndl et al. 2005]. The context threading approach uses subroutine threading to change indirect branches to call/returns, which better exploits the hardware return-address stack to reduce the cost of dispatches. However, this approach requires some mechanism to generate native executable machine code at run time. We have not implemented this dispatch mechanism, although we believe that it is slightly less efficient than inline threading, which eliminates indirect branches entirely.

As the cost of dispatches falls, any benefit from using a register VM instead of a stack VM falls. However, `switch` and token-threaded dispatch are the most commonly used interpreter techniques because two of the main motivations for using an interpreter are to avoid additional translation steps, and to maintain the small size of bytecode. If ANSI C must be used (as is the case in the interpreters for many scripting languages) then `switch` is the only efficient alternative.

## 2.2 Accessing the Operands

The location of the operands must appear explicitly in register code, whereas in stack code, operands are found relative to the stack pointer. Thus, the average register instruction is longer than the corresponding stack instruction, register

code is larger than stack code, and register code requires more memory fetches to execute. Small code size and small numbers of memory bytecode fetches are the main reasons why stack architectures are so popular for VMs.

From the viewpoint of a VM interpreter, a stack VM must keep track of the bytecode instruction pointer (IP), the stack pointer (SP), and the frame pointer (FP) while a register VM only needs the IP and FP. Thus, when the register VM is implemented using an interpreter on a real processor, one variable fewer is required in the inner loop of the interpreter than for the stack VM. This reduces real machine register pressure, and may result in less spilling and reloading of variables. On platforms with small numbers of architected registers[2], such as Intel x86 processors which have only eight general purpose registers, this reduction in register pressure may impact performance. Moreover, a stack VM must update the SP as values are pushed or popped.

### 2.3 Performing the Computation

Given that most VM instructions perform a simple computation, such as adding or loading, this is usually the smallest part of the cost. The basic computation has to be performed regardless of the instruction format. However, eliminating loop invariants and redundant loads (common subexpressions) is only possible on a register VM[3]. In Section 3.3, we exploit this property to eliminate repeated loads of identical values in a register VM.

## 3. TRANSLATION AND OPTIMIZATION

In this section, we describe a system of translating JVM stack code to virtual register code and its optimization in a just-in-time manner. This JIT translation was chosen as a useful mechanism to allow us to compare stack and register versions of the JVM easily. Whether or not JIT translation (or any particular run-time translation) from stack format to register format is the optimal way to use virtual register machines remains an open question. In a realistic system, it is most likely that one would use only the register machine, and compile for that directly. Finally, it should be noted that standard, well-known JIT compiler techniques are used for this run-time translation, given that the focus of this research is on the results of the translation, and not on the translation itself.

[2]Architected registers are those registers available to the programmer, and described in the instruction set architecture (ISA). For example, the Pentium III supports the x86 ISA, which has 8 general purpose, 8 floating point and several architected control registers. However, the Pentium III implementation of the x86 ISA uses 64 integer and 64 floating point physical registers which are used internally within the processor. The programmer has no direct access to these physical registers.

[3]In theory, the stack VM can benefit from eliminating complex common subexpressions by storing the computational results in local variables and reloading those values onto the operand stack when needed. In practice, we do not find any such complex common subexpressions, which may be due to the optimization already done by Java compiler. The stack VM will not benefit from simple redundant loads because the value will be loaded onto the stack anyway.

138ch



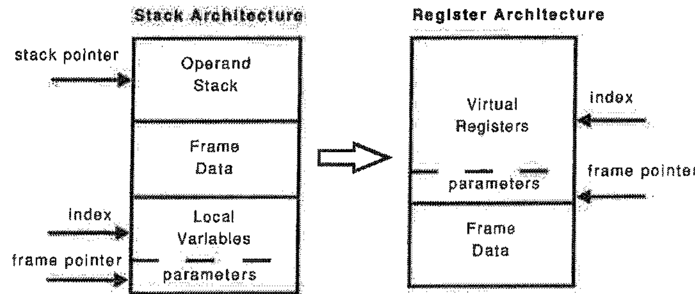



Fig. 1. The structure of a Java frame.

| Stack bytecode | Register bytecode |
|---|---|
| iload_1 | move r1 -> r10 |
| iload_2 | move r2 -> r11 |
| iadd | iadd r10 r11 -> r10 |
| istore_3 | move r10 -> r3 |

Fig. 2. Stack bytecode to register bytecode translation. Assumption: current stack pointer before the code shown above is 10. The registers after -> are destination registers.

### 3.1 Translation from Stack to Register

Our implementation of the JVM pushes a new Java frame onto a run-time stack for each method call. The Java frame for a stack architecture contains local variables, frame data, and the operand stack for the method (see Figure 1). In the stack JVM, a local variable is accessed using an index, and the operand stack is accessed via the stack pointer. In the register JVM, both the local variables and operand stack can be considered as virtual registers for the method. There is a simple mapping from stack locations to register numbers, because the height and contents of the JVM operand stack are known at any point in a program [Gosling 1995]. In practice, the number of virtual registers (local variables and stack slots) in a method will only be limited by the size of the operand used to specify the register number. Each method call has its own set of virtual registers on its Java frame.

In the stack JVM, most operands of an instruction are implicit; they are found on the top of the operand stack. Most of the stack JVM instructions are translated into corresponding register JVM instructions, with implicit operands translated to explicit operand registers. For example, the stack JVM instruction `if_icmpeq branchbyte`$_1$ `branchbyte`$_2$ is encoded with three bytes, one byte for the opcode and two bytes for the branch offset. The instruction takes two operands from the operand stack and performs a comparison to decide the next execution path. After conversion to register code, the operands are no longer implicit. Thus, the instruction becomes `if_icmpeq r`$_1$ `r`$_2$ `branchbyte`$_1$ `branchbyte`$_2$, occupying a total of five bytes, one byte for the opcode, one byte for each register and two bytes for the branch offset. Figure 2 shows another simple example of bytecode translation. The bytecode adds two integers from two local variables and stores the result back into another local variable.

138Qi



There are a few exceptions to the above one-to-one translation rule:

(1) pop and pop2 can be eliminated immediately because they are not needed in the virtual register machine code. Many invoke instructions (method calls) push a return value that is not used by the following instruction onto the operand stack and the stack JVM also uses a number of pop/pop2 instructions purely to maintain consistency of the operand stack.
(2) Instructions that load a local variable onto the operand stack or store a value from the operand stack in a local variable are translated into move instructions.
(3) Stack manipulation instructions (e.g. dup, dup2, ...) are translated into appropriate sequences of move instructions by tracking the state of the operand stack.
(4) Wide stack VM instructions which address local variables numbered above 255 can be handled through the addition of one or more move_wide instructions. These are register VM instructions with four bytes of operands (two bytes for the source and two for the destination). In the case of Java, this simple extension would enable up to 65535 local variables to be addressed, as permitted by the Java Standard.
(5) The iinc instruction in the stack JVM is used to increment a local variable by a constant value. iinc is an interesting VM instruction in stack JVMs because the computation is done without the operand stack. The computation should push an operand and a constant, add, and store the result to a local variable. It can be regarded as a type of register VM instruction that is available in the stack JVM. We translate an iadd or isub into an iinc VM instruction if one of its operands is a small integer constant (i.e. it is preceded by a VM instruction that pushes a small integer constant onto the stack).

### 3.2 Method Invocation

JVM method invocation instructions, such as invoke_virtual, are unusual in that they take a variable number of operands from the stack. As with other instructions, we include the register locations of each of these operands in the register version of the instruction. The result is that method invocation instructions are variable length in the register VM. The number of bytes in the instruction depends on the number of items that the original method call takes from the stack when the method call is made.

In a stack JVM, operands (parameters) always come from the top of the stack, and become the first local variables of the called method (see Figure 1). A common way to implement a stack JVM is to overlap the current Java frame's operand stack (which contains a method call's parameters) and a new Java frame's local variables.

In the register JVM, we do not overlap the Java frames to pass method parameters. Instead, we copy all the parameters from the virtual registers in the

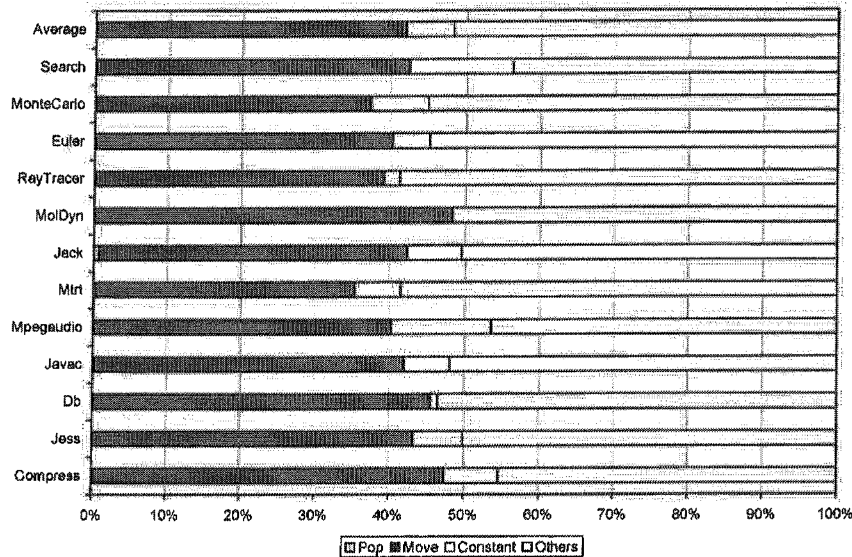



Fig. 3. Different categories of dynamically executed stack VM instructions.

calling method's Java frame into the virtual registers in the Java frame for the new (called) method. We considered a similar mechanism in our virtual register machine as in a stack JVM. We would place the parameters for a method invocation in consecutive registers, in the highest numbered registers for the method. Instead of copying the values of these registers into the stack frame of the called method, we could simply move the frame pointer to point to the first of these parameters. Although this would provide an efficient parameter passing mechanism, it prevents us from copy propagating into the source registers (parameters) of a method call. Even though the operands of method invocation VM instructions are contiguous after initial translation, once we have performed copy propagation and other optimizations this ordering is lost. However, the benefits of our optimizations are much greater than the small loss in efficiency of parameter passing.

### 3.3 Optimization

In the stack architecture, computation is done through the operand stack. The operands of an instruction are pushed onto the operand stack before they can be used, and results are stored from the operand stack to local variables to save the value. In the register architecture, most of the operand stack load and store instructions are redundant. The main objective of optimization is to take advantage of the opportunities provided by a virtual register machine architecture. There are two main categories of redundant loads.

—Loads and stores between operand stack and local variables are translated into `move` instructions in register code. On average, more than 42% (see Figure 3) of executed VM instructions in the SPECjvm98 and Java Grande

13894k



benchmark suites (including library code) consist of loads and stores between local variables and the operand stack.

—Redundant loads of constant values and other arithmetic common subexpressions: In the stack architecture, constants are loaded onto the operand stack each time when needed for computation. The same constant could be loaded multiple times in a method, which is required on a stack-based architecture. An average of 6% (see Figure 3) of original executed instructions are constant load instructions.

In order to make a fair comparison, we try:

—**Not** to perform optimizations that do anything other than take advantage of the register architecture. Such optimizations would give the register VM an unfair advantage over stack code.

—to keep the instruction set and their implementation in the interpreter the same except for the adaptation to the new instruction format and those differences mentioned in the Section 3.1.

An important question is whether the resulting comparison is fair. If we applied the same optimizations to the stack code, would it also be improved? In fact, the Soot optimization framework [Vallée-Rai et al. 1999] was used to translate stack JVM code to three-address code. They applied more aggressive optimizations than we use, and translated the resulting code back to stack JVM code. In order to achieve any improvements in running time, inter-procedural optimizations (which we do not perform) were required. They concluded that *intra*-procedural optimizations generally have very little effect on Java bytecode, on the basis that these can only work on scalar operations. This strongly suggests that the differences in performance we measure are the result of inherent differences between stack and register code, rather than the result of applying optimizations to one and not the other.

A similar question could be asked about the quality of the register code. If we were to design a register machine from scratch and generate code for it from source, we might produce a more efficient VM implementation. However, it is essential to our comparison that there are as few differences as possible between the stack and register VM. Otherwise, our results might be affected by other implementation issues.

A brief description of the optimizations is as follows:

—Copy propagation: Copy propagation [Muchnick 1997] is applied to eliminate `move` instructions in basic blocks. The stack pointer is used to find out whether an operand on the stack is alive or dead. Forward copy propagation is used to eliminate operand stack loads and backward copy propagation is used to eliminate operand stack stores.

—Global redundant load elimination: An immediate dominator tree is used to discover and eliminate redundant constant load instructions and other common subexpressions globally.

138 a L



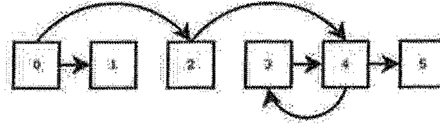



Fig. 4. The control flow of the example.

```
public int hashCode()
{
    if (cachedHashCode != 0)
      return cachedHashCode;

    int hashCode = 0;
    int limit = count + offset;
    for (int i = offset; i < limit; i++)
      hashCode = hashCode * 31 + value[i];
    return cachedHashCode = hashCode;
}
```

Fig. 5. Source code for the `hashCode()` method in the `java.lang.String`(GNU Classpath 0.90) class.

—Loop invariant motion: An immediate dominator tree and loop information are used to discover and move constant load instructions and other loop-invariant instruction out of loops.

### 3.4 Putting it all together

The runtime process for translating stack bytecode and optimizing the resulting register instructions for a Java method are as follows:

(1) Translate original bytecode into virtual register intermediate representation and build a factored control flow graph [Choi et al. 1999]
(2) Apply local copy propagation on basic blocks [Muchnick 1997]
(3) Build a dominator tree [Lengauer and Tarjan 1979] and enhance the intermediate representation with SSA form [Cytron et al. 1991]
(4) Remove dead code [Cytron et al. 1991]
(5) Apply global copy propagation
(6) Apply global redundant load elimination
(7) Apply loop invariant code motion [Muchnick 1997]
(8) Virtual register allocation [Mössenböck 2000; Briggs et al. 1994] and remove SSA $\phi$ functions
(9) Write the optimized register code into virtual register bytecode in memory.

To demonstrate the effect of the optimizations, we present the following example (see Figure 5 for Java source code and Figure 6 for corresponding bytecodes) with 6 basic blocks and one loop (see Figure 4 for its control flow graph). In Figure 6:

—The VM instruction operands with # are immediate operands.
—Virtual register numbers are prefixed with the initial r.
—Field identifiers are shown using the names of the fields.

```
Stack VM Code
Basic block(0):
01. ALOAD_0
02. GETFIELD cachedHashCode
03. IFEQ basic_block_2

Basic block(1):
04. ALOAD_0
05. GETFIELD cachedHashCode
06. IRETURN

Basic block(2):
07. ICONST_0
08. ISTORE_1
09. ALOAD_0
10. GETFIELD count
11. ALOAD_0
12. GETFIELD offset
13. IADD
14. ISTORE_2
15. ALOAD_0
16. GETFIELD offset
17. ISTORE_3
18. GOTO basic_block_4

Basic block(3)
19. ILOAD_1
20. BIPUSH #31
21. IMUL
22. ALOAD_0
23. GETFIELD value
24. ILOAD_3
26. CALOAD
27. IADD
28. ISTORE_1
29. IINC 3, #1

Basic block(4)
30. ILOAD_3
31. ILOAD_2
32. IF_ICMPLT basic_block_3

Basic block(5)
33. ALOAD_0
34. ILOAD_1
35. DUP_X1
36. PUTFIELD cachedHashCode
37. IRETURN
```

```
Register VM Code
Basic block(0):
02. GETFIELD r0.cachedHashCode -> r1
03. IFEQ r1 basic_block_2

Basic block(1):
05. GETFIELD r0.cachedHashCode -> r1
06. IRETURN r1

Basic block(2):
20. ICONST #31 -> r1
07. ICONST_0 -> r6
10. GETFIELD r0.count -> r2
12. GETFIELD r0.offset -> r3
13. IADD r2 r3 -> r2
16. GETFIELD r0.offset -> r7
18. GOTO basic_block_4

Basic block(3)
21. IMUL r6 r1 -> r3
23. GETFIELD r0.value -> r5
26. CALOAD r5 r7 -> r5
27. IADD r3 r5 -> r6
29. IINC r7 #1 -> r7

Basic block(4)
32. IF_ICMPLT r7 r2 basic_block_3

Basic block(5)
36. PUTFIELD r6 -> r0.cachedHashCode
37. IRETURN r6
```

Fig. 6. Original stack VM code and corresponding register VM code for the `hashCode()` method in the `java.lang.String`(GNU Classpath 0.90) class.

—In each instruction, the register number after -> is the destination register.
—The stack VM instructions are numbered 1 to 37.
—The instruction numbers in the register code show the stack instruction from which each register instruction originated.

All the local load and store VM instructions have been eliminated by the translation to register code. In Figure 6 the constant load instruction (number 20) is loop invariant and has been moved out of the loop to its preheader. A total of 37 VM instructions has been reduced to just 19. Most importantly, the number of VM instructions in the loop (basic blocks 3 and 4) has been reduced from 13 to 6.

138an



Table I. Hardware and Software Configuration

| Processor | OS | Compiler |
| --- | --- | --- |
| AMD Athlon(tm) 64 X2 Dual Core Processor 4400+ | Linux 2.6.14 | GCC 4.0.3 |
| Intel(R) Pentium(R) 4 CPU 2.26GHz | Linux 2.6.13 | GCC 2.95 |
| DEC Alpha 800MHz 21264B | Linux 2.6.8 | GCC 3.3.5 |
| Motorola MPC 7447a (PowerPC) 1066MHz | Linux 2.6.18 | GCC 4.0.2 |
| Intel(R) Core(TM)2 CPU 2.13GHz | Linux 2.6.18 | GCC 3.2.3 |

## 4. EXPERIMENTAL EVALUATION

### 4.1 Setup

For the present work, we used Cacao 0.95 (interpreter only with JIT disabled) as a base VM to implement the virtual register machine[4]. Cacao, released under the GPL, uses GNU Classpath as its class library and has a Boehm-Demers-Weiser garbage collector. Additionally, since version 0.93, Cacao has included a vmgen [Ertl et al. 2002] interpreter generator, which is used to define the virtual register machine instruction set and generate the interpreter. Both the virtual register and virtual stack interpreters support inline-threaded [Piumarta and Riccardi 1998; Ertl et al. 2006], direct-threaded, token-threaded, and switch dispatches.

We use the SPECjvm98 client benchmarks [SPEC 1998] (size 100 inputs) and Java Grande [Bull et al. 2000] (Section 3, data set size A). Methods are translated to register code the first time they are executed; thus all measurements in the following analysis include only methods that are executed at least once. The measurements include both the benchmark program code and the Java library code (GNU Classpath 0.90) executed by the VMs.

Table I shows the hardware and software configuration for the experiments. In the rest of the paper, we refer to these different processor architectures as AMD64, Intel Pentium 4, Alpha, PPC, and Intel Core 2 Duo. The choices of GCC compilers on different hardware platforms are based on their availability. We tested different GCC versions and selected the one that generated the interpreter with the best performance for each platform. We tried to make sure that the frequently used variables, such as the virtual IP, the virtual SP and the frame stack pointer are located in processor registers. Moreover, we had to avoid certain GCC versions because of GCC bugs: PR15242 and PR25285. These bugs could degrade the performance of interpreters up to 300% because of some optimizations performed on computed `gotos`. These bugs increase the dispatch cost of the interpreter. We believe that the increase in dispatch cost will affect stack-based VMs more than register-based VMs.

### 4.2 Static Instruction Analysis of Register Code

Figure 7 shows the breakdown of statically appearing VM instructions after converting to register code (translating and optimizing). On average 1.8% of VM

[4]Cacao changes different types of constant instructions (such as `iconst_0` and `iconst_1`) into one generic one (such as `iconst #immediate`). In order to make a fair comparison between stack and register implementations, we retain all those forms of constant instructions, forgoing this default Cacao transformation.

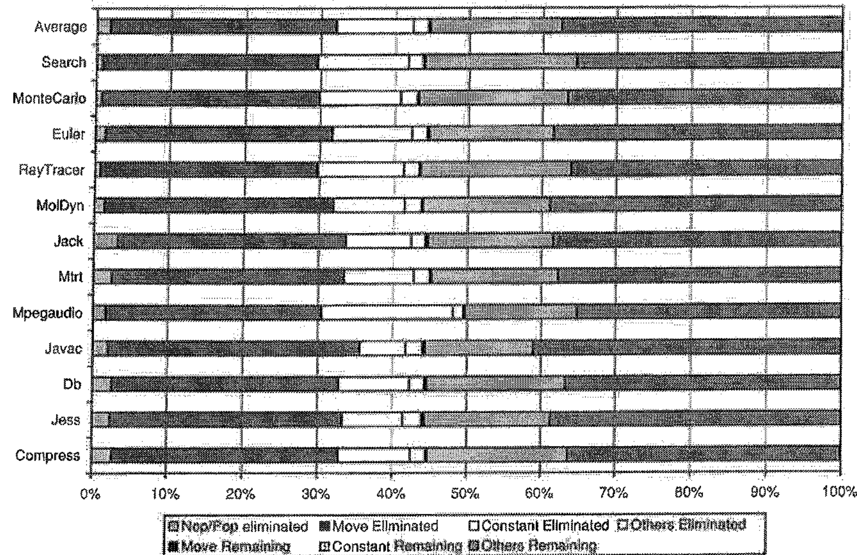



Fig. 7. Breakdown of statically appearing VM instructions before and after optimization for all the benchmarks.

instructions are pop or pop2 instructions. These can simply be translated to `nop` instructions in the register VM and eliminated, because they move the stack pointer, but do not move any values or perform any computation. A significant number of statically appearing `move` instructions are eliminated. Originally, `move` instructions account for 31% of VM instructions, but this is reduced to only 0.32% (of the original instructions) after translation. Similarly, optimization results in the elimination of constant load instructions, from an average of 28% of total statically appearing VM instructions down to 18% (of the original instructions) after translation. This is achieved by reusing commonly occurring constants, thus allowing the constant loads to be removed. Eliminating other common subexpressions allows a further 2.1% of static VM instruction to be optimized away. Overall, an average of 44% of static VM instructions are eliminated.

### 4.3 Stack Frame Space

Each method in the stack JVM has both a set of local variables and an operand stack. In order to perform computations, values must be copied from the local variables to the operand stack. Thus, within the interpreter, the stack frame for each method must contain two separate regions for local values which cannot be used interchangeably. The register VM, on the other hand, has only a single, unified set of registers which can both store local values and be used to perform operations on those values. Thus, there is potential for the register VM to require fewer slots in the stack frame than the stack VM.

As part of the translation from stack to register code we apply a simple graph-colouring register allocation to pack the values which were previously split

138ap



Table II. The Comparison of Required Stack/Local Variable Slots (Virtual Registers) Between Stack and Register Architectures

| Benchmark | Stack | Register without redundant load elimination | Register with redundant load elimination |
|---|---|---|---|
| Compress | 5.29 | 3.86 | 4.17 |
| Jess | 5.13 | 3.74 | 4.03 |
| Db | 5.33 | 3.89 | 4.20 |
| Javac | 6.34 | 4.72 | 5.02 |
| Mpegaudio | 5.56 | 4.14 | 5.37 |
| Mtrt | 5.38 | 3.97 | 4.28 |
| Jack | 5.19 | 3.83 | 4.26 |
| MolDyn | 5.62 | 4.14 | 4.49 |
| RayTracer | 5.60 | 4.07 | 4.32 |
| Euler | 5.57 | 4.09 | 4.44 |
| MonteCarlo | 5.31 | 3.90 | 4.13 |
| Search | 5.34 | 3.85 | 4.26 |
| Average | 5.47 | 4.02 | 4.41 |

between the locals and the evaluation stack into a smaller number of virtual registers. Table II shows the average number of stack frame slots required in a method for the locals and operand stack in the stack machine, and for the virtual registers in the register machine. On average, methods for the stack VM require 5.47 slots. The corresponding number for our register VM code is 4.61. It is important to note, however, that the register VM code normally has more live values. Eliminating redundant constant load instructions will keep more variables alive at the same time, which means more virtual registers are required. If we do not apply these redundancy elimination optimizations, we find that the register machine needs an average of only 4.02 slots. Even though a smaller stack frame size has little impact on the execution time of the VM interpreter, it may be beneficial to embedded or other small devices with tight memory constraints.

### 4.4 Dynamic Instruction Analysis of Register Code

In order to study the dynamic (runtime) behaviour of our register JVM code, we counted the number of VM instructions executed in the stack and register VMs. Figure 8 shows the breakdown of VM instructions dynamically executed before and after converting to register code.

(1) The biggest category of eliminated instructions is `move` instructions, accounting for a much greater percentage (42%) of executed VM instructions than static ones (30%). The remaining `moves` account for only 0.28% of the original VM instructions executed.
(2) The second largest category of executed instruction elimination is constant load instructions (3.5% on average), which is much lower than the constant load instruction elimination (10% on average) in static code. The remaining dynamically executed constant VM instructions account for 2.9%. However, there are far more remaining constant load instructions (18%) in static

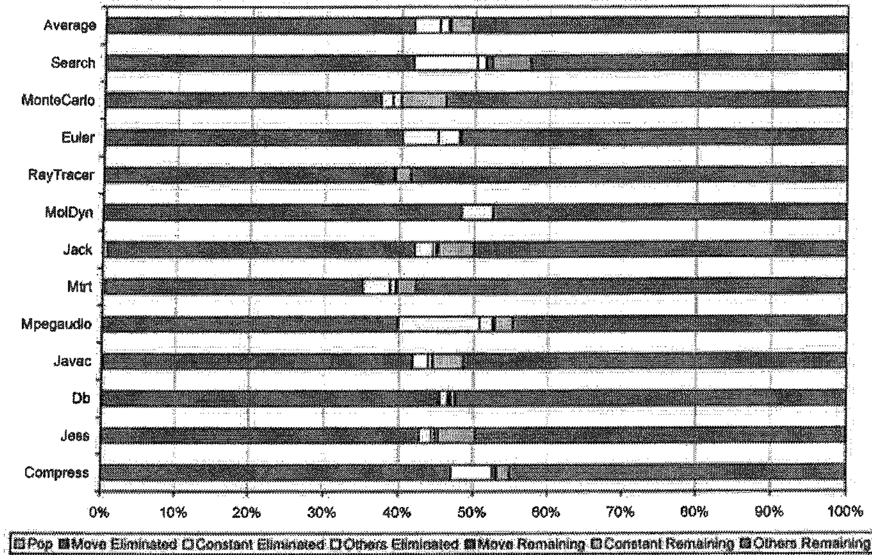



Fig. 8. Breakdown of dynamically appearing VM instructions before and after optimization for all the benchmarks.

code than those dynamically run (2.9%) in the benchmarks. We discovered that there are a large number of constant instructions in the initialization bytecode which are usually executed only once.

(3) Elimination of other instructions accounts for 1.2% of VM instructions executed while the static elimination is an average of 2.1%.

(4) Elimination of pop/pop2 only contributes to a 0.14% reduction in dynamically executed instructions.

Overall, we eliminate an average of 46% of dynamically executed VM instructions. In general, copy propagations of move instructions produce the most effective result. Other optimizations are more dependent on the characteristics of the particular program. For example, in the benchmark *moldyn*, eliminated constant load VM instructions account for only 0.11% of total executed instructions, although such instructions account for 10% of static instructions.

### 4.5 Code Size

The register VM code size is usually larger than that of a stack VM. There are actually two effects in action here. Register machine instructions are larger than stack instructions because the locations of the operands must be expressed explicitly. On the other hand, register machines need fewer VM instructions to do the same work, so there are fewer VM instructions in the code. Figure 9 shows the increase in code size of our register machine code compared to the original stack code. On average, the register code size is 26% larger than that of the original stack code, despite the fact that the register machine requires 44% fewer static instructions than the stack architecture. This is a significant

138ar



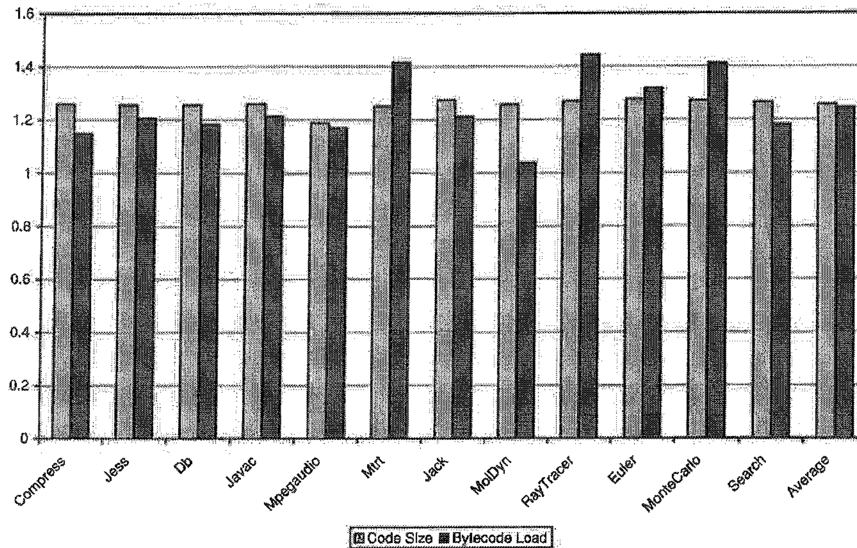



Fig. 9. Fractional increase in total code size (static) of executed methods and bytecode loads (dynamic) for register over stack architecture.

increase in code size, but it is far lower than the 45% increase reported by Davis et al. [2003].

Although static code size is an important issue, we also consider the size increase in executed bytecode when moving from stack to register VMs. This increase in executed bytecode is a direct result of the (static) increased code size of the register JVM. Because of this, more VM instruction bytecodes (both opcodes and operands) must be loaded, on average, from memory as the program is interpreted. Figure 9 also shows the resulting increase in bytecode loads. Interestingly, the increase in overall code size is often very different from the increase in instruction bytecodes loaded in the parts of the program that are executed most frequently. Nonetheless, the average increase in loads (25%) is similar to the average increase in code size (26%). An alternative to fetching each operand location separately is to use a four-byte VM instruction containing the opcode and three register indices. This entire VM instruction could be fetched in a single load. However, it would still be necessary to extract the opcode and register numbers inside the four-byte VM instruction. This would involve shifting and masking the loaded VM instruction. Clearly the cost of such operations varies from one processor to another. For example the Northwood-core Pentium 4 has no barrel shifter, so large shifts are expensive. In general, if a piece of code loads four successive bytes and does something with them, most compilers generate separate byte loads, rather than a single word load and using shifts and masks to extract the bytes[5].

[5]Preliminary experiments by the authors on the Pentium IV suggest that shifting/masking is the slower approach on that particular architecture.

138qs



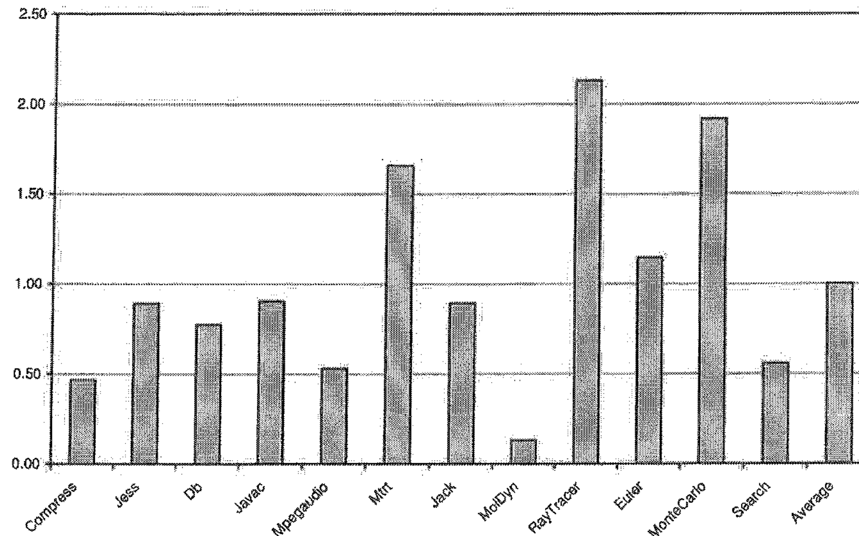



Fig. 10. Increase in dynamically loaded bytecode instructions per VM instruction dispatch eliminated by using a register rather than stack architecture. In other words, if the register VM uses one less dispatch, how many extra bytes of bytecode must it load?

Despite the cost of increased bytecode loads due to larger code, the fact that that fewer VM instructions are needed by the register VM gives a significant performance advantage. To measure the relative importance of these two factors, we compared the number of extra dynamic bytecode loads required by the register machine per dynamically executed VM instruction eliminated. Figure 10 shows that the number of additional byte loads per executed VM instruction eliminated is small at an average of only 1.00 loads. On most architectures, even one CPU load costs much less to execute than an instruction dispatch, with its difficult-to-predict indirect branch. This strongly suggests that register machines can be interpreted more efficiently on most modern architectures.

### 4.6 CPU Loads and Stores

Apart from CPU loads of instruction bytecodes, the main source of CPU loads in a JVM interpreter comes from moving data between the local variables and the stack. In most interpreter-based JVM implementations, the stack and the local variables are represented as arrays in memory. Thus, moving a value from a local variable to the stack (or vice versa) involves both a CPU load to read the value from one array, and a CPU store to write the value to the other array. A simple operation such as adding two numbers can involve large numbers of CPU loads and stores to implement the shuffling between the stack and registers.

In our register machine, the virtual registers are also represented as an array. However, VM instructions can access their operands in the virtual register array directly, without first moving the values to an operand stack array. Thus, the virtual register machine can actually require fewer CPU loads and stores

138at



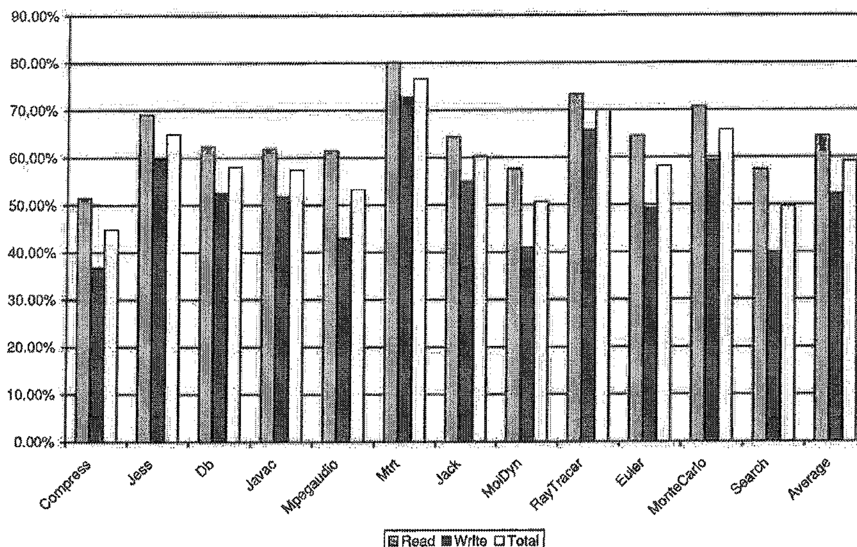



Fig. 11. Dynamic number of CPU loads and stores required to access virtual registers in our virtual register machine, expressed as a percentage of the corresponding loads and stores used to access the stack and local variables in a virtual stack machine.

to perform the same computation. Figure 11 shows (a simulated measure of) the number of dynamic CPU loads and stores required for accessing the virtual register array, as a percentage of the corresponding loads and stores for the stack JVM to access the local variable and operand stack arrays. The register VM requires only 65% as many CPU loads and 52% as many CPU writes, with an overall figure of 59%.

In order to compare these numbers with the number of additional loads required for fetching instruction bytecodes, we express these memory operations as a ratio to the dynamically executed VM instructions eliminated by using the virtual register machine. Figure 12 shows that on average, the register VM requires 1.74 fewer CPU memory operations to access such variables per instruction dispatch eliminated. This is much larger than the number of additional loads required due to the larger size of virtual register code (1.00). Thus, the interpreter for the register VM would execute fewer memory operations overall.

However, these measures of memory accesses for the local variables, the operand stack and the virtual registers depend entirely on the assumption that they are implemented as arrays in memory. In practice, we have little choice but to use an array for the virtual registers, because there is no way to index CPU registers like an array on most real architectures. However, stack caching [Ertl 1995] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated CPU loads and stores. For example, around 50% of stack access CPU memory operations could be eliminated by keeping just the topmost stack item in a register [Ertl 1995]. Thus, in many implementations the virtual register architecture is likely to need more CPU loads and stores to access these kinds of values.

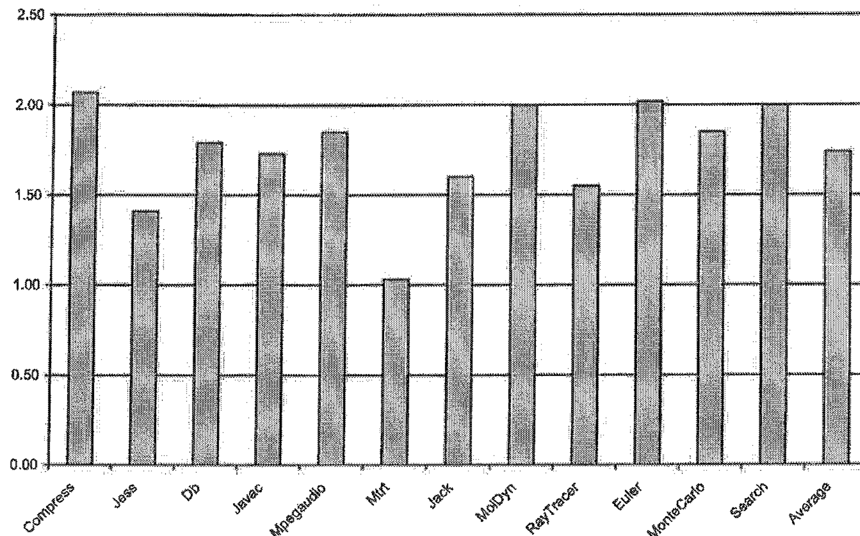



Fig. 12. The reduction of CPU memory accesses for each executed VM instruction eliminated by using a register VM rather than a stack VM. This is analogous to the measurement in Figure 10.

## 4.7 Timing Results

To measure the benchmark running times of the stack and register-based implementations of the JVM, we ran both VMs on AMD64, Intel Pentium 4, Intel Core 2 Duo, Alpha and PowerPC systems (See Table I). The stack JVMs simply interpret standard JVM bytecode. The running time for the register JVMs includes the time necessary to translate and optimize each method the first time it is executed. However, our translation routines are fast. In the version of the virtual register machine that uses token-threaded dispatch, the process of translation and optimization accounts for an average of only 0.8% of total execution time. As a result, we believe the comparison is fair. In our performance benchmarking, we run SPECjvm98 with a heap size of 70MB and Java Grande with a heap size of 160MB. Each benchmark is run independently.

We compare the performance of stack JVM interpreters and register JVM interpreters with four different dispatch mechanism: (1) switch dispatch, (2) token-threaded dispatch, (3) direct-threaded dispatch and (4) inline-threaded dispatch [Piumarta and Riccardi 1998] (see Section 2). For fairness, we always compare the performance of stack and register interpreter implementations which use the same dispatch mechanism.

Figure 13 shows the speedup in running time of our implementation of the virtual register machine compared to the virtual stack machine on the AMD64 machine using the various dispatch mechanisms. With `switch` dispatch, the register VM has the highest average speedup (1.48) because `switch` dispatch is the most expensive. Even with the efficient inline-threaded dispatch, the register VM still has an average speedup of 1.15.

138av



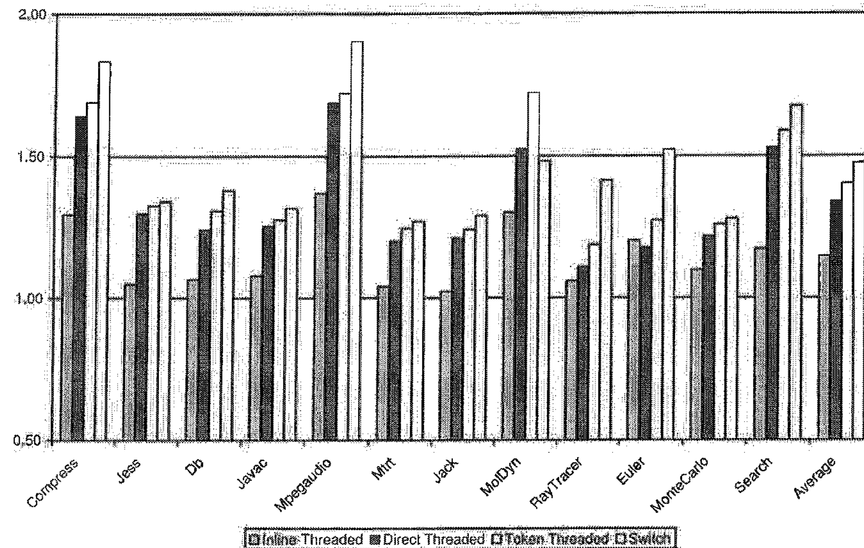



Fig. 13. AMD64: register VM speedups over stack VM of same dispatch (based on average real running time of two runs).

Figure 14 shows the same figures for a Pentium 4 machine, whose processor utilizes a trace cache. With inline-threaded dispatch, the register VM has an average speedup of 1.00, and some benchmarks are very close to or worse than on the stack VM. The `switch` register VM has highest speedup (1.46). The `mtrt` benchmark performs very poorly for various dispatches, which may be due to high cost of threading using GCC 2.95 compiler.

Figure 15 shows the speedup of register VMs over stack VMs on the Intel Core 2 Duo processor. The average speedups of register over stack-based VMs are 1.15 (inline-threaded), 1.32 (direct-threaded), 1.29 (token-threaded), and 1.65 (switch).

Figure 16 shows the speedups of register VMs over stack VMs on the IBM PowerPC processor. The average speedups for the four dispatch mechanisms (inline-threaded, direct-threaded, token-threaded and switch) are 1.16, 1.30, 1.29, and 1.41 respectively.

Figure 17 shows the speedup of register VMs over stack VMs on the Alpha processor. The inline-threaded dispatch is not working for Alpha and there are still bugs which prevent `javac` from running correctly (and thus is excluded from the benchmark results). The average speedups are 1.22 (direct-threaded), 1.25 (token-threaded), and 1.64 (switch).

### 4.8 Performance Counter Results

To obtain a finer grained view of benchmark performance, we use AMD64 hardware performance counters to measure various processor events during the execution of the programs. Figures 18 and 19 show performance counter results for the SPECjvm98 benchmarks *Compress* and *Jack*. We measure the data cache accesses, data cache misses, instruction cache fetches, instruction cache misses,

138aW



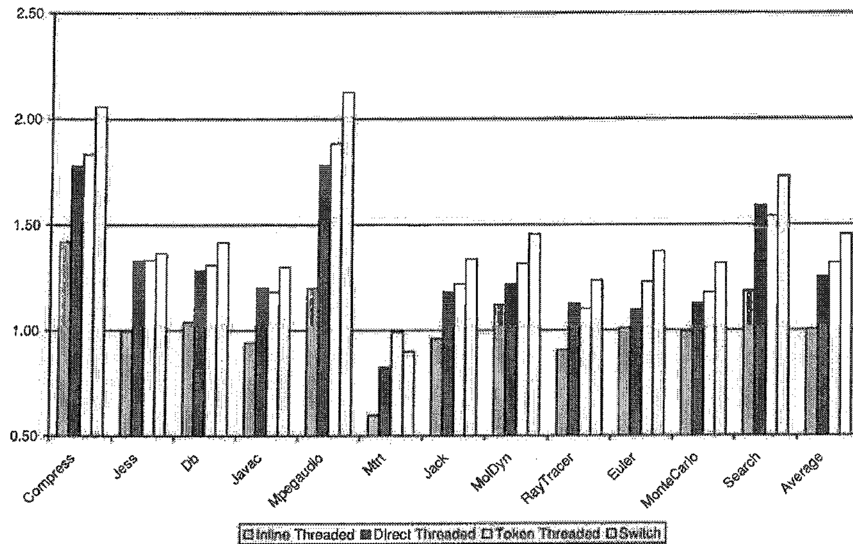



Fig. 14. Intel Pentium 4: register VM speedups over stack VM of same dispatch (based on average real running time of five runs).

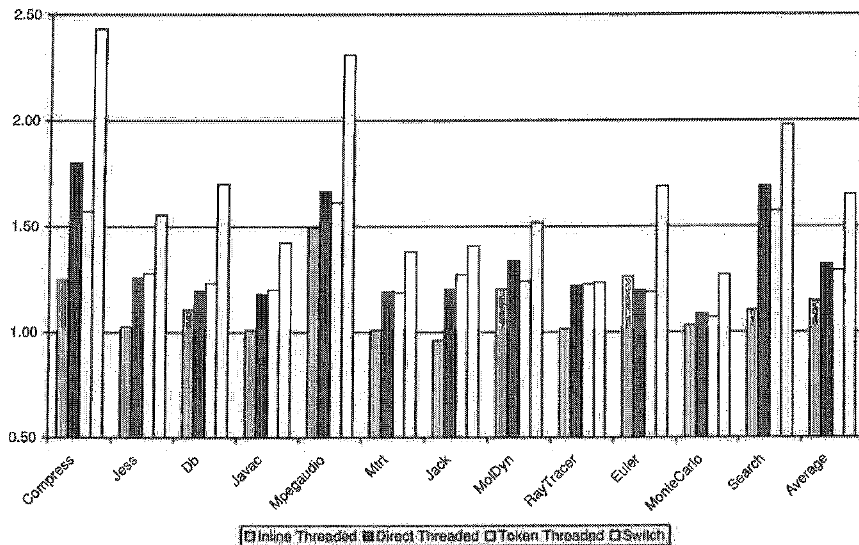



Fig. 15. Intel Core 2 Duo: register VM speedups over stack VM of same dispatch (based on average real running time of three runs).

retired taken branches (which include indirect branches; unfortunately there is no way to measure indirect branches alone by using AMD64's performance counters), retired taken branches mispredicted (indirect branches are the main source of misprediction), and retired instructions[6].

[6]On out-of-order processors, a retired instruction is one that has been executed and completed.

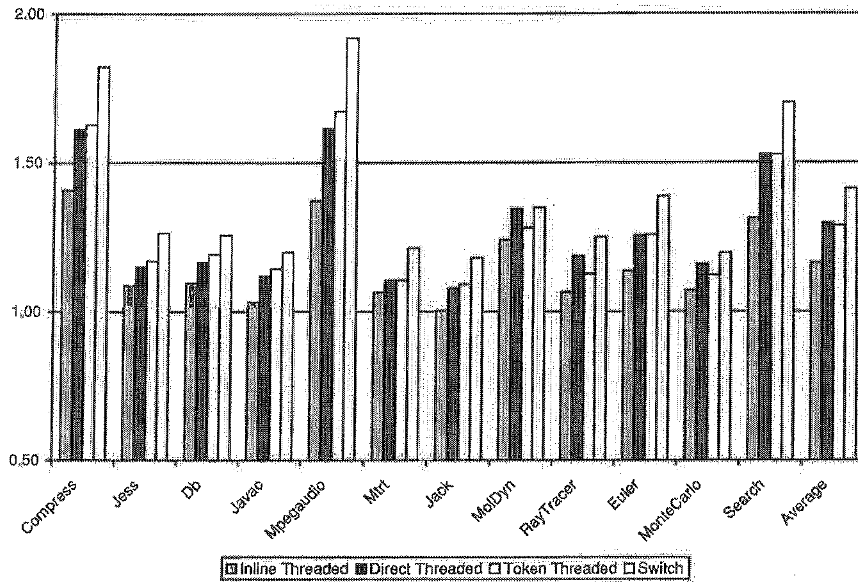



Fig. 16. IBM PowerPC: register VM speedups over stack VM of same dispatch (based on average real running time of three runs).

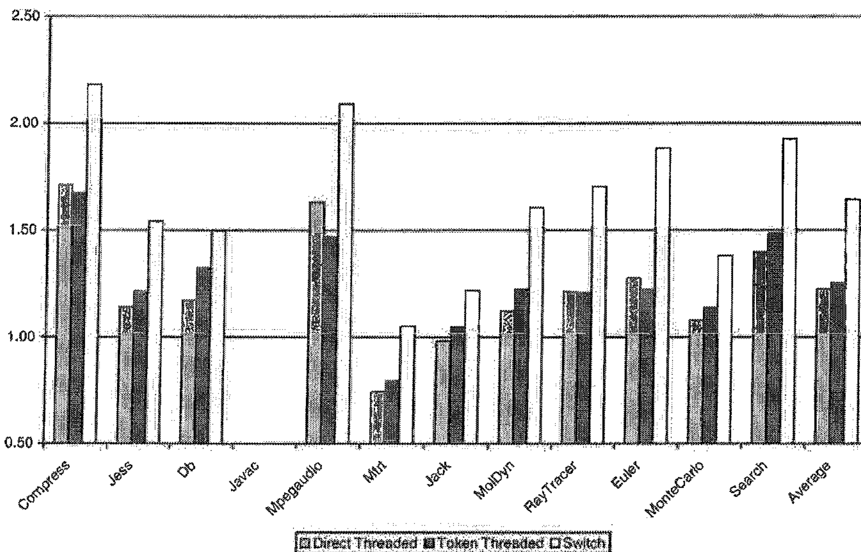



Fig. 17. Alpha: register VM speedups over stack VM of same dispatch (based on average real running time of five runs).

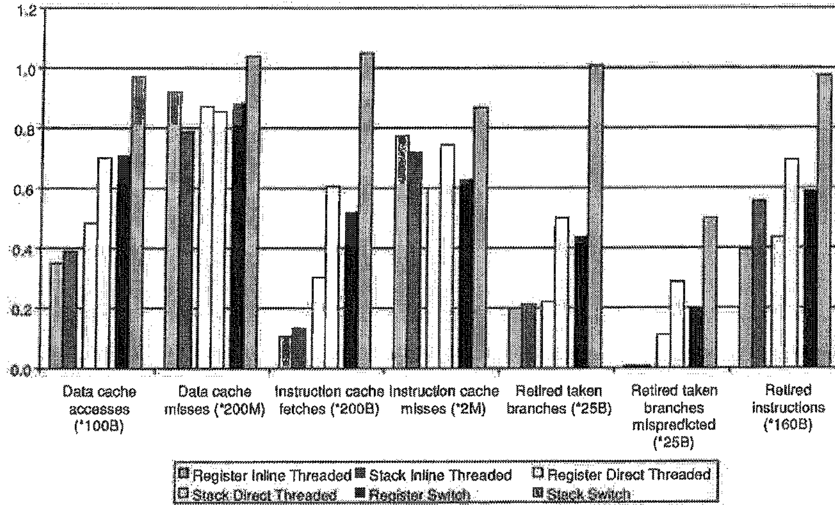



Fig. 18. Compress: AMD64 performance counters.

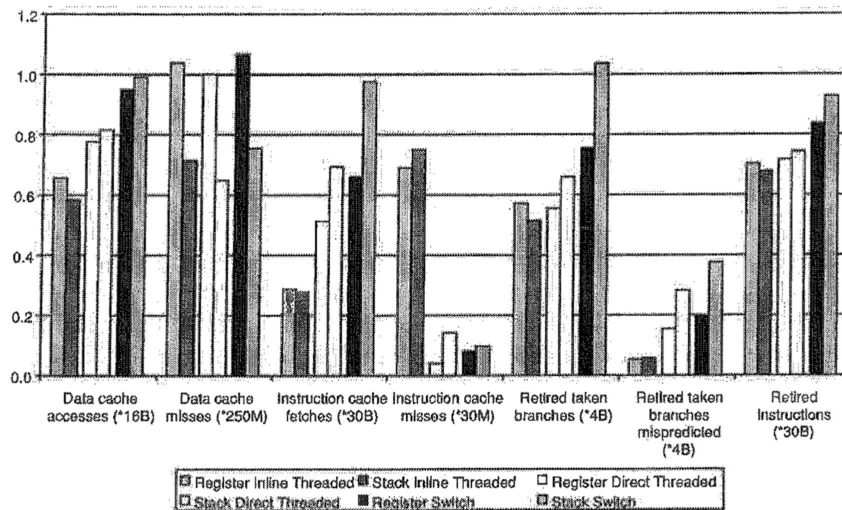



Fig. 19. Jack: AMD64 performance counters.

Figure 18 shows the measured performance counters for inline-threaded, direct-threaded and switch dispatches for the `compress` benchmark. From Figure 8, we know that, for the `compress` benchmark, 54% of executed VM instructions are eliminated compared to the stack architecture. As the dispatch method becomes more efficient, the difference between corresponding performance counters for register VMs and stack VMs becomes smaller. For inline-threaded dispatch, retired taken branches are almost the same for register and stack VMs. The main source of advantage is fewer retired instructions, which gives the register VM a speedup of 1.15 over the stack VM for inline-threaded dispatch.

138az



For the compress benchmark, the register version of the machine always executes fewer real machine instructions. As we saw in Figure 10, translation to register format actually results in less than 0.5 extra bytecode loads per VM instruction eliminated. However, compress is the benchmark with the greatest reduction in real machine memory operations for manipulating local values (see Figure 11). This accounts for the much lower number of retired real machine instructions.

Figure 19 shows the measured performance counters of the jack benchmark for inline-threaded, direct-threaded and switch dispatches. From Figure 8, we know that 44% of executed instructions are eliminated from jack in the register VM. The data cache miss ratio and instruction cache miss ratio are much higher than those of the compress benchmark. For inline-threaded dispatch, the register VM shows more data cache accesses, data cache misses, retired taken branches, and retired instructions than those of the stack VM. Nonetheless, instruction cache misses and retired taken branches mispredicted are lower. The inline-threaded dispatch speedup of register VM over stack VM for the jack benchmark is only 1.02. For inline-threaded dispatch, both stack and register VMs show very high numbers of instruction cache misses when compared with other dispatch mechanisms because of binary executable code replication.

### 4.9 Dispatch Comparison

All the comparisons to this point have been between stack and register architecture pairs using the same dispatch mechanism. For example, we have shown performance of the register VM interpreter using token-threaded dispatch as a speedup over the performance of the corresponding stack VM. In this section, we compare differences between the dispatch mechanisms. The performance of the stack VM interpreter using switch dispatch is the baseline value (speedup=1.0) and all other variants are shown relative to that value (see Figure 20). Sun's JDK 1.6.0 (interpreter mode only) gives an indication of the speed of Cacao's stack and register VMs and should be treated with caution because of the different implementation.

We see that the more complex, less portable dispatch mechanisms give the greatest speedups. We also observe that, at least for the benchmark results presented, the register machine has a significant edge. For example, if one has to choose between using direct-threaded dispatch on a stack VM and switch dispatch on a register VM, it should be noted that there is little difference in execution speed between the two implementations. However, switch dispatch is simpler to implement and much more portable. Furthermore interpreted bytecode is a fraction (typically 25%–50%) of the size of threaded code, so there is also a significant space saving. Therefore, the register VM interpreter with switch dispatch is preferable. If one is purely concerned with code size, a token-threaded stack VM is still the most compact option.

### 4.10 Discussion

Although our implementation of the register JVM translates the stack bytecode into register bytecode at runtime, we do not envision this in a real-life

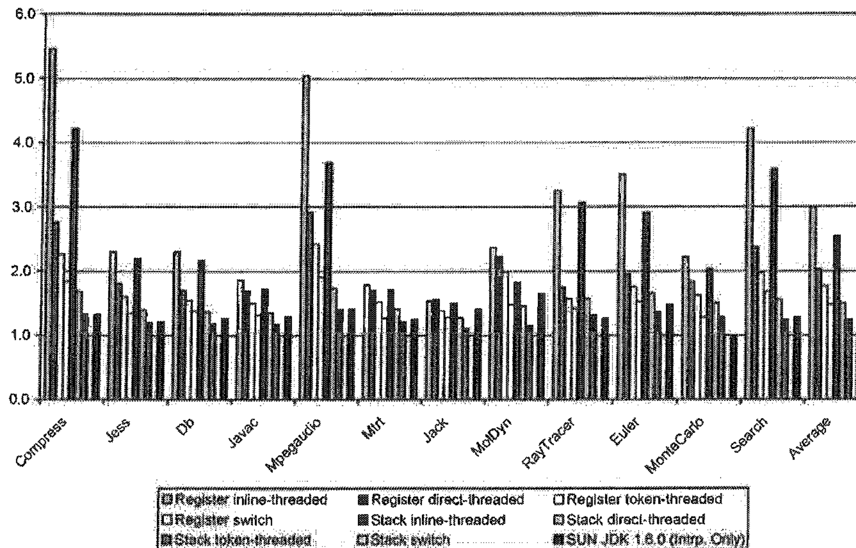



Fig. 20. AMD64: speedups over the stack switch interpreter.

implementation. The purpose of our implementation is to evaluate a virtual register JVM against an equivalent stack based one. Our register JVM implementation originates directly from a modification of a stack-based JVM implementation, thus giving us two VMs identical in every other regard. Apart from the necessary adaptation of the interpreter loop, along with some garbage-collection and exception handling modifications, there are very few changes to the original code segments responsible for interpreting bytecode instructions. The objective is to provide a fair comparison between the stack-based JVM and the register-based JVM.

Given a computation task, a register VM inherently needs far fewer instructions than a stack VM does. For example, our register JVM implementation can reduce the static number of bytecode instructions by 44% and the dynamic number of executed bytecode instructions by 46% when compared to those of the stack JVM. The reduction of executed bytecode instructions leads to fewer real machine instructions for the benchmarks and a significantly smaller number of indirect branches. These indirect branches are very costly when they are mispredicted. Moreover, the elimination of large numbers of stack load and store (`move`) instructions reduces the number of loads and stores in a real processor. In terms of running time, the benchmark results show that our register JVM still outperforms an equivalent stack JVM, even when both are implemented using the most efficient dispatch mechanism. This is a very strong indication that the register architecture can be implemented to be faster than the stack architecture.

An important question is whether we would generate better register code if we were to compile directly from Java source code, rather than translating from stack code. The *javac* compiler generates optimized stack code, but the

138bb



optimizations may not suit register code. Furthermore, eliminating (partially) redundant expressions in stack code is rarely worthwhile, because the common expression must be stored and later recovered, which is often more expensive than recomputing the expression. Although eliminating simple redundant computations in stack code is easy, we might find it easier to eliminate more redundancy if we were working from source code. In particular, eliminating some kinds of redundant expressions, such as those described in the next section, depends on pointer analysis to ensure that the transformation is safe. Pointer analysis may be also be easier to perform on source code rather than after its translation to register code.

## 5. MORE OPTIMIZATIONS

### 5.1 Redundant Heap Load Elimination

As we saw in Section 3.3, register machines can take advantage of redundant computations more easily than stack machines. This is because (unlike stack VMs) register VMs do not destroy operands to VM instructions as they use them. The results presented in Section 4 are for register machine code where redundant loads of constants and some simple common subexpressions involving local variables were eliminated.

There is another category of redundant loads—the loads from class or object fields and array elements, and there has been some work on eliminating these redundant loads in compilers [Fink et al. 2000]. However, it is very important to note that eliminating such loads from heap data structures requires sophisticated pointer alias analysis to ensure that the object or array element is not modified between apparently redundant loads [Diwan et al. 1998]. In particular, we need to know whether a reference to an object has escaped into another thread, which may modify the object. Alias analysis is complex and slow; we have not implemented it in our translation.

In order to examine the potential of the register machine to allow even more redundant loads to be eliminated, we performed some preliminary experiments without sophisticated alias analysis. Our very simple analysis is not safe—in particular it does not check for references escaping to another thread, but it allows us to get *some* idea of the potential benefit from register machines exploiting this sort of redundancy.

Figure 21 shows that an average of 5% of original executed VM instructions can be eliminated by removing redundant `getfield` VM instructions. The corresponding figure for array loads is 2%. All benchmarks benefit from redundant `getfield` elimination, while only a few benchmarks benefit from redundant array load elimination. In the Euler benchmark, eliminated redundant array loads account for 13% of original executed VM instructions. After all optimizations, the register machine requires only 23% of the original stack machine instructions.

Figure 22 shows the same dispatch speedup results for the AMD64. The average speedup for inline-threaded goes from 1.15 to 1.29 and that of switch dispatch from 1.48 to 1.74 as this optimization is added.

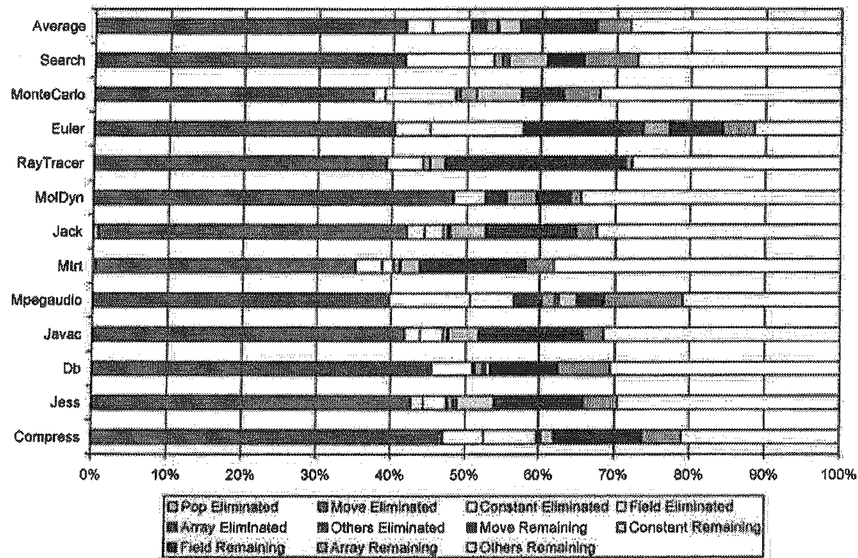



Fig. 21. Breakdown of dynamically appearing VM instructions with additional redundant heap load elimination for all the benchmarks. These results are indicative only, because our translator makes unsafe assumptions about aliasing.

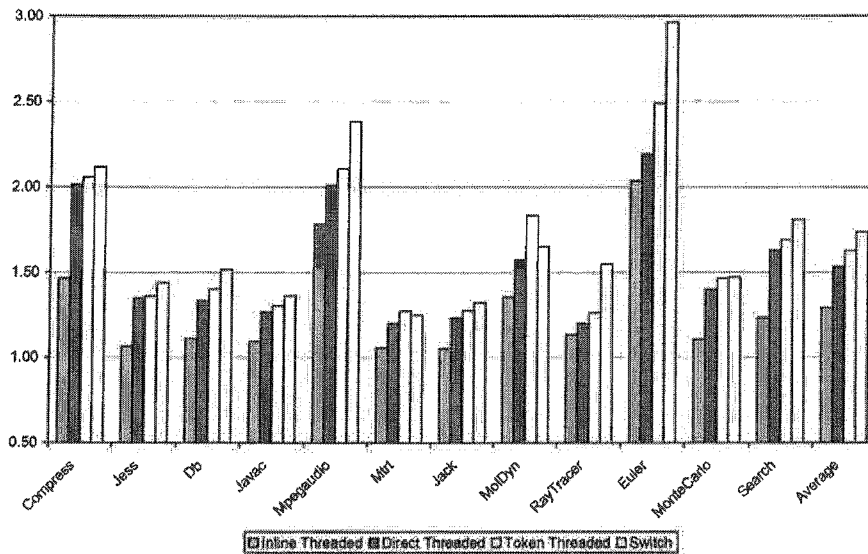



Fig. 22. AMD64: Register VM speedups with additional redundant heap load elimination (based on average real running time of two runs). These results are indicative only, because our translator makes unsafe assumptions about aliasing.

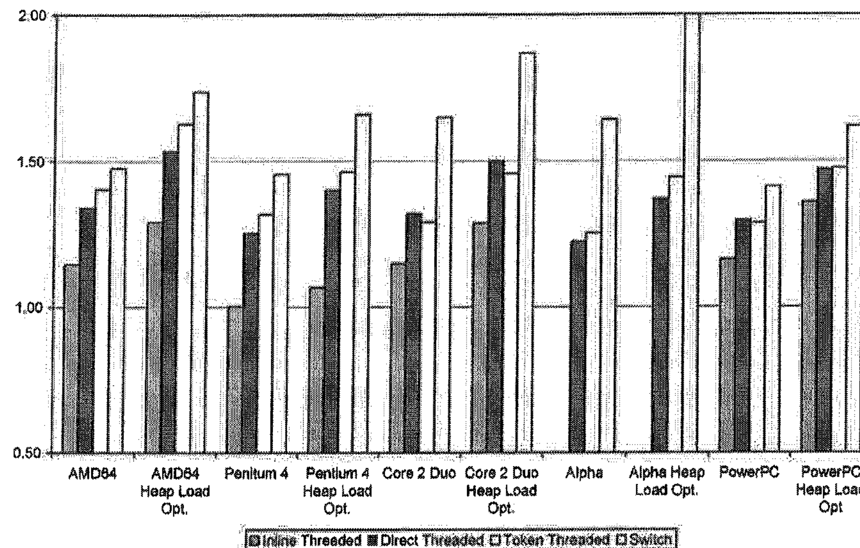



Fig. 23. The average speedups of register VM over stack VM using the same dispatch for different processors. The results which include heap load optimization are indicative only, because our translator makes unsafe assumptions about aliasing.

Figure 23 summarizes the average speedups of register VM over stack VM using the same dispatches with/without redundant heap load elimination. The register VM could potentially benefit significantly from eliminating these loads, but a real implementation of this optimization would require very sophisticated alias and escape analysis.

## 5.2 Stack Caching for Stack VM

Stack caching [Ertl 1995] can be used to keep the topmost stack values in registers, and eliminate large numbers of associated CPU loads and stores. Take, for example, the real machine memory operations required for the operand stack access. Around 50% of these real machine memory operations could be eliminated by keeping just the topmost stack item in a register [Ertl 1995]. Figure 24 shows the speedup over the stack VM with/without stack caching[7] using the same dispatch mechanisms. Stack caching did show improvements for the stack VMs. This improvement results in the speedups of the register VM going from 1.16 and 1.30 (over stack VM with no caching) down to 1.14 and 1.26 (over stack VM with caching) for inline-threaded and direct-threaded dispatch respectively.

## 5.3 Static Superinstructions

One way to reduce the number of VM interpreter dispatches is to add static *superinstructions* to the instruction set of the VM. These are new VM

[7]We can only present the results of caching the topmost stack item for inline-threaded and direct-threaded dispatches

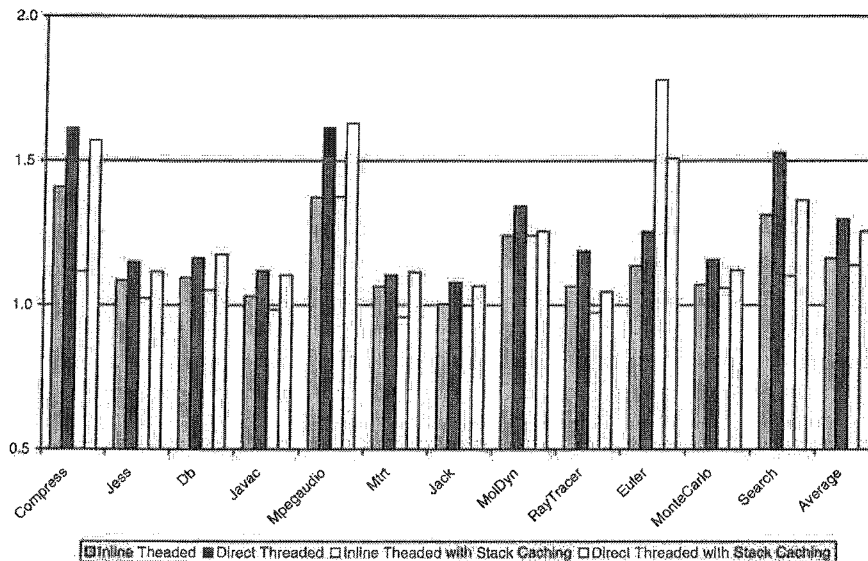



Fig. 24. PowerPC: register VM speedups over stack VM (with and without stack caching) of same dispatch (based on average real running time of two runs).

instructions that behave in the same way as a sequence of regular VM instructions. For example, if one found that `aload` VM instructions are often followed directly by a `getfield` VM instruction, one might introduce an `aload-getfield` superinstruction. Wherever this sequence appears in the program, it can be replaced by the superinstruction, reducing the number of dispatches. It has been argued that superinstructions can achieve the same effect as translating to a register machine, without the damaging increases in VM code size. In fact, this is not achievable in practice.

The main problem with superinstructions is choosing appropriate sequences. The superinstructions must be hardwired into the interpreter, at a time when the program to be run is usually unknown. Perhaps the best strategy for selecting sequences is to look at a large variety of programs and identify the most important sequences of VM instructions in those programs. Eller [2005] investigated using SPECjvm98 benchmarks to select superinstructions using a wide variety of selection strategies. He found that superinstructions could be added to a stack-based VM which would reduce the number of dispatches by up to 40%. However, to achieve that reduction, 1000 superinstructions were needed. This means that it is no longer possible to encode the instruction opcode in a single byte. Furthermore, the interpreter code to implement the superinstructions becomes significantly larger than that of the original interpreter. In contrast, the register machine does not require any additional VM instructions.

### 5.4 Two-Address Instructions

Our register JVM uses a three-address instruction format for arithmetic instructions. An obvious way to reduce code size would be to use a two-address

1386f



instruction format for these instructions instead, where one of the source registers would also be the target register of the instruction. This would reduce the size of these instructions from four bytes (one opcode and three register indices) to only three bytes. We investigated this possibility, but found that arithmetic instructions account for only an average of only 6.3% of statically appearing register VM instructions in the SPECjvm98 benchmarks. Thus, the overall reduction in code size from two-address instructions is likely to be small. Furthermore, there are some disadvantages with two-address instructions. They make sharing of common subexpressions more difficult, because one of the input values is overwritten by the output of the instruction. Additional `move` instructions must be introduced (or retained) to prevent values from being destroyed, which would both increase code size and reduce the efficiency of the VM. A more complicated allocation of variables to registers would also be needed to minimize the number of `move` operations introduced. Given that the potential reduction in code size was small anyway, we decided that this optimization was not worthwhile.

## 6. APPLICABILITY OF RESULTS TO RELATED QUESTIONS

Although our experiments in this work have been limited to the JVM, we believe that the results will extend to other VMs which employ an interpreter. Already, the conversion of the Lua VM from stack machine to register machine (the upgrade from version 4.0 to version 5.0) has yielded a substantial improvement in performance. Ierusalimschy et al [2005] have compared the stack machine implementation of version 4.0 to an equivalent register machine implementation (with no additional optimizations). Across their selected benchmarks, the register machine was an average of 1.30 times faster than the stack machine. More significantly, on the benchmark they specifically selected to test the execution engine, a speedup of 2.28 was reported.

The main benefit of the transition from stack VM to register VM is the reduction in the number of VM instruction dispatches, and consequently a reduction in branch mispredictions. There are other benefits which we have observed such as a reduction in real machine instructions at the CPU level. For coarse-grained VMs with higher-level instruction sets which have a significantly lower number of instruction dispatches to begin with, the transition to a register VM will not yield the same speedups. For example, Vitale and Abdelrahman [2004] found that inline threading had little benefit for a Tcl virtual machine, because each VM instruction performed a lot of work. Hence, dispatch accounted for only a small proportion of running time. It is likely that we would see similar results in a comparison of stack and register VMs for Tcl. Where the cost of dispatch is a small proportion of total time, there will be little benefit in any optimization to reduce dispatches.

Another interesting question is whether a stack or register VM is more suitable as a source language for JIT compilation. Winterbottom and Pike [1997] suggest that a register IR may be easier to compile to native code because it is closer to the register architecture used by real processors. Others argue that a stack machine is better, because stack code does not make assumptions about the number of available registers.

1386g



Unfortunately, our results apply only to interpreters, but we believe that JIT compiling from well-behaved stack architectures like the JVM is probably a little easier than from register architectures. This is because stack code is similar to the tree representations of expressions often used in real compilers. On the other hand, a register architecture allows more optimizations to be expressed, because common subexpressions can be eliminated in the register code, rather than relying on the JIT compiler to perform these kinds of optimizations.

However, an optimizing JIT compiler is typically a complex piece of software, and translating the VM bytecode to a format more useful to the compiler is likely to be only a small part of compilation regardless of whether a stack or register VM is used. On the other hand, our results indicate that for a simple code-inlining JIT, there are some significant gains to be made in choosing a register IR rather than a stack IR. For a more aggressive JIT, the choice is not so clear. Finally, for mixed mode JIT compilers such as Sun's Hotspot VM [Sun-Microsystems 2001], interpretation speed is still important, and therefore a register VM may be used to improve the performance of interpretation.

## 7. RELATED WORK

A large number of JIT compilers have been constructed for stack-based VMs such as the JVM. These include Cacao [Krall and Grafl 1997] and Jikes RVM [Arnold et al. 2002]. Our translator uses the same standard, well-known techniques that are used in these JIT compilers. The HotSpot JVM [Sun-Microsystems 2001] uses a mixed-mode interpreter and JIT compiler. Interpreting the large amount of rarely executed code in many Java programs avoids the time and memory overhead of compilation and can be faster than JIT compilation.

Myers [1977] attempts to refute the idea that stack machines will necessarily result in smaller code, with lower cost to access operands. The argument is based on measurements of real programs which show that the expression in most assignment statements is extremely simple. Thus, in most cases, operands must be loaded to the stack for use, rather than already being there as part of the evaluation of a complex expression. Beyond measurements of the complexity of expressions, Myers presents only a handful of small examples showing situations where register code is superior to stack code. Schulthess and Mumprecht [Schulthess and Mumprecht 1977] argue that Myers' work is inconclusive, because programs contain features other than expressions that are better expressed using stacks (e.g. subroutine calls, parameter passing and multitasking). No quantitative data is provided.

The controversy between stack and register code has arisen again recently because of the decision to make the Parrot VM, the intermediate representation for the Perl 6 language, a register rather than stack machine. Arguments for this design decision [Sugalski 2002] have been based on just a couple of small examples, rather than studies of real programs. The Lua VM [R. Ierusalimschy et al. 1996] was also switched from a stack to a register machine, with the release of version 5.0 in 2003. Similar suggestions were proposed for the JVM [McGlashan and Bower 1999] and the Inferno VM [Winterbottom and Pike 1997], again without studies of real programs.

1386h



Much of the early work on RISC architectures was based on systematic studies of programs that examined the real, rather than presumed, frequency of instruction usage. Studies on the IBM 360 [Shustek 1978; Alexander and Wortman 1975] and the VAX [Wiecek 1982] caused researchers to rethink the complex instruction sets of the time, and led to the first RISC architectures [Patterson and Ditzel 1980]. Today, basing design decisions on measured, rather than presumed, frequencies of instruction usage has become widely accepted as sound engineering practice.

## 8. CONCLUSIONS

A long standing question has been whether virtual stack or virtual register VMs can be executed more efficiently using an interpreter. Register VMs can be an attractive alternative to stack architectures because they enable the number of executed VM instructions to be substantially reduced. In this paper we have built on the previous work of Davis et al. [Davis et al. 2003; Gregg et al. 2005], which counted the number of instructions for the two architectures using a simple translation scheme. We have presented a much more sophisticated translation and optimization scheme for translating stack VM code to register VM code, which we believe gives a more accurate measure of the potential of virtual register machine architectures. We have also presented experimental results for a fully-featured register JVM.

We found that a register architecture requires an average of 46% fewer executed VM instructions. The resulting register code is 26% larger than the corresponding stack code. The increased cost of fetching more VM code due to larger code size involves only around one extra CPU load per VM instruction eliminated. On an AMD64 machine, the register machine has an average speedup of 1.48 if dispatch is performed using a C switch statement. Even if the more efficient inline-threaded dispatch is available, the average speedup over a corresponding stack JVM is still 1.15 for the register architecture on an AMD64.

### ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of VEE 2005 and ACM TACO, whose comments greatly improved earlier versions of this paper.

### REFERENCES

ALEXANDER, W. AND WORTMAN, D. 1975. Static and dynamic chracteristics of XPL programs. *Computer 8*, 11 (Nov.), 41–46.

ANTONIOLI, D. N. AND PILZ, M. 1998. Analysis of the Java class file format. Tech. rep.

ARNOLD, M., HIND, M., AND RYDER, B. G. 2002. Online feedback-directed optimization of Java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York. 111–129.

BELL, J. R. 1973. Threaded code. *Commun. ACM 16*, 6, 370–372.

BERNDL, M., VITALE, B., ZALESKI, M., AND BROWN, A. D. 2005. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *2005 International Symposium on Code Generation and Optimization*.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems 16*, 3 (May), 428–455.

1386 i




BULL, M., SMITH, L., WESTHEAD, M., HENTY, D., AND DAVEY, R. 2000. Benchmarking Java Grande applications. In *Second Ineternational Conference and Exhibtion on the Practical Application of Java*. Manchester, UK.

CHOI, J.-D., GROVE, D., HIND, M., AND SARKAR, V. 1999. Efficient and precise modeling of exceptions for the analysis of Java programs. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, New York. 21–31.

CLAUSEN, L. R., SCHULTZ, U. P., CONSEL, C., AND MULLER, G. 2000. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst. 22*, 3, 471–489.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct.), 451–490.

DAVIS, B., BEATTY, A., CASEY, K., GREGG, D., AND WALDRON, J. 2003. The case for virtual register machines. In *Interpreters, Virtual Machines and Emulators (IVME '03)*. 41–49.

DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*. 106–117.

ELLER, H. 2005. Optimizing interpreters with superinstructions. M.S. thesis, Institut für Computersprachen, Technische Universität Wien. http://www.complang.tuwien.ac.at/ Dimplomarbeiten/eller05.ps.gz.

ERTL, M. A. 1995. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*. 315–327.

ERTL, M. A. AND GREGG, D. 2003. The structure and performance of *efficient* interpreters. *The Journal of Instruction-Level Parallelism 5*. http://www.jilp.org/vol5/.

ERTL, M. A., GREGG, D., KRALL, A., AND PAYSAN, B. 2002. vmgen—A generator of efficient virtual machine interpreters. *Software—Practice and Experience 32*, 3, 265–294.

ERTL, M. A., THALINGER, C., AND KRALL, A. 2006. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies 4*, 25–32. Journal papers from *.NET Technologies 2006* conference.

FINK, S., KNOBE, K., AND SARKAR, V. 2000. Unified analysis of array and object references in strongly typed languages. *Lecture Notes in Computer Science* Volume 1824 (Feb.), 155–174.

GOSLING, J. 1995. Java Intermediate Bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*. ACM Sigplan Notices, vol. 30:3. San Francisco, CA. 111–118.

GREGG, D., BEATTY, A., CASEY, K., DAVIS, B., AND NISBET, A. 2005. The case for virtual register machines. *Science of Computer Programming, Special Issue on Interpreters Virtual Machines and Emulators 57*, 319–338.

IERUSALIMSCHY, R. L., DE FIGUEIREDO, H., AND CELES, W. 1996. Lua—an extensible extension language. *Software: Practice and Experience 26*, 6, 635–652.

IERUSALIMSCHY, R., DE FIGUEIREDO, L., AND CELES, W. 2005. The implementation of Lua 5.0. *Journal of Universal Computer Science 11*, 7, 1159–1176. http://www.jucs.org/jucs_11_7/the_implementation_of_lua.

KISTLER, T. AND FRANZ, M. 1999. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming 27*, 1, 21–33.

KLINT, P. 1981. Interpretation techniques. *Software—Practice and Experience 11*, 963–973.

KRALL, A. AND GRAFL, R. 1997. Cacao—a 64-bit JavaVM just-in-time compiler. *Concurrency—Practice and Experience 9*, 11, 1017–1030.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1, 121–141.

MCGLASHAN, B. AND BOWER, A. 1999. The interpreter is dead (slow). Isn't it? In *OOPSLA'99 Workshop: Simplicity, Performance and Portability in Virtual Machine Design.*

MÖSSENBÖCK, H. 2000. Adding static single assignment form and a graph coloring register allocator to the Java Hotspot client compiler. Technical Report TR-15, Johannes Kepler University Linz Institute for Practical Computer Science, Altenbergerstra 69, A-4040 Linz.

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann, San Francisco, CA.

MYERS, G. J. 1977. The case against stack-oriented instruction sets. *Computer Architecture News 6*, 3 (Aug.), 7–10.

1386j



PATTERSON, D. AND DITZEL, D. 1980. The case for the reduced instruction set computer. *Computer Architecture News 8*, 6 (Oct.), 25–33.

PIUMARTA, I. AND RICCARDI, F. 1998. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*. 291–300.

PUGH, W. 1999. Compressing Java class files. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York. 247–258.

RADHAKRISHNAN, R., VIJAYKRISHNAN, N., JOHN, L. K., AND SIVASUBRAMANIAM, A. 2000. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture* (8–12 Jan.). Toulouse, France. 387–398.

SCHULTHESS, P. AND MUMPRECHT, E. 1977. Reply to the case against stack-oriented instruction sets. *Computer Architecture News 6*, 5 (Dec.), 24–27.

SHI, Y., GREGG, D., BEATTY, A., AND ERTL, M. A. 2005. Virtual machine showdown: stack versus registers. In *ACM/SIGPLAN Conference on Virtual Execution Environments*. ACM Press, New York. 153–163.

SHUSTEK, L. 1978. Aanalysis and performance of computer instruction sets. Ph.D. thesis, Stanford University.

SPEC. 1998. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Press Release. http://www.spec.org/jvm98/press.html.

SUGALSKI, D. 2002. Parrot in detail. In *Yet Another Perl Conference (YAPC 02)*. Saint Louis, Missouri. http://www.parrotcode.org/talks/ParrotInDetail2.pdf.

SUN-MICROSYSTEMS. 2001. The Java Hotspot virtual machine. Tech. rep., Sun Microsystems Inc.

TIP, F., SWEENEY, P. F., LAFFRA, C., EISMA, A., AND STREETER, D. 2002. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst. 24*, 6, 625–666.

VALLÉE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. 1999. Soot—a Java optimization framework. In *Proceedings of CASCON 1999*. 125–135.

VITALE, B. AND ABDELRAHMAN, T. S. 2004. Catenation and specialization for Tcl virtual machine performance. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, Virtual Machines and Emulators*. ACM Press, New York. 42–50.

WIECEK, C. 1982. A case study of the VAX 11 instruction set usage for compiler execution. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operation Systems*. IEEE/ACM, Palo Alto, California. 177–184.

WINTERBOTTOM, P. AND PIKE, R. 1997. The design of the Inferno virtual machine. In *IEEE Compcon 97 Proceedings*. San Jose, California. 241–244.

Received August 2006; revised February 2007; accepted April 2007

138ca

# Dalvik VM Instruction Formats

Copyright © 2007 Google Inc. All rights reserved.

## Introduction and Oveview

This document lists the instruction formats used by Dalvik bytecode and is meant to be used in conjunction with the bytecode reference document.

### Bitwise descriptions

The first column in the format table lists the bitwise layout of the format. It consists of one or more space-separated "words" each of which describes a 16-bit code unit. Each character in a word represents four bits, read from high bits to low, with vertical bars ("|") interspersed to aid in reading. Uppercase letters in sequence from "A" are used to indicate fields within the format (which then get defined further by the syntax column). The term "op" is used to indicate the position of the eight-bit opcode within the format. A slashed zero ("ø") is used to indicate that all bits should be zero in the indicated position.

For example, the format "B|A|*op* CCCC" indicates that the format consists of two 16-bit code units. The first word consists of the opcode in the low eight bits and a pair of four-bit values in the high eight bits; and the second word consists of a single 16-bit value.

### Format IDs

The second column in the format table indicates the short identifier for the format, which is used in other documents and in code to identify the format.

Format IDs consist of three characters, two digits followed by a letter. The first digit indicates the number of 16-bit code units in the format. The second digit indicates the maximum number of registers that the format contains (maximum, since some formats can accomodate a variable number of registers), with the special designation "r" indicating that a range of registers is encoded. The final letter semi-mnemonically indicates the type of any extra data encoded by the format. For example, format "21t" is of length two, contains one register reference, and additionally contains a branch target.

Suggested static linking formats have an additional "s" suffix, making them four characters total.

The full list of typecode letters are as follows. Note that some forms have different sizes, depending on the format:

| Mnemonic | Bit Sizes | Meaning |
|---|---|---|
| b | 8 | immediate signed byte |
| c | 16, 32 | constant pool index |
| f | 16 | interface constants (only used in statically linked formats) |
| h | 16 | immediate signed hat (high-order bits of a 32- or 64-bit value; low-order bits are all 0) |

138cb

| Mnemonic | Bit Sizes | Meaning |
|---|---|---|
| i | 32 | immediate signed int, or 32-bit float |
| l | 64 | immediate signed long, or 64-bit double |
| m | 16 | method constants (only used in statically linked formats) |
| n | 4 | immediate signed nibble |
| s | 16 | immediate signed short |
| t | 8, 16, 32 | branch target |
| x | 0 | no additional data |

## Syntax

The third column of the format table indicates the human-oriented syntax for instructions which use the indicated format. Each instruction starts with the named opcode and is optionally followed by one or more arguments, themselves separated with commas.

Wherever an argument refers to a field from the first column, the letter for that field is indicated in the syntax, repeated once for each four bits of the field. For example, an eight-bit field labeled "`BB`" in the first column would also be labeled "`BB`" in the syntax column.

Arguments which name a register have the form "`v`*X*". The prefix "`v`" was chosen instead of the more common "`r`" exactly to avoid conflicting with (non-virtual) architectures on which a Dalvik virtual machine might be implemented which themselves use the prefix "`r`" for their registers. (That is, this decision makes it possible to talk about both virtual and real registers together without the need for circumlocution.)

Arguments which indicate a literal value have the form "`#+`*X*". Some formats indicate literals that only have non-zero bits in their high-order bits; for these, the zeroes are represented explicitly in the syntax, even though they do not appear in the bitwise representation.

Arguments which indicate a relative instruction address offset have the form "`+`*X*".

Arguments which indicate a literal constant pool index have the form "*kind*`@`*X*", where "*kind*" indicates which constant pool is being referred to. Each opcode that uses such a format explicitly allows only one kind of constant; see the opcode reference to figure out the correspondence. The four kinds of constant pool are "`string`" (string pool index), "`type`" (type pool index), "`field`" (field pool index), and "`meth`" (method pool index).

Similar to the representation of constant pool indices, there are also suggested (optional) forms that indicate prelinked offsets or indices. These prelinked values include "`vtaboff`" (vtable offset), "`fieldoff`" (field offset), and "`iface`" (interface pool index).

In the cases where a format value isn't explictly part of the syntax but instead picks a variant, each variant is listed with the prefix "`[`*X*`=`*N*`]`" (e.g., "`[B=2]`") to indicate the correspondence.

# The Formats

138cc

| Format | ID | Syntax | Notable Opcodes Covered |
|---|---|---|---|
| ØØ\|*op* | 10x | *op* | |
| B\|A\|*op* | 12x | *op* vA, vB | |
| | 11n | *op* vA, #+B | |
| AA\|*op* | 11x | *op* vAA | |
| | 10t | *op* +AA | goto |
| ØØ\|*op* AAAA | 20t | *op* +AAAA | goto/16 |
| AA\|*op* BBBB | 22x | *op* vAA, vBBBB | |
| | 21t | *op* vAA, +BBBB | |
| | 21s | *op* vAA, #+BBBB | |
| | 21h | *op* vAA, #+BBBB0000<br>*op* vAA, #+BBBB000000000000 | |
| | 21c | *op* vAA, type@BBBB<br>*op* vAA, field@BBBB<br>*op* vAA, string@BBBB | check-cast<br>const-class<br>const-string |
| AA\|*op* CC\|BB | 23x | *op* vAA, vBB, vCC | |
| | 22b | *op* vAA, vBB, #+CC | |
| B\|A\|*op* CCCC | 22t | *op* vA, vB, +CCCC | |
| | 22s | *op* vA, vB, #+CCCC | |
| | 22c | *op* vA, vB, type@CCCC<br>*op* vA, vB, field@CCCC | instance-of |
| | 22cs | *op* vA, vB, fieldoff@CCCC | *(suggested format for statically linked field access instructions of format 22c)* |
| ØØ\|*op* $AAAA_{lo}$ $AAAA_{hi}$ | 30t | *op* +AAAAAAAA | goto/32 |
| ØØ\|*op* AAAA BBBB | 32x | *op* vAAAA, vBBBB | |
| AA\|*op* $BBBB_{lo}$ $BBBB_{hi}$ | 31i | *op* vAA, #+BBBBBBBB | |
| | 31t | *op* vAA, +BBBBBBBB | |
| | 31c | *op* vAA, string@BBBBBBBB | const-string/jumbo |
| B\|A\|*op* CCCC G\|F\|E\|D | 35c | *[B=5] op* {vD, vE, vF, vG, vA}, meth@CCCC<br>*[B=5] op* {vD, vE, vF, vG, vA}, type@CCCC<br>*[B=4] op* {vD, vE, vF, vG}, *kind*@CCCC<br>*[B=3] op* {vD, vE, vF}, *kind*@CCCC<br>*[B=2] op* {vD, vE}, *kind*@CCCC<br>*[B=1] op* {vD}, *kind*@CCCC<br>*[B=0] op* {}, *kind*@CCCC | |
| B\|A\|*op* CCCC G\|F\|E\|D | 35ms | *[B=5] op* {vD, vE, vF, vG, vA}, vtaboff@CCCC<br>*[B=4] op* {vD, vE, vF, vG}, vtaboff@CCCC<br>*[B=3] op* {vD, vE, vF}, vtaboff@CCCC<br>*[B=2] op* {vD, vE}, vtaboff@CCCC<br>*[B=1] op* {vD}, vtaboff@CCCC | *(suggested format for statically linked* invoke-virtual *and* invoke-super *instructions of format 35c)* |
| B\|A\|*op* DDCC H\|G\|F\|E | 35fs | *[B=5] op* vB, {vE, vF, vG, vH, vA}, vtaboff@CC, iface@DD<br>*[B=4] op* vB, {vE, vF, vG, vH}, vtaboff@CC, iface@DD<br>*[B=3] op* vB, {vE, vF, vG}, vtaboff@CC, iface@DD<br>*[B=2] op* vB, {vE, vF}, vtaboff@CC, iface@DD | *(suggested format for statically linked* invoke-interface *instructions of format 35c)* |

138cd

| Format | ID | Syntax | Notable Opcodes Covered |
| --- | --- | --- | --- |
| | | *[B=1] op* vB, {vE}, vtaboff@CC, iface@DD | |
| AA\|*op* BBBB CCCC | 3rc | *op* {vCCCC .. vNNNN}, meth@BBBB<br>*op* {vCCCC .. vNNNN}, type@BBBB<br><br>*(where* NNNN = CCCC+AA-1*, that is* A *determines the count* 0..255*, and* C *determines the first register)* | |
| AA\|*op* BBBB CCCC | 3rms | *op* {vCCCC .. vNNNN}, vtaboff@BBBB<br><br>*(where* NNNN = CCCC+AA-1*, that is* A *determines the count* 0..255*, and* C *determines the first register)* | *(suggested format for statically linked* invoke-virtual *and* invoke-super *instructions of format* 3rc*)* |
| AA\|*op* CCBB DDDD | 3rfs | *op* {vDDDD .. vNNNN}, vtaboff@BB, iface@CC<br><br>*(where* NNNN = DDDD+AA-1*, that is* A *determines the count* 0..255*, and* D *determines the first register)* | *(suggested format for statically linked* invoke-interface *instructions of format* 3rc*)* |
| AA\|*op* $BBBB_{lo}$ BBBB BBBB $BBBB_{hi}$ | 51l | *op* vAA, #+BBBBBBBBBBBBBBBB | const-wide |

138ce

# Bytecode for the Dalvik VM

Copyright © 2007 Google Inc. All rights reserved.

## General Design

- The machine model and calling conventions are meant to approximately imitate common real architectures and C-style calling conventions:
  - The VM is register-based, and frames are fixed in size upon creation. Each frame consists of a particular number of registers (specified by the method) as well as any adjunct data needed to execute the method, such as (but not limited to) the program counter and a reference to the `.dex` file that contains the method.
  - The *N* arguments to a method land in the last *N* registers of the method's invocation frame.
  - Registers are 32 bits wide. Adjacent register pairs are used for 64-bit values.
  - In terms of bitwise representation, `(Object) null == (int) 0`.
- The storage unit in the instruction stream is a 16-bit unsigned quantity. Some bits in some instructions are ignored / must-be-zero.
- Instructions aren't gratuitously limited to a particular type. For example, instructions that move 32-bit register values without interpretation don't have to specify whether they are moving ints or floats.
- There are separately enumerated and indexed constant pools for references to strings, types, fields, and methods.
- Bitwise literal data is represented in-line in the instruction stream.
- Because, in practice, it is uncommon for a method to need more than 16 registers, and because needing more than eight registers *is* reasonably common, many instructions may only address the first 16 registers. When reasonably possible, instructions allow references to up to the first 256 registers. In cases where an instruction variant isn't available to address a desired register, it is expected that the register contents get moved from the original register to a low register (before the operation) and/or moved from a low result register to a high register (after the operation).
- When installed on a running system, some instructions may be altered, changing their format, as an install-time static linking optimization. This is to allow for faster execution once linkage is known. See the associated instruction formats document for the suggested variants. The word "suggested" is used advisedly; it is not mandatory to implement these.
- Human-syntax and mnemonics:
  - Dest-then-source ordering for arguments.
  - Some opcodes have a disambiguating suffix with respect to the type(s) they operate on: Type-general 64-bit opcodes are suffixed with `-wide`. Type-specific opcodes are suffixed with their type (or a straightforward abbreviation), one of: `-boolean` `-byte` `-char` `-short` `-int` `-long` `-float` `-double` `-object` `-string` `-class` `-void`. Type-general 32-bit opcodes are unmarked.
  - Some opcodes have a disambiguating suffix to distinguish otherwise-identical operations that have different instruction layouts or options. These suffixes are separated from the main names with a slash ("`/`") and mainly exist at all to make there be a one-to-one mapping with static constants in the code that generates and interprets executables (that is, to reduce ambiguity for humans).
- See the instruction formats document for more details about the various instruction formats (listed under "Op & Format") as well as details about the opcode syntax.

138cf

# Summary of Instruction Set

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| 00 10x | nop | | Waste cycles. |
| 01 12x | move vA, vB | A: destination register (4 bits)<br>B: source register (4 bits) | Move the contents of one non-object register to another. |
| 02 22x | move/from16 vAA, vBBBB | A: destination register (8 bits)<br>B: source register (16 bits) | Move the contents of one non-object register to another. |
| 03 32x | move/16 vAAAA, vBBBB | A: destination register (16 bits)<br>B: source register (16 bits) | Move the contents of one non-object register to another. |
| 04 12x | move-wide vA, vB | A: destination register pair (4 bits)<br>B: source register pair (4 bits) | Move the contents of one register-pair to another.<br>**Note:** It is legal to move from v*N* to either v*N-1* or v*N+1*, so implementations must arrange for both halves of a register pair to be read before anything is written. |
| 05 22x | move-wide/from16 vAA, vBBBB | A: destination register pair (8 bits)<br>B: source register pair (16 bits) | Move the contents of one register-pair to another.<br>**Note:** Implementation considerations are the same as move-wide, above. |
| 06 32x | move-wide/16 vAAAA, vBBBB | A: destination register pair (16 bits)<br>B: source register pair (16 bits) | Move the contents of one register-pair to another.<br>**Note:** Implementation considerations are the same as move-wide, above. |
| 07 12x | move-object vA, vB | A: destination register (4 bits)<br>B: source register (4 bits) | Move the contents of one object-bearing register to another. |
| 08 22x | move-object/from16 vAA, vBBBB | A: destination register (8 bits)<br>B: source register (16 bits) | Move the contents of one object-bearing register to another. |
| 09 32x | move-object/16 vAAAA, vBBBB | A: destination register (16 bits)<br>B: source register (16 bits) | Move the contents of one object-bearing register to another. |
| 0a 11x | move-result vAA | A: destination register (8 bits) | Move the single-word non-object result of the most recent invoke-*kind* into the indicated register. This must be done as the instruction immediately after an invoke-*kind* whose (single-word, non-object) result is not to be ignored; anywhere else is invalid. |
| 0b 11x | move-result-wide vAA | A: destination register pair (8 bits) | Move the double-word result of the most recent invoke-*kind* into the indicated register pair. This must be done as the instruction immediately after an invoke-*kind* whose (double-word) result is not to be ignored; anywhere else is invalid. |
| 0c 11x | move-result-object vAA | A: destination register (8 bits) | Move the object result of the most recent invoke-*kind* into the indicated register. This must be done as the instruction immediately after an invoke-*kind* or filled-new-array whose (object) result is not to be ignored; anywhere else is invalid. |
| 0d 11x | move-exception vAA | A: destination register (8 bits) | Save a just-caught exception into the given register. This should be the first instruction of any exception handler whose caught exception is not to be ignored, and this instruction may *only* ever occur as the first instruction of an exception handler; anywhere else is invalid. |
| 0e 10x | return-void | | Return from a void method. |

138cg

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| 0f 11x | return vAA | A: return value register (8 bits) | Return from a single-width (32-bit) non-object value-returning method. |
| 10 11x | return-wide vAA | A: return value register-pair (8 bits) | Return from a double-width (64-bit) value-returning method. |
| 11 11x | return-object vAA | A: return value register (8 bits) | Return from an object-returning method. |
| 12 11n | const/4 vA, #+B | A: destination register (4 bits)<br>B: signed int (4 bits) | Move the given literal value (sign-extended to 32 bits) into the specified register. |
| 13 21s | const/16 vAA, #+BBBB | A: destination register (8 bits)<br>B: signed int (16 bits) | Move the given literal value (sign-extended to 32 bits) into the specified register. |
| 14 31i | const vAA, #+BBBBBBBB | A: destination register (8 bits)<br>B: arbitrary 32-bit constant | Move the given literal value into the specified register. |
| 15 21h | const/high16 vAA, #+BBBB0000 | A: destination register (8 bits)<br>B: signed int (16 bits) | Move the given literal value (right-zero-extended to 32 bits) into the specified register. |
| 16 21s | const-wide/16 vAA, #+BBBB | A: destination register (8 bits)<br>B: signed int (16 bits) | Move the given literal value (sign-extended to 64 bits) into the specified register-pair. |
| 17 31i | const-wide/32 vAA, #+BBBBBBBB | A: destination register (8 bits)<br>B: signed int (32 bits) | Move the given literal value (sign-extended to 64 bits) into the specified register-pair. |
| 18 51l | const-wide vAA, #+BBBBBBBBBBBBBBBB | A: destination register (8 bits)<br>B: arbitrary double-width (64-bit) constant | Move the given literal value into the specified register-pair. |
| 19 21h | const-wide/high16 vAA, #+BBBB000000000000 | A: destination register (8 bits)<br>B: signed int (16 bits) | Move the given literal value (right-zero-extended to 64 bits) into the specified register-pair. |
| 1a 21c | const-string vAA, string@BBBB | A: destination register (8 bits)<br>B: string index | Move a reference to the string specified by the given index into the specified register. |
| 1b 31c | const-string/jumbo vAA, string@BBBBBBBB | A: destination register (8 bits)<br>B: string index | Move a reference to the string specified by the given index into the specified register. |
| 1c 21c | const-class vAA, type@BBBB | A: destination register (8 bits)<br>B: type index | Move a reference to the class specified by the given index into the specified register. In the case where the indicated type is primitive, this will store a reference to the primitive type's degenerate class. |
| 1d 11x | monitor-enter vAA | A: reference-bearing register (8 bits) | Acquire the monitor for the indicated object. |
| 1e 11x | monitor-exit vAA | A: reference-bearing register (8 bits) | Release the monitor for the indicated object.<br>**Note:** If this instruction needs to throw an exception, it must do so as if the pc has already advanced past the instruction. It may be useful to think of this as the instruction successfully executing (in a sense), and the exception getting thrown *after* the instruction but *before* the next one gets a chance to run. This definition makes it possible for a method to use a monitor cleanup catch-all (e.g., `finally`) block as the monitor cleanup for that block itself, as a way to handle the arbitrary exceptions that might get thrown due to the historical implementation of `Thread.stop()`, while still managing to have proper monitor hygiene. |
| 1f 21c | check-cast vAA, type@BBBB | A: reference-bearing register (8 bits)<br>B: type index (16 bits) | Throw if the reference in the given register cannot be cast to the indicated type. The type must be a reference type (not a primitive type). |
| 20 22c | instance-of vA, vB, type@CCCC | A: destination register (4 bits)<br>B: reference-bearing register (4 bits)<br>C: type index (16 bits) | Store in the given destination register 1 if the indicated reference is an instance of the given type, or 0 if not. The type must be a reference type (not a primitive type). |

138ch

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| 21 12x | array-length vA, vB | A: destination register (4 bits)<br>B: array reference-bearing register (4 bits) | Store in the given destination register the length of the indicated array, in entries |
| 22 21c | new-instance vAA, type@BBBB | A: destination register (8 bits)<br>B: type index | Construct a new instance of the indicated type, storing a reference to it in the destination. The type must refer to a non-array class. |
| 23 22c | new-array vA, vB, type@CCCC | A: destination register (8 bits)<br>B: size register<br>C: type index | Construct a new array of the indicated type and size. The type must be an array type. |
| 24 35c | filled-new-array {vD, vE, vF, vG, vA}, type@CCCC | B: array size and argument word count (4 bits)<br>C: type index (16 bits)<br>D..G, A: argument registers (4 bits each) | Construct an array of the given type and size, filling it with the supplied contents. The type must be an array type. The array's contents must be single-word (that is, no arrays of `long` or `double`). The constructed instance is stored as a "result" in the same way that the method invocation instructions store their results, so the constructed instance must be moved to a register with a subsequent `move-result-object` instruction (if it is to be used). |
| 25 3rc | filled-new-array/range {vCCCC .. vNNNN}, type@BBBB | A: array size and argument word count (8 bits)<br>B: type index (16 bits)<br>C: first argument register (16 bits)<br>N = A + C - 1 | Construct an array of the given type and size, filling it with the supplied contents. Clarifications and restrictions are the same as `filled-new-array`, described above. |
| 26 31t | fill-array-data vAA, +BBBBBBBB *(with supplemental data as specified below in "fill-array-data Format")* | A: array reference (8 bits)<br>B: signed "branch" offset to table data (32 bits) | Fill the given array with the indicated data. The reference must be to an array of primitives, and the data table must match it in type and size.<br>**Note:** The address of the table is guaranteed to be even (that is, 4-byte aligned). If the code size of the method is otherwise odd, then an extra code unit is inserted between the main code and the table whose value is the same as a `nop`. |
| 27 11x | throw vAA | A: exception-bearing register (8 bits) | Throw the indicated exception. |
| 28 10t | goto +AA | A: signed branch offset (8 bits) | Unconditionally jump to the indicated instruction.<br>**Note:** The branch offset may not be `0`. (A spin loop may be legally constructed either with `goto/32` or by including a `nop` as a target before the branch.) |
| 29 20t | goto/16 +AAAA | A: signed branch offset (16 bits) | Unconditionally jump to the indicated instruction.<br>**Note:** The branch offset may not be `0`. (A spin loop may be legally constructed either with `goto/32` or by including a `nop` as a target before the branch.) |
| 2a 30t | goto/32 +AAAAAAAA | A: signed branch offset (32 bits) | Unconditionally jump to the indicated instruction. |
| 2b 31t | packed-switch vAA, +BBBBBBBB *(with supplemental data as specified below in "packed-switch Format")* | A: register to test<br>B: signed "branch" offset to table data (32 bits) | Jump to a new instruction based on the value in the given register, using a table of offsets corresponding to each value in a particular integral range, or fall through to the next instruction if there is no match.<br>**Note:** The address of the table is guaranteed to be even (that is, 4-byte aligned). If the code size of the method is otherwise odd, then an extra code unit is inserted between the main code and the table whose value is |

138ci

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| | | | the same as a nop. |
| 2c 31t | sparse-switch vAA, +BBBBBBBB *(with supplemental data as specified below in "sparse-switch Format")* | A: register to test<br>B: signed "branch" offset to table data (32 bits) | Jump to a new instruction based on the value in the given register, using an ordered table of value-offset pairs, or fall through to the next instruction if there is no match.<br>**Note:** Alignment and padding considerations are identical to packed-switch, above. |
| 2d..31 23x | cmp*kind* vAA, vBB, vCC<br>2d: cmpl-float *(lt bias)*<br>2e: cmpg-float *(gt bias)*<br>2f: cmpl-double *(lt bias)*<br>30: cmpg-double *(gt bias)*<br>31: cmp-long | A: destination register (8 bits)<br>B: first source register or pair<br>C: second source register or pair | Perform the indicated floating point or long comparison, storing 0 if the two arguments are equal, 1 if the second argument is larger, or -1 if the first argument is larger. The "bias" listed for the floating point operations indicates how NaN comparisons are treated: "Gt bias" instructions return 1 for NaN comparisons, and "lt bias" instructions return -1.<br>For example, to check to see if floating point a < b, then it is advisable to use cmpg-float; a result of -1 indicates that the test was true, and the other values indicate it was false either due to a valid comparison or because one or the other values was NaN. |
| 32..37 22t | if-*test* vA, vB, +CCCC<br>32: if-eq<br>33: if-ne<br>34: if-lt<br>35: if-ge<br>36: if-gt<br>37: if-le | A: first register to test (4 bits)<br>B: second register to test (4 bits)<br>C: signed branch offset (16 bits) | Branch to the given destination if the given two registers' values compare as specified.<br>**Note:** The branch offset may not be 0. (A spin loop may be legally constructed either by branching around a backward goto or by including a nop as a target before the branch.) |
| 38..3d 21t | if-*testz* vAA, +BBBB<br>38: if-eqz<br>39: if-nez<br>3a: if-ltz<br>3b: if-gez<br>3c: if-gtz<br>3d: if-lez | A: register to test (8 bits)<br>B: signed branch offset (16 bits) | Branch to the given destination if the given register's value compares with 0 as specified.<br>**Note:** The branch offset may not be 0. (A spin loop may be legally constructed either by branching around a backward goto or by including a nop as a target before the branch.) |
| 3e..43 10x | *(unused)* | | *(unused)* |
| 44..51 23x | *arrayop* vAA, vBB, vCC<br>44: aget<br>45: aget-wide<br>46: aget-object<br>47: aget-boolean<br>48: aget-byte<br>49: aget-char<br>4a: aget-short<br>4b: aput<br>4c: aput-wide<br>4d: aput-object<br>4e: aput-boolean<br>4f: aput-byte<br>50: aput-char<br>51: aput-short | A: value register or pair; may be source or dest (8 bits)<br>B: array register (8 bits)<br>C: index register (8 bits) | Perform the identified array operation at the identified index of the given array, loading or storing into the value register. |
| 52..5f 22c | i*instanceop* vA, vB, field@CCCC<br>52: iget<br>53: iget-wide<br>54: iget-object<br>55: iget-boolean<br>56: iget-byte<br>57: iget-char<br>58: iget-short<br>59: iput<br>5a: iput-wide<br>5b: iput-object<br>5c: iput-boolean<br>5d: iput-byte<br>5e: iput-char<br>5f: iput-short | A: value register or pair; may be source or dest (4 bits)<br>B: object register (4 bits)<br>C: instance field reference index (16 bits) | Perform the identified object instance field operation with the identified field, loading or storing into the value register.<br>**Note:** These opcodes are reasonable candidates for static linking, altering the field argument to be a more direct offset. |

138cj

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| 60..6d 21c | *sstaticop* vAA, field@BBBB<br>60: sget<br>61: sget-wide<br>62: sget-object<br>63: sget-boolean<br>64: sget-byte<br>65: sget-char<br>66: sget-short<br>67: sput<br>68: sput-wide<br>69: sput-object<br>6a: sput-boolean<br>6b: sput-byte<br>6c: sput-char<br>6d: sput-short | A: value register or pair; may be source or dest (8 bits)<br>B: static field reference index (16 bits) | Perform the identified object static field operation with the identified static field, loading or storing into the value register.<br>**Note:** These opcodes are reasonable candidates for static linking, altering the field argument to be a more direct offset. |
| 6e..72 35c | invoke-*kind* {vD, vE, vF, vG, vA}, meth@CCCC<br>6e: invoke-virtual<br>6f: invoke-super<br>70: invoke-direct<br>71: invoke-static<br>72: invoke-interface | B: argument word count (4 bits)<br>C: method index (16 bits)<br>D..G, A: argument registers (4 bits each) | Call the indicated method. The result (if any) may be stored with an appropriate move-result* variant as the immediately subsequent instruction.<br>invoke-virtual is used to invoke a normal virtual method (a method that is not static or final, and is not a constructor).<br>invoke-super is used to invoke the closest superclass's virtual method (as opposed to the one with the same method_id in the calling class).<br>invoke-direct is used to invoke a non-static direct method (that is, an instance method that is by its nature non-overridable, namely either a private instance method or a constructor).<br>invoke-static is used to invoke a static method (which is always considered a direct method).<br>invoke-interface is used to invoke an interface method, that is, on an object whose concrete class isn't known, using a method_id that refers to an interface.<br>**Note:** These opcodes are reasonable candidates for static linking, altering the method argument to be a more direct offset (or pair thereof). |
| 73 10x | *(unused)* | | *(unused)* |
| 74..78 3rc | invoke-*kind*/range {vCCCC .. vNNNN}, meth@BBBB<br>74: invoke-virtual/range<br>75: invoke-super/range<br>76: invoke-direct/range<br>77: invoke-static/range<br>78: invoke-interface/range | A: argument word count (8 bits)<br>B: method index (16 bits)<br>C: first argument register (16 bits)<br>N = A + C - 1 | Call the indicated method. See first invoke-*kind* description above for details, caveats, and suggestions. |
| 79..7a 10x | *(unused)* | | *(unused)* |
| 7b..8f 12x | *unop* vA, vB<br>7b: neg-int<br>7c: not-int<br>7d: neg-long<br>7e: not-long<br>7f: neg-float<br>80: neg-double<br>81: int-to-long<br>82: int-to-float<br>83: int-to-double<br>84: long-to-int<br>85: long-to-float<br>86: long-to-double<br>87: float-to-int<br>88: float-to-long<br>89: float-to-double<br>8a: double-to-int<br>8b: double-to-long<br>8c: double-to-float<br>8d: int-to-byte | A: destination register or pair (4 bits)<br>B: source register or pair (4 bits) | Perform the identified unary operation on the source register, storing the result in the destination register. |

138ck

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| | 8e: int-to-char<br>8f: int-to-short | | |
| 90..af 23x | *binop* vAA, vBB, vCC<br>90: add-int<br>91: sub-int<br>92: mul-int<br>93: div-int<br>94: rem-int<br>95: and-int<br>96: or-int<br>97: xor-int<br>98: shl-int<br>99: shr-int<br>9a: ushr-int<br>9b: add-long<br>9c: sub-long<br>9d: mul-long<br>9e: div-long<br>9f: rem-long<br>a0: and-long<br>a1: or-long<br>a2: xor-long<br>a3: shl-long<br>a4: shr-long<br>a5: ushr-long<br>a6: add-float<br>a7: sub-float<br>a8: mul-float<br>a9: div-float<br>aa: rem-float<br>ab: add-double<br>ac: sub-double<br>ad: mul-double<br>ae: div-double<br>af: rem-double | A: destination register or pair (8 bits)<br>B: first source register or pair (8 bits)<br>C: second source register or pair (8 bits) | Perform the identified binary operation on the two source registers, storing the result in the first source register. |
| b0..cf 12x | *binop*/2addr vA, vB<br>b0: add-int/2addr<br>b1: sub-int/2addr<br>b2: mul-int/2addr<br>b3: div-int/2addr<br>b4: rem-int/2addr<br>b5: and-int/2addr<br>b6: or-int/2addr<br>b7: xor-int/2addr<br>b8: shl-int/2addr<br>b9: shr-int/2addr<br>ba: ushr-int/2addr<br>bb: add-long/2addr<br>bc: sub-long/2addr<br>bd: mul-long/2addr<br>be: div-long/2addr<br>bf: rem-long/2addr<br>c0: and-long/2addr<br>c1: or-long/2addr<br>c2: xor-long/2addr<br>c3: shl-long/2addr<br>c4: shr-long/2addr<br>c5: ushr-long/2addr<br>c6: add-float/2addr<br>c7: sub-float/2addr<br>c8: mul-float/2addr<br>c9: div-float/2addr<br>ca: rem-float/2addr<br>cb: add-double/2addr<br>cc: sub-double/2addr<br>cd: mul-double/2addr<br>ce: div-double/2addr<br>cf: rem-double/2addr | A: destination and first source register or pair (4 bits)<br>B: second source register or pair (4 bits) | Perform the identified binary operation on the two source registers, storing the result in the first source register. |
| d0..d7 22s | *binop*/lit16 vA, vB, #+CCCC<br>d0: add-int/lit16<br>d1: rsub-int (reverse subtract)<br>d2: mul-int/lit16<br>d3: div-int/lit16<br>d4: rem-int/lit16<br>d5: and-int/lit16<br>d6: or-int/lit16<br>d7: xor-int/lit16 | A: destination register (4 bits)<br>B: source register (4 bits)<br>C: signed int constant (16 bits) | Perform the indicated binary op on the indicated register (first argument) and literal value (second argument), storing the result in the destination register.<br>**Note:** `rsub-int` does not have a suffix since this version is the main opcode of its family. Also, see below for details on its semantics. |

138cl

| Op & Format | Mnemonic / Syntax | Arguments | Description |
|---|---|---|---|
| d8..e2 22b | *binop*/lit8 vAA, vBB, #+CC<br>d8: add-int/lit8<br>d9: rsub-int/lit8<br>da: mul-int/lit8<br>db: div-int/lit8<br>dc: rem-int/lit8<br>dd: and-int/lit8<br>de: or-int/lit8<br>df: xor-int/lit8<br>e0: shl-int/lit8<br>e1: shr-int/lit8<br>e2: ushr-int/lit8 | A: destination register (8 bits)<br>B: source register (8 bits)<br>C: signed int constant (8 bits) | Perform the indicated binary op on the indicated register (first argument) and literal value (second argument), storing the result in the destination register.<br>**Note:** See below for details on the semantics of `rsub-int`. |
| e3..ff 10x | *(unused)* | | *(unused)* |

## `packed-switch` Format

| Name | Format | Description |
|---|---|---|
| `ident` | `ushort = 0x0100` | identifying pseudo-opcode |
| `size` | `ushort` | number of entries in the table |
| `first_key` | `int` | first (and lowest) switch case value |
| `targets` | `int[]` | list of `size` relative branch targets. The targets are relative to the address of the switch opcode, not of this table. |

**Note:** The total number of code units for an instance of this table is `(size * 2) + 4`.

## `sparse-switch` Format

| Name | Format | Description |
|---|---|---|
| `ident` | `ushort = 0x0200` | identifying pseudo-opcode |
| `size` | `ushort` | number of entries in the table |
| `keys` | `int[]` | list of `size` key values, sorted low-to-high |
| `targets` | `int[]` | list of `size` relative branch targets, each corresponding to the key value at the same index. The targets are relative to the address of the switch opcode, not of this table. |

**Note:** The total number of code units for an instance of this table is `(size * 4) + 2`.

## `fill-array-data` Format

| Name | Format | Description |
| --- | --- | --- |
| `ident` | `ushort = 0x0300` | identifying pseudo-opcode |
| `element_width` | `ushort` | number of bytes in each element |
| `size` | `uint` | number of elements in the table |
| `data` | `ubyte[]` | data values |

**Note:** The total number of code units for an instance of this table is `(size * element_width + 1) / 2 + 4`.

# Mathematical Operation Details

**Note:** Floating point operations must follow IEEE 754 rules, using round-to-nearest and gradual underflow, except where stated otherwise.

| Opcode | C Semantics | Notes |
| --- | --- | --- |
| `neg-int` | `int32 a;`<br>`int32 result = -a;` | Unary twos-complement. |
| `not-int` | `int32 a;`<br>`int32 result = ~a;` | Unary ones-complement. |
| `neg-long` | `int64 a;`<br>`int64 result = -a;` | Unary twos-complement. |
| `not-long` | `int64 a;`<br>`int64 result = ~a;` | Unary ones-complement. |
| `neg-float` | `float a;`<br>`float result = -a;` | Floating point negation. |
| `neg-double` | `double a;`<br>`double result = -a;` | Floating point negation. |
| `int-to-long` | `int32 a;`<br>`int64 result = (int64) a;` | Sign extension of `int32` into `int64`. |
| `int-to-float` | `int32 a;`<br>`float result = (float) a;` | Conversion of `int32` to `float`, using round-to-nearest. This loses precision for some values. |
| `int-to-double` | `int32 a;`<br>`double result = (double) a;` | Conversion of `int32` to `double`. |
| `long-to-int` | `int64 a;`<br>`int32 result = (int32) a;` | Truncation of `int64` into `int32`. |
| `long-to-float` | `int64 a;`<br>`float result = (float) a;` | Conversion of `int64` to `float`, using round-to-nearest. This loses precision for some values. |
| `long-to-double` | `int64 a;`<br>`double result = (double) a;` | Conversion of `int64` to `double`, using round-to-nearest. This loses precision for some values. |
| `float-to-int` | `float a;`<br>`int32 result = (int32) a;` | Conversion of `float` to `int32`, using round-toward-zero. `NaN` and `-0.0` (negative zero) convert to the integer `0`. Infinities and values with too large a magnitude to be represented get converted to either `0x7fffffff` or `-0x80000000` |

138cn

| Opcode | C Semantics | Notes |
|---|---|---|
| | | depending on sign. |
| float-to-long | float a; int64 result = (int64) a; | Conversion of float to int32, using round-toward-zero. The same special case rules as for float-to-int apply here, except that out-of-range values get converted to either 0x7fffffffffffffff or -0x8000000000000000 depending on sign. |
| float-to-double | float a; double result = (double) a; | Conversion of float to double, preserving the value exactly. |
| double-to-int | double a; int32 result = (int32) a; | Conversion of double to int32, using round-toward-zero. The same special case rules as for float-to-int apply here. |
| double-to-long | double a; int64 result = (int64) a; | Conversion of double to int64, using round-toward-zero. The same special case rules as for float-to-long apply here. |
| double-to-float | double a; float result = (float) a; | Conversion of double to float, using round-to-nearest. This loses precision for some values. |
| int-to-byte | int32 a; int32 result = (a << 24) >> 24; | Truncation of int32 to int8, sign extending the result. |
| int-to-char | int32 a; int32 result = a & 0xffff; | Truncation of int32 to uint16, without sign extension. |
| int-to-short | int32 a; int32 result = (a << 16) >> 16; | Truncation of int32 to int16, sign extending the result. |
| add-int | int32 a, b; int32 result = a + b; | Twos-complement addition. |
| sub-int | int32 a, b; int32 result = a - b; | Twos-complement subtraction. |
| rsub-int | int32 a, b; int32 result = b - a; | Twos-complement reverse subtraction. |
| mul-int | int32 a, b; int32 result = a * b; | Twos-complement multiplication. |
| div-int | int32 a, b; int32 result = a / b; | Twos-complement division, rounded towards zero (that is, truncated to integer). This throws ArithmeticException if b == 0. |
| rem-int | int32 a, b; int32 result = a % b; | Twos-complement remainder after division. The sign of the result is the same as that of a, and it is more precisely defined as result == a - (a / b) * b. This throws ArithmeticException if b == 0. |
| and-int | int32 a, b; int32 result = a & b; | Bitwise AND. |
| or-int | int32 a, b; int32 result = a \| b; | Bitwise OR. |
| xor-int | int32 a, b; int32 result = a ^ b; | Bitwise XOR. |
| shl-int | int32 a, b; int32 result = a << (b & 0x1f); | Bitwise shift left (with masked argument). |
| shr-int | int32 a, b; int32 result = a >> (b & 0x1f); | Bitwise signed shift right (with masked argument). |
| ushr-int | uint32 a, b; int32 result = a >> (b & 0x1f); | Bitwise unsigned shift right (with masked argument). |

138co

| Opcode | C Semantics | Notes |
|---|---|---|
| add-long | `int64 a, b;` `int64 result = a + b;` | Twos-complement addition. |
| sub-long | `int64 a, b;` `int64 result = a - b;` | Twos-complement subtraction. |
| mul-long | `int64 a, b;` `int64 result = a * b;` | Twos-complement multiplication. |
| div-long | `int64 a, b;` `int64 result = a / b;` | Twos-complement division, rounded towards zero (that is, truncated to integer). This throws `ArithmeticException` if `b == 0`. |
| rem-long | `int64 a, b;` `int64 result = a % b;` | Twos-complement remainder after division. The sign of the result is the same as that of a, and it is more precisely defined as `result == a - (a / b) * b`. This throws `ArithmeticException` if `b == 0`. |
| and-long | `int64 a, b;` `int64 result = a & b;` | Bitwise AND. |
| or-long | `int64 a, b;` `int64 result = a \| b;` | Bitwise OR. |
| xor-long | `int64 a, b;` `int64 result = a ^ b;` | Bitwise XOR. |
| shl-long | `int64 a, b;` `int64 result = a << (b & 0x3f);` | Bitwise shift left (with masked argument). |
| shr-long | `int64 a, b;` `int64 result = a >> (b & 0x3f);` | Bitwise signed shift right (with masked argument). |
| ushr-long | `uint64 a, b;` `int64 result = a >> (b & 0x3f);` | Bitwise unsigned shift right (with masked argument). |
| add-float | `float a, b;` `float result = a + b;` | Floating point addition. |
| sub-float | `float a, b;` `float result = a - b;` | Floating point subtraction. |
| mul-float | `float a, b;` `float result = a * b;` | Floating point multiplication. |
| div-float | `float a, b;` `float result = a / b;` | Floating point division. |
| rem-float | `float a, b;` `float result = a % b;` | Floating point remainder after division. This function is different than IEEE 754 remainder and is defined as `result == a - roundTowardZero(a / b) * b`. |
| add-double | `double a, b;` `double result = a + b;` | Floating point addition. |
| sub-double | `double a, b;` `double result = a - b;` | Floating point subtraction. |
| mul-double | `double a, b;` `double result = a * b;` | Floating point multiplication. |
| div-double | `double a, b;` `double result = a / b;` | Floating point division. |
| rem-double | `double a, b;` `double result = a % b;` | Floating point remainder after division. This function is different than IEEE 754 remainder and is defined as `result == a - roundTowardZero(a / b) * b`. |

138da

# `.dex` — Dalvik Executable Format

Copyright © 2007 Google Inc. All rights reserved.

This document describes the layout and contents of `.dex` files, which are used to hold a set of class definitions and their associated adjunct data.

# Guide To Types

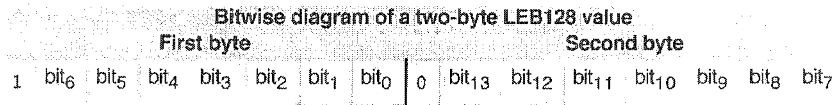
| Name | Description |
|---|---|
| `byte` | 8-bit signed int |
| `ubyte` | 8-bit unsigned int |
| `short` | 16-bit signed int, little-endian |
| `ushort` | 16-bit unsigned int, little-endian |
| `int` | 32-bit signed int, little-endian |
| `uint` | 32-bit unsigned int, little-endian |
| `long` | 64-bit signed int, little-endian |
| `ulong` | 64-bit unsigned int, little-endian |
| `sleb128` | signed LEB128, variable-length (see below) |
| `uleb128` | unsigned LEB128, variable-length (see below) |
| `uleb128p1` | unsigned LEB128 plus `1`, variable-length (see below) |

## LEB128

LEB128 ("**L**ittle-**E**ndian **B**ase **128**") is a variable-length encoding for arbitrary signed or unsigned integer quantities. The format was borrowed from the DWARF3 specification. In a `.dex` file, LEB128 is only ever used to encode 32-bit quantities.

Each LEB128 encoded value consists of one to five bytes, which together represent a single 32-bit value. Each byte has its most significant bit set except for the final byte in the sequence, which has its most significant bit clear. The remaining seven bits of each byte are payload, with the least significant seven bits of the quantity in the first byte, the next seven in the second byte and so on. In the case of a signed LEB128 (`sleb128`), the most significant payload bit of the final byte in the sequence is sign-extended to produce the final value. In the unsigned case (`uleb128`), any bits not explicitly represented are interpreted as `0`.

138db



| Bitwise diagram of a two-byte LEB128 value | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First byte | | | | | | | | Second byte | | | | | | | |
| 1 | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ | 0 | $bit_{13}$ | $bit_{12}$ | $bit_{11}$ | $bit_{10}$ | $bit_9$ | $bit_8$ | $bit_7$ |

The variant `uleb128p1` is used to represent a signed value, where the representation is of the value *plus one* encoded as a `uleb128`. This makes the encoding of `-1` (alternatively thought of as the unsigned value `0xffffffff`) — but no other negative number — a single byte, and is useful in exactly those cases where the represented number must either be non-negative or `-1` (or `0xffffffff`), and where no other negative values are allowed (or where large unsigned values are unlikely to be needed).

Here are some examples of the formats:

| Encoded Sequence | As `sleb128` | As `uleb128` | As `uleb128p1` |
|---|---|---|---|
| 00 | 0 | 0 | -1 |
| 01 | 1 | 1 | 0 |
| 7f | -1 | 127 | 126 |
| 80 7f | -128 | 16256 | 16255 |

# Overall File Layout

| Name | Format | Description |
|---|---|---|
| header | header_item | the header |
| string_ids | string_id_item[] | string identifiers list. These are identifiers for all the strings used by this file, either for internal naming (e.g., type descriptors) or as constant objects referred to by code. This list must be sorted by string contents, using UTF-16 code point values (not in a locale-sensitive manner). |
| type_ids | type_id_item[] | type identifiers list. These are identifiers for all types (classes, arrays, or primitive types) referred to by this file, whether defined in the file or not. This list must be sorted by `string_id` index. |
| proto_ids | proto_id_item[] | method prototype identifiers list. These are identifiers for all prototypes referred to by this file. This list must be sorted in return-type (by `type_id` index) major order, and then by arguments (also by `type_id` index). |
| field_ids | field_id_item[] | field identifiers list. These are identifiers for all fields referred to by this file, whether defined in the file or not. This list must be sorted, where the defining type (by `type_id` index) is the major order, field name (by `string_id` index) is the intermediate order, and type (by `type_id` index) is the minor order. |
| method_ids | method_id_item[] | method identifiers list. These are identifiers for all methods referred to by this file, whether defined in the file or not. This list must be sorted, where the defining type (by `type_id` index) is the major order, method name (by `string_id` index) is the intermediate |

| Name | Format | Description |
|---|---|---|
| | | order, and method prototype (by `proto_id` index) is the minor order. |
| class_defs | class_def_item[] | class definitions list. The classes must be ordered such that a given class's superclass and implemented interfaces appear in the list earlier than the referring class. |
| data | ubyte[] | data area, containing all the support data for the tables listed above. Different items have different alignment requirements, and padding bytes are inserted before each item if necessary to achieve proper alignment. |
| link_data | ubyte[] | data used in statically linked files. The format of the data in this section is left unspecified by this document; this section is empty in unlinked files, and runtime implementations may use it as they see fit. |

# Bitfield, String, and Constant Definitions

## DEX_FILE_MAGIC

*embedded in header_item*

The constant array/string `DEX_FILE_MAGIC` is the list of bytes that must appear at the beginning of a `.dex` file in order for it to be recognized as such. The value intentionally contains a newline (`"\n"` or `0x0a`) and a null byte (`"\0"` or `0x00`) in order to help in the detection of certain forms of corruption. The value also encodes a format version number as three decimal digits, which is expected to increase monotonically over time as the format evolves.

```
ubyte[8] DEX_FILE_MAGIC = { 0x64 0x65 0x78 0x0a 0x30 0x33 0x35 0x00 }
                        = "dex\n035\0"
```

**Note:** At least a couple earlier versions of the format have been used in widely-available public software releases. For example, version `009` was used for the M3 releases of the Android platform (November-December 2007), and version `013` was used for the M5 releases of the Android platform (February-March 2008). In several respects, these earlier versions of the format differ significantly from the version described in this document.

## ENDIAN_CONSTANT and REVERSE_ENDIAN_CONSTANT

*embedded in header_item*

The constant `ENDIAN_CONSTANT` is used to indicate the endianness of the file in which it is found. Although the standard `.dex` format is little-endian, implementations may choose to perform byte-swapping. Should an implementation come across a header whose `endian_tag` is `REVERSE_ENDIAN_CONSTANT` instead of `ENDIAN_CONSTANT`, it would know that the file has been byte-swapped from the expected form.

138dd

```
uint ENDIAN_CONSTANT = 0x12345678;
uint REVERSE_ENDIAN_CONSTANT = 0x78563412;
```

## NO_INDEX

*embedded in `class_def_item` and `debug_info_item`*

The constant `NO_INDEX` is used to indicate that an index value is absent.

**Note:** This value isn't defined to be `0`, because that is in fact typically a valid index.

**Also Note:** The chosen value for `NO_INDEX` is representable as a single byte in the `uleb128p1` encoding.

```
uint NO_INDEX = 0xffffffff;    // == -1 if treated as a signed int
```

## `access_flags` Definitions

*embedded in `class_def_item`, `field_item`, `method_item`, and `InnerClass`*

Bitfields of these flags are used to indicate the accessibility and overall properties of classes and class members.

| Name | Value | For Classes (and `InnerClass` annotations) | For Fields | For Methods |
|---|---|---|---|---|
| ACC_PUBLIC | 0x1 | `public`: visible everywhere | `public`: visible everywhere | `public`: visible everywhere |
| ACC_PRIVATE | 0x2 | * `private`: only visible to defining class | `private`: only visible to defining class | `private`: only visible to defining class |
| ACC_PROTECTED | 0x4 | * `protected`: visible to package and subclasses | `protected`: visible to package and subclasses | `protected`: visible to package and subclasses |
| ACC_STATIC | 0x8 | * `static`: is not constructed with an outer `this` reference | `static`: global to defining class | `static`: does not take a `this` argument |
| ACC_FINAL | 0x10 | `final`: not subclassable | `final`: immutable after construction | `final`: not overridable |
| *(unused)* | 0x20 | | | |
| ACC_VOLATILE | 0x40 | | `volatile`: special access rules to help with thread safety | |
| ACC_BRIDGE | 0x40 | | | bridge method, added automatically by compiler as a type-safe bridge |
| ACC_TRANSIENT | 0x80 | | `transient`: not to be saved by default serialization | |

138de

| Name | Value | For Classes (and `InnerClass` annotations) | For Fields | For Methods |
|---|---|---|---|---|
| ACC_VARARGS | 0x80 | | | last argument should be treated as a "rest" argument by compiler |
| ACC_NATIVE | 0x100 | | | `native`: implemented in native code |
| ACC_INTERFACE | 0x200 | `interface`: multiply-implementable abstract class | | |
| ACC_ABSTRACT | 0x400 | `abstract`: not directly instantiable | | `abstract`: unimplemented by this class |
| ACC_STRICT | 0x800 | | | `strictfp`: strict rules for floating-point arithmetic |
| ACC_SYNTHETIC | 0x1000 | not directly defined in source code | not directly defined in source code | not directly defined in source code |
| ACC_ANNOTATION | 0x2000 | declared as an annotation class | | |
| ACC_ENUM | 0x4000 | declared as an enumerated type | declared as an enumerated value | |
| *(unused)* | 0x8000 | | | |
| ACC_CONSTRUCTOR | 0x10000 | | | constructor method (class or instance initializer) |

* Only allowed on for `InnerClass` annotations, and must not ever be on in a `class_def_item`.

# MUTF-8 (Modified UTF-8) Encoding

As a concession to easier legacy support, the `.dex` format encodes its string data in a de facto standard modified UTF-8 form, hereafter referred to as MUTF-8. This form is identical to standard UTF-8, except:

- Only the one-, two-, and three-byte encodings are used.
- Code points in the range `U+10000` … `U+10ffff` are encoded as a surrogate pair, each of which is represented as a three-byte encoded value.
- The code point `U+0000` is encoded in two-byte form.
- A plain null byte (value `0`) indicates the end of a string, as is the standard C language interpretation.

The first two items above can be summarized as: MUTF-8 is an encoding format for UTF-16, instead of being a more direct encoding format for Unicode characters.

The final two items above make it simultaneously possible to include the code point `U+0000` in a string *and* still manipulate it as a C-style null-terminated string.

However, the special encoding of `U+0000` means that, unlike normal UTF-8, the result of calling the standard C function `strcmp()` on a pair of MUTF-8 strings does not always indicate the properly signed result of comparison of *unequal* strings. When ordering (not just equality) is a concern, the most straightforward way to compare MUTF-8 strings is to decode them character by character, and compare the decoded values. (However, more clever implementations are also possible.)

138df

Please refer to The Unicode Standard for further information about character encoding. MUTF-8 is actually closer to the (relatively less well-known) encoding CESU-8 than to UTF-8 per se.

## encoded_value Encoding

*embedded in annotation_element and encoded_array_item*

An encoded_value is an encoded piece of (nearly) arbitrary hierarchically structured data. The encoding is meant to be both compact and straightforward to parse.

| Name | Format | Description |
|---|---|---|
| (value_arg << 5) \| value_type | ubyte | byte indicating the type of the immediately subsequent value along with an optional clarifying argument in the high-order three bits. See below for the various value definitions. In most cases, value_arg encodes the length of the immediately-subsequent value in bytes, as (size - 1), e.g., 0 means that the value requires one byte, and 7 means it requires eight bytes; however, there are exceptions as noted below. |
| value | ubyte[] | bytes representing the value, variable in length and interpreted differently for different value_type bytes, though always little-endian. See the various value definitions below for details. |

## Value Formats

| Type Name | value_type | value_arg Format | value Format | Description |
|---|---|---|---|---|
| VALUE_BYTE | 0x00 | *(none; must be 0)* | ubyte[1] | signed one-byte integer value |
| VALUE_SHORT | 0x02 | size - 1 (0…1) | ubyte[size] | signed two-byte integer value, sign-extended |
| VALUE_CHAR | 0x03 | size - 1 (0…1) | ubyte[size] | unsigned two-byte integer value, zero-extended |
| VALUE_INT | 0x04 | size - 1 (0…3) | ubyte[size] | signed four-byte integer value, sign-extended |
| VALUE_LONG | 0x06 | size - 1 (0…7) | ubyte[size] | signed eight-byte integer value, sign-extended |
| VALUE_FLOAT | 0x10 | size - 1 (0…3) | ubyte[size] | four-byte bit pattern, zero-extended *to the right*, and interpreted as an IEEE754 32-bit floating point value |
| VALUE_DOUBLE | 0x11 | size - 1 (0…7) | ubyte[size] | eight-byte bit pattern, zero-extended *to the right*, and interpreted as an IEEE754 64-bit floating point value |
| VALUE_STRING | 0x17 | size - 1 (0…3) | ubyte[size] | unsigned (zero-extended) four-byte integer value, interpreted as an index into the string_ids section and representing a string value |
| VALUE_TYPE | 0x18 | size - 1 (0…3) | ubyte[size] | unsigned (zero-extended) four-byte integer value, interpreted as an index into the type_ids section and representing a reflective type/class value |

138 dg

| Type Name | value_type | value_arg Format | value Format | Description |
|---|---|---|---|---|
| VALUE_FIELD | 0x19 | size - 1 (0…3) | ubyte[size] | unsigned (zero-extended) four-byte integer value, interpreted as an index into the field_ids section and representing a reflective field value |
| VALUE_METHOD | 0x1a | size - 1 (0…3) | ubyte[size] | unsigned (zero-extended) four-byte integer value, interpreted as an index into the method_ids section and representing a reflective method value |
| VALUE_ENUM | 0x1b | size - 1 (0…3) | ubyte[size] | unsigned (zero-extended) four-byte integer value, interpreted as an index into the field_ids section and representing the value of an enumerated type constant |
| VALUE_ARRAY | 0x1c | *(none; must be 0)* | encoded_array | an array of values, in the format specified by "encoded_array Format" below. The size of the value is implicit in the encoding. |
| VALUE_ANNOTATION | 0x1d | *(none; must be 0)* | encoded_annotation | a sub-annotation, in the format specified by "encoded_annotation Format" below. The size of the value is implicit in the encoding. |
| VALUE_NULL | 0x1e | *(none; must be 0)* | *(none)* | null reference value |
| VALUE_BOOLEAN | 0x1f | boolean (0…1) | *(none)* | one-bit value; 0 for false and 1 for true. The bit is represented in the value_arg. |

## encoded_array Format

| Name | Format | Description |
|---|---|---|
| size | uleb128 | number of elements in the array |
| values | encoded_value[size] | a series of size encoded_value byte sequences in the format specified by this section, concatenated sequentially. |

## encoded_annotation Format

| Name | Format | Description |
|---|---|---|
| type_idx | uleb128 | type of the annotation. This must be a class (not array or primitive) type. |
| size | uleb128 | number of name-value mappings in this annotation |
| elements | annotation_element[size] | elements of the annotataion, represented directly in-line (not as offsets). Elements must be sorted in increasing order by string_id index. |

## annotation_element Format

138dh

| Name | Format | Description |
|---|---|---|
| `name_idx` | `uleb128` | element name, represented as an index into the `string_ids` section. The string must conform to the syntax for *MemberName*, defined above. |
| `value` | `encoded_value` | element value |

# String Syntax

There are several kinds of item in a `.dex` file which ultimately refer to a string. The following BNF-style definitions indicate the acceptable syntax for these strings.

## *SimpleName*

A *SimpleName* is the basis for the syntax of the names of other things. The `.dex` format allows a fair amount of latitude here (much more than most common source languages). In brief, a simple name may consist of any low-ASCII alphabetic character or digit, a few specific low-ASCII symbols, and most non-ASCII code points that are not control, space, or special characters. Note that surrogate code points (in the range `U+d800` ... `U+dfff`) are not considered valid name characters, per se, but Unicode supplemental characters *are* valid (which are represented by the final alternative of the rule for *SimpleNameChar*), and they should be represented in a file as pairs of surrogate code points in the MUTF-8 encoding.

```
SimpleName →
    SimpleNameChar (SimpleNameChar)*
SimpleNameChar →
    'A' ... 'Z'
  | 'a' ... 'z'
  | '0' ... '9'
  | '$'
  | '-'
  | '_'
  | U+00a1 ... U+1fff
  | U+2010 ... U+2027
  | U+2030 ... U+d7ff
  | U+e000 ... U+ffef
  | U+10000 ... U+10ffff
```

## *MemberName*

*used by `field_id_item` and `method_id_item`*

A *MemberName* is the name of a member of a class, members being fields, methods, and inner classes.

```
MemberName →
    SimpleName
  | '<' SimpleName '>'
```

138d;

## *FullClassName*

A *FullClassName* is a fully-qualified class name, including an optional package specifier followed by a required name.

*FullClassName* →
    *OptionalPackagePrefix SimpleName*

*OptionalPackagePrefix* →
    (*SimpleName* '/')*

## *TypeDescriptor*

*used by* type_id_item

A *TypeDescriptor* is the representation of any type, including primitives, classes, arrays, and `void`. See below for the meaning of the various versions.

*TypeDescriptor* →
    'V'
  | *FieldTypeDescriptor*

*FieldTypeDescriptor* →
    *NonArrayFieldTypeDescriptor*
  | ('[' * 1...255) *NonArrayFieldTypeDescriptor*

*NonArrayFieldTypeDescriptor* →
    'Z'
  | 'B'
  | 'S'
  | 'C'
  | 'I'
  | 'J'
  | 'F'
  | 'D'
  | 'L' *FullClassName* ';'

## *ShortyDescriptor*

*used by* proto_id_item

A *ShortyDescriptor* is the short form representation of a method prototype, including return and parameter types, except that there is no distinction between various reference (class or array) types. Instead, all reference types are represented by a single `'L'` character.

*ShortyDescriptor* →
    *ShortyReturnType* (*ShortyFieldType*)*

*ShortyReturnType* →
    'V'
  | *ShortyFieldType*

*ShortyFieldType* →
    'Z'
  | 'B'
  | 'S'
  | 'C'

138dj

| | |
|---|---|
| I | 'I' |
| J | 'J' |
| F | 'F' |
| D | 'D' |
| L | 'L' |

## *TypeDescriptor* Semantics

This is the meaning of each of the variants of *TypeDescriptor*.

| Syntax | Meaning |
|---|---|
| V | void; only valid for return types |
| Z | boolean |
| B | byte |
| S | short |
| C | char |
| I | int |
| J | long |
| F | float |
| D | double |
| L*fully/qualified/Name*; | the class *fully.qualified.Name* |
| [*descriptor* | array of *descriptor*, usable recursively for arrays-of-arrays, though it is invalid to have more than 255 dimensions. |

# Items and Related Structures

This section includes definitions for each of the top-level items that may appear in a .dex file.

### header_item

*appears in the header section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| magic | ubyte[8] = DEX_FILE_MAGIC | magic value. See discussion above under "DEX_FILE_MAGIC" for more details. |
| checksum | uint | adler32 checksum of the rest of the file (everything but magic and this field); used to detect file corruption |

138dk

## map_list

*appears in the data section*
*referenced from header_item*
*alignment: 4 bytes*

This is a list of the entire contents of a file, in order. It contains some redundancy with respect to the header_item but is intended to be an easy form to use to iterate over an entire file. A given type may appear at most once in a map, but there is no restriction on what order types may appear in, other than the restrictions implied by the rest of the format (e.g., a header section must appear first, followed by a string_ids section, etc.). Additionally, the map entries must be ordered by initial offset and must not overlap.

| Name | Format | Description |
|---|---|---|
| size | uint | size of the list, in entries |
| list | map_item[size] | elements of the list |

## map_item Format

| Name | Format | Description |
|---|---|---|
| type | ushort | type of the items; see table below |
| unused | ushort | *(unused)* |
| size | uint | count of the number of items to be found at the indicated offset |
| offset | uint | offset from the start of the file to the items in question |

## Type Codes

| Item Type | Constant | Value | Item Size In Bytes |
|---|---|---|---|
| header_item | TYPE_HEADER_ITEM | 0x0000 | 0x70 |
| string_id_item | TYPE_STRING_ID_ITEM | 0x0001 | 0x04 |
| type_id_item | TYPE_TYPE_ID_ITEM | 0x0002 | 0x04 |
| proto_id_item | TYPE_PROTO_ID_ITEM | 0x0003 | 0x0c |
| field_id_item | TYPE_FIELD_ID_ITEM | 0x0004 | 0x08 |
| method_id_item | TYPE_METHOD_ID_ITEM | 0x0005 | 0x08 |
| class_def_item | TYPE_CLASS_DEF_ITEM | 0x0006 | 0x20 |
| map_list | TYPE_MAP_LIST | 0x1000 | 4 + (item.size * 12) |
| type_list | TYPE_TYPE_LIST | 0x1001 | 4 + (item.size * 2) |

138dL

| Item Type | Constant | Value | Item Size In Bytes |
|---|---|---|---|
| annotation_set_ref_list | TYPE_ANNOTATION_SET_REF_LIST | 0x1002 | 4 + (item.size * 4) |
| annotation_set_item | TYPE_ANNOTATION_SET_ITEM | 0x1003 | 4 + (item.size * 4) |
| class_data_item | TYPE_CLASS_DATA_ITEM | 0x2000 | *implicit; must parse* |
| code_item | TYPE_CODE_ITEM | 0x2001 | *implicit; must parse* |
| string_data_item | TYPE_STRING_DATA_ITEM | 0x2002 | *implicit; must parse* |
| debug_info_item | TYPE_DEBUG_INFO_ITEM | 0x2003 | *implicit; must parse* |
| annotation_item | TYPE_ANNOTATION_ITEM | 0x2004 | *implicit; must parse* |
| encoded_array_item | TYPE_ENCODED_ARRAY_ITEM | 0x2005 | *implicit; must parse* |
| annotations_directory_item | TYPE_ANNOTATIONS_DIRECTORY_ITEM | 0x2006 | *implicit; must parse* |

## string_id_item

*appears in the string_ids section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| string_data_off | uint | offset from the start of the file to the string data for this item. The offset should be to a location in the data section, and the data should be in the format specified by "string_data_item" below. There is no alignment requirement for the offset. |

## string_data_item

*appears in the data section*
*alignment: none (byte-aligned)*

| Name | Format | Description |
|---|---|---|
| utf16_size | uleb128 | size of this string, in UTF-16 code units (which is the "string length" in many systems). That is, this is the decoded length of the string. (The encoded length is implied by the position of the 0 byte.) |
| data | ubyte[] | a series of MUTF-8 code units (a.k.a. octets, a.k.a. bytes) followed by a byte of value 0. See "MUTF-8 (Modified UTF-8) Encoding" above for details and discussion about the data format.<br><br>**Note:** It is acceptable to have a string which includes (the encoded form of) UTF-16 surrogate code units (that is, U+d800 … U+dfff) either in isolation or out-of-order with respect to the usual encoding of Unicode into UTF-16. It is up to higher-level uses of strings to reject such invalid encodings, if appropriate. |

138dm

## type_id_item

*appears in the type_ids section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| descriptor_idx | uint | index into the string_ids list for the descriptor string of this type. The string must conform to the syntax for *TypeDescriptor*, defined above. |

## proto_id_item

*appears in the proto_ids section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| shorty_idx | uint | index into the string_ids list for the short-form descriptor string of this prototype. The string must conform to the syntax for *ShortyDescriptor*, defined above, and must correspond to the return type and parameters of this item. |
| return_type_idx | uint | index into the type_ids list for the return type of this prototype |
| parameters_off | uint | offset from the start of the file to the list of parameter types for this prototype, or 0 if this prototype has no parameters. This offset, if non-zero, should be in the data section, and the data there should be in the format specified by "type_list" below. Additionally, there should be no reference to the type void in the list. |

## field_id_item

*appears in the field_ids section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| class_idx | ushort | index into the type_ids list for the definer of this field. This must be a class type, and not an array or primitive type. |
| type_idx | ushort | index into the type_ids list for the type of this field |
| name_idx | uint | index into the string_ids list for the name of this field. The string must conform to the syntax for *MemberName*, defined above. |

138dn

## method_id_item

*appears in the method_ids section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| class_idx | ushort | index into the `type_ids` list for the definer of this method. This must be a class or array type, and not a primitive type. |
| proto_idx | ushort | index into the `proto_ids` list for the prototype of this method |
| name_idx | uint | index into the `string_ids` list for the name of this method. The string must conform to the syntax for *MemberName*, defined above. |

## class_def_item

*appears in the class_defs section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| class_idx | uint | index into the `type_ids` list for this class. This must be a class type, and not an array or primitive type. |
| access_flags | uint | access flags for the class (`public`, `final`, etc.). See "`access_flags` Definitions" for details. |
| superclass_idx | uint | index into the `type_ids` list for the superclass, or the constant value `NO_INDEX` if this class has no superclass (i.e., it is a root class such as `Object`). If present, this must be a class type, and not an array or primitive type. |
| interfaces_off | uint | offset from the start of the file to the list of interfaces, or `0` if there are none. This offset should be in the `data` section, and the data there should be in the format specified by "`type_list`" below. Each of the elements of the list must be a class type (not an array or primitive type), and there must not be any duplicates. |
| source_file_idx | uint | index into the `string_ids` list for the name of the file containing the original source for (at least most of) this class, or the special value `NO_INDEX` to represent a lack of this information. The `debug_info_item` of any given method may override this source file, but the expectation is that most classes will only come from one source file. |
| annotations_off | uint | offset from the start of the file to the annotations structure for this class, or `0` if there are no annotations on this class. This offset, if non-zero, should be in the `data` section, and the data there should be in the format specified by "`annotations_directory_item`" below, with all items referring to this class as the definer. |
| class_data_off | uint | offset from the start of the file to the associated class data for this item, or `0` if there is no class data for this class. (This may be the case, for example, if this class is a marker interface.) The offset, if non-zero, should be in the `data` section, and the data there should be in the format specified by "`class_data_item`" below, with all items referring to this class as the definer. |
| static_values_off | uint | offset from the start of the file to the list of initial values for `static` fields, or `0` if there are none (and all `static` fields are to be initialized with `0` or `null`). This offset should be in the `data` section, and the data there should be in the format specified by |

| Name | Format | Description |
|---|---|---|
| | | "encoded_array_item" below. The size of the array must be no larger than the number of static fields declared by this class, and the elements correspond to the static fields in the same order as declared in the corresponding field_list. The type of each array element must match the declared type of its corresponding field. If there are fewer elements in the array than there are static fields, then the leftover fields are initialized with a type-appropriate 0 or null. |

## class_data_item

*referenced from class_def_item*
*appears in the data section*
*alignment: none (byte-aligned)*

| Name | Format | Description |
|---|---|---|
| static_fields_size | uleb128 | the number of static fields defined in this item |
| instance_fields_size | uleb128 | the number of instance fields defined in this item |
| direct_methods_size | uleb128 | the number of direct methods defined in this item |
| virtual_methods_size | uleb128 | the number of virtual methods defined in this item |
| static_fields | encoded_field[static_fields_size] | the defined static fields, represented as a sequence of encoded elements. The fields must be sorted by field_idx in increasing order. |
| instance_fields | encoded_field[instance_fields_size] | the defined instance fields, represented as a sequence of encoded elements. The fields must be sorted by field_idx in increasing order. |
| direct_methods | encoded_method[direct_methods_size] | the defined direct (any of static, private, or constructor) methods, represented as a sequence of encoded elements. The methods must be sorted by method_idx in increasing order. |
| virtual_methods | encoded_method[virtual_methods_size] | the defined virtual (none of static, private, or constructor) methods, represented as a sequence of encoded elements. This list should *not* include inherited methods unless overridden by the class that this item represents. The methods must be sorted by method_idx in increasing order. |

138dp

**Note:** All elements' field_ids and method_ids must refer to the same defining class.

### encoded_field Format

| Name | Format | Description |
|---|---|---|
| field_idx_diff | uleb128 | index into the field_ids list for the identity of this field (includes the name and descriptor), represented as a difference from the index of previous element in the list. The index of the first element in a list is represented directly. |
| access_flags | uleb128 | access flags for the field (public, final, etc.). See "access_flags Definitions" for details. |

### encoded_method Format

| Name | Format | Description |
|---|---|---|
| method_idx_diff | uleb128 | index into the method_ids list for the identity of this method (includes the name and descriptor), represented as a difference from the index of previous element in the list. The index of the first element in a list is represented directly. |
| access_flags | uleb128 | access flags for the method (public, final, etc.). See "access_flags Definitions" for details. |
| code_off | uleb128 | offset from the start of the file to the code structure for this method, or 0 if this method is either abstract or native. The offset should be to a location in the data section. The format of the data is specified by "code_item" below. |

## type_list

*referenced from class_def_item and proto_id_item*
*appears in the data section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| size | uint | size of the list, in entries |
| list | type_item[size] | elements of the list |

### type_item Format

| Name | Format | Description |
|---|---|---|
| type_idx | ushort | index into the type_ids list |

138d9

## code_item

*referenced from method_item*
*appears in the data section*
*alignment: 4 bytes*

| Name | Format | Description |
|---|---|---|
| registers_size | ushort | the number of registers used by this code |
| ins_size | ushort | the number of words of incoming arguments to the method that this code is for |
| outs_size | ushort | the number of words of outgoing argument space required by this code for method invocation |
| tries_size | ushort | the number of try_items for this instance. If non-zero, then these appear as the tries array just after the insns in this instance. |
| debug_info_off | uint | offset from the start of the file to the debug info (line numbers + local variable info) sequence for this code, or 0 if there simply is no information. The offset, if non-zero, should be to a location in the data section. The format of the data is specified by "debug_info_item" below. |
| insns_size | uint | size of the instructions list, in 16-bit code units |
| insns | ushort[insns_size] | actual array of bytecode. The format of code in an insns array is specified by the companion document "Bytecode for the Dalvik VM". Note that though this is defined as an array of ushort, there are some internal structures that prefer four-byte alignment. Also, if this happens to be in an endian-swapped file, then the swapping is *only* done on individual ushorts and not on the larger internal structures. |
| padding | ushort *(optional)* = 0 | two bytes of padding to make tries four-byte aligned. This element is only present if tries_size is non-zero and insns_size is odd. |
| tries | try_item[tries_size] *(optional)* | array indicating where in the code exceptions may be caught and how to handle them. Elements of the array must be non-overlapping in range and in order from low to high address. This element is only present if tries_size is non-zero. |
| handlers | encoded_catch_handler_list *(optional)* | bytes representing a list of lists of catch types and associated handler addresses. Each try_item has a byte-wise offset into this structure. This element is only present if tries_size is non-zero. |

## try_item Format

| Name | Format | Description |
|---|---|---|
| start_addr | uint | start address of the block of code covered by this entry. The address is a count of 16-bit code units to the start of the first covered instruction. |
| insn_count | ushort | number of 16-bit code units covered by this entry. The last code unit covered (inclusive) is start_addr + insn_count - 1. |
| handler_off | ushort | offset in bytes from the start of the associated encoded handler data to the catch_handler_item for this entry |

## encoded_catch_handler_list Format

| Name | Format | Description |
|---|---|---|
| size | uleb128 | size of this list, in entries |
| list | encoded_catch_handler[handlers_size] | actual list of handler lists, represented directly (not as offsets), and concatenated sequentially |

## encoded_catch_handler Format

| Name | Format | Description |
|---|---|---|
| size | sleb128 | number of catch types in this list. If non-positive, then this is the negative of the number of catch types, and the catches are followed by a catch-all handler. For example: A `size` of `0` means that there is a catch-all but no explicitly typed catches. A `size` of `2` means that there are two explicitly typed catches and no catch-all. And a `size` of `-1` means that there is one typed catch along with a catch-all. |
| handlers | encoded_type_addr_pair[abs(size)] | stream of `abs(size)` encoded items, one for each caught type, in the order that the types should be tested. |
| catch_all_addr | uleb128 *(optional)* | bytecode address of the catch-all handler. This element is only present if `size` is non-positive. |

## encoded_type_addr_pair Format

| Name | Format | Description |
|---|---|---|
| type_idx | uleb128 | index into the `type_ids` list for the type of the exception to catch |
| addr | uleb128 | bytecode address of the associated exception handler |

# debug_info_item

*referenced from `code_item`*
*appears in the data section*
*alignment: none (byte-aligned)*

Each `debug_info_item` defines a DWARF3-inspired byte-coded state machine that, when interpreted, emits the positions table and (potentially) the local variable information for a `code_item`. The sequence begins with a variable-length header (the length of which depends on the number of method parameters), is followed by the state machine bytecodes, and ends with an `DBG_END_SEQUENCE` byte.

The state machine consists of five registers. The `address` register represents the instruction offset in the

associated insns_item in 16-bit code units. The address register starts at 0 at the beginning of each debug_info sequence and may only monotonically increase. The line register represents what source line number should be associated with the next positions table entry emitted by the state machine. It is initialized in the sequence header, and may change in positive or negative directions but must never be less than 1. The source_file register represents the source file that the line number entries refer to. It is initialized to the value of source_file_idx in class_def_item. The other two variables, prologue_end and epilogue_begin, are boolean flags (initialized to false) that indicate whether the next position emitted should be considered a method prologue or epilogue. The state machine must also track the name and type of the last local variable live in each register for the DBG_RESTART_LOCAL code.

The header is as follows:

| Name | Format | Description |
|---|---|---|
| line_start | uleb128 | the initial value for the state machine's line register. Does not represent an actual positions entry. |
| parameters_size | uleb128 | the number of parameter names that are encoded. There should be one per method parameter, excluding an instance method's this, if any. |
| parameter_names | uleb128p1[parameters_size] | string index of the method parameter name. An encoded value of NO_INDEX indicates that no name is available for the associated parameter. The type descriptor and signature are implied from the method descriptor and signature. |

The byte code values are as follows:

| Name | Value | Format | Arguments | Description |
|---|---|---|---|---|
| DBG_END_SEQUENCE | 0x00 | | *(none)* | terminates a debug info seque code_item |
| DBG_ADVANCE_PC | 0x01 | uleb128 addr_diff | addr_diff: amount to add to address register | advances the address registe emitting a positions entry |
| DBG_ADVANCE_LINE | 0x02 | sleb128 line_diff | line_diff: amount to change line register by | advances the line register wit a positions entry |
| DBG_START_LOCAL | 0x03 | uleb128 register_num<br>uleb128p1 name_idx<br>uleb128p1 type_idx | register_num: register that will contain local<br>name_idx: string index of the name<br>type_idx: type index of the type | introduces a local variable at t address. Either name_idx o type_idx may be NO_IND indicate that that value is unkn |
| DBG_START_LOCAL_EXTENDED | 0x04 | uleb128 register_num<br>uleb128p1 name_idx<br>uleb128p1 type_idx<br>uleb128p1 sig_idx | register_num: register that will contain local<br>name_idx: string index of the name<br>type_idx: type index of the type<br>sig_idx: string index of the type signature | introduces a local with a type the current address. Any of n type_idx, or sig_idx ma NO_INDEX to indicate that th unknown. (If sig_idx is -1, same data could be represent efficiently using the opcode DBG_START_LOCAL.)<br>**Note:** See the discussion unc "dalvik.annotation.S below for caveats about handl signatures. |

| Name | Value | Format | Arguments | Description |
|---|---|---|---|---|
| DBG_END_LOCAL | 0x05 | uleb128 register_num | register_num: register that contained local | marks a currently-live local v out of scope at the current adc |
| DBG_RESTART_LOCAL | 0x06 | uleb128 register_num | register_num: register to restart | re-introduces a local variable current address. The name ar the same as the last local thai the specified register. |
| DBG_SET_PROLOGUE_END | 0x07 | | *(none)* | sets the prologue_end sta register, indicating that the ne: entry that is added should be c the end of a method prologue appropriate place for a metho breakpoint). The prologue_ register is cleared by any spe 0x0a) opcode. |
| DBG_SET_EPILOGUE_BEGIN | 0x08 | | *(none)* | sets the epilogue_begin machine register, indicating th position entry that is added sh considered the beginning of a epilogue (an appropriate place execution before method exit) epilogue_begin register by any special (>= 0x0a) op |
| DBG_SET_FILE | 0x09 | uleb128p1 name_idx | name_idx: string index of source file name; NO_INDEX if unknown | indicates that all subsequent li entries make reference to this name, instead of the default na specified in code_item |
| *Special Opcodes* | 0x0a...0xff | | *(none)* | advances the line and add registers, emits a position ent clears prologue_end and epilogue_begin. See bel description. |

## Special Opcodes

Opcodes with values between 0x0a and 0xff (inclusive) move both the line and address registers by a small amount and then emit a new position table entry. The formula for the increments are as follows:

```
DBG_FIRST_SPECIAL = 0x0a  // the smallest special opcode
DBG_LINE_BASE   = -4      // the smallest line number increment
DBG_LINE_RANGE  = 15      // the number of line increments represented

adjusted_opcode = opcode - DBG_FIRST_SPECIAL

line += DBG_LINE_BASE + (adjusted_opcode % DBG_LINE_RANGE)
address += (adjusted_opcode / DBG_LINE_RANGE)
```

## annotations_directory_item

*referenced from class_def_item*
*appears in the data section*
*alignment: 4 bytes*

138dv

| Name | Format | Description |
| --- | --- | --- |
| class_annotations_off | uint | offset from the start of the file to the annotations made directly on the class, or 0 if the class has no direct annotations. The offset, if non-zero, should be to a location in the data section. The format of the data is specified by "annotation_set_item" below. |
| fields_size | uint | count of fields annotated by this item |
| annotated_methods_off | uint | count of methods annotated by this item |
| annotated_parameters_off | uint | count of method parameter lists annotated by this item |
| field_annotations | field_annotation[fields_size] *(optional)* | list of associated field annotations. The elements of the list must be sorted in increasing order, by field_idx. |
| method_annotations | method_annotation[methods_size] *(optional)* | list of associated method annotations. The elements of the list must be sorted in increasing order, by method_idx. |
| parameter_annotations | parameter_annotation[parameters_size] *(optional)* | list of associated method parameter annotations. The elements of the list must be sorted in increasing order, by method_idx. |

**Note:** All elements' field_ids and method_ids must refer to the same defining class.

## field_annotation Format

| Name | Format | Description |
| --- | --- | --- |
| field_idx | uint | index into the field_ids list for the identity of the field being annotated |
| annotations_off | uint | offset from the start of the file to the list of annotations for the field. The offset should be to a location in the data section. The format of the data is specified by "annotation_set_item" below. |

## method_annotation Format

| Name | Format | Description |
| --- | --- | --- |
| method_idx | uint | index into the method_ids list for the identity of the method being annotated |

138dv

| Name | Format | Description |
| --- | --- | --- |
| annotations_off | uint | offset from the start of the file to the list of annotations for the method. The offset should be to a location in the data section. The format of the data is specified by "annotation_set_item" below. |

### parameter_annotation Format

| Name | Format | Description |
| --- | --- | --- |
| method_idx | uint | index into the method_ids list for the identity of the method whose parameters are being annotated |
| annotations_off | uint | offset from the start of the file to the list of annotations for the method parameters. The offset should be to a location in the data section. The format of the data is specified by "annotation_set_ref_list" below. |

## annotation_set_ref_list

*referenced from parameter_annotations_item*
*appears in the data section*
*alignment: 4 bytes*

| Name | Format | Description |
| --- | --- | --- |
| size | uint | size of the list, in entries |
| list | annotation_set_ref_item[size] | elements of the list |

### annotation_set_ref_item Format

| Name | Format | Description |
| --- | --- | --- |
| annotations_off | uint | offset from the start of the file to the referenced annotation set or 0 if there are no annotations for this element. The offset, if non-zero, should be to a location in the data section. The format of the data is specified by "annotation_set_item" below. |

## annotation_set_item

*referenced from annotations_directory_item, field_annotations_item, method_annotations_item, and annotation_set_ref_item*
*appears in the data section*

138dw

*alignment: 4 bytes*

| Name | Format | Description |
| --- | --- | --- |
| size | uint | size of the set, in entries |
| entries | annotation_off_item[size] | elements of the set. The elements must be sorted in increasing order, by type_idx. |

### annotation_off_item Format

| Name | Format | Description |
| --- | --- | --- |
| annotation_off | uint | offset from the start of the file to an annotation. The offset should be to a location in the data section, and the format of the data at that location is specified by "annotation_item" below. |

## annotation_item

*referenced from annotation_set_item*
*appears in the data section*
*alignment: none (byte-aligned)*

| Name | Format | Description |
| --- | --- | --- |
| visibility | ubyte | intended visibility of this annotation (see below) |
| annotation | encoded_annotation | encoded annotation contents, in the format described by "encoded_annotation Format" under "encoded_value Encoding" above. |

### Visibility values

These are the options for the visibility field in an annotation_item:

| Name | Value | Description |
| --- | --- | --- |
| VISIBILITY_BUILD | 0x00 | intended only to be visible at build time (e.g., during compilation of other code) |
| VISIBILITY_RUNTIME | 0x01 | intended to visible at runtime |
| VISIBILITY_SYSTEM | 0x02 | intended to visible at runtime, but only to the underlying system (and not to regular user code) |

138dx

### encoded_array_item

*referenced from class_def_item*
*appears in the data section*
*alignment: none (byte-aligned)*

| Name | Format | Description |
|---|---|---|
| value | encoded_array | bytes representing the encoded array value, in the format specified by "encoded_array Format" under "encoded_value Encoding" above. |

# System Annotations

System annotations are used to represent various pieces of reflective information about classes (and methods and fields). This information is generally only accessed indirectly by client (non-system) code.

System annotations are represented in `.dex` files as annotations with visibility set to `VISIBILITY_SYSTEM`.

### dalvik.annotation.AnnotationDefault

*appears on methods in annotation interfaces*

An `AnnotationDefault` annotation is attached to each annotation interface which wishes to indicate default bindings.

| Name | Format | Description |
|---|---|---|
| value | Annotation | the default bindings for this annotation, represented as an annotation of this type. The annotation need not include all names defined by the annotation; missing names simply do not have defaults. |

### dalvik.annotation.EnclosingClass

*appears on classes*

An `EnclosingClass` annotation is attached to each class which is either defined as a member of another class, per se, or is anonymous but not defined within a method body (e.g., a synthetic inner class). Every class that has this annotation must also have an `InnerClass` annotation. Additionally, a class may not have both an `EnclosingClass` and an `EnclosingMethod` annotation.

138d2

| Name | Format | Description |
| --- | --- | --- |
| value | Class[] | array of the member classes |

## dalvik.annotation.Signature

*appears on classes, fields, and methods*

A Signature annotation is attached to each class, field, or method which is defined in terms of a more complicated type than is representable by a type_id_item. The .dex format does not define the format for signatures; it is merely meant to be able to represent whatever signatures a source language requires for successful implementation of that language's semantics. As such, signatures are not generally parsed (or verified) by virtual machine implementations. The signatures simply get handed off to higher-level APIs and tools (such as debuggers). Any use of a signature, therefore, should be written so as not to make any assumptions about only receiving valid signatures, explicitly guarding itself against the possibility of coming across a syntactically invalid signature.

Because signature strings tend to have a lot of duplicated content, a Signature annotation is defined as an *array* of strings, where duplicated elements naturally refer to the same underlying data, and the signature is taken to be the concatenation of all the strings in the array. There are no rules about how to pull apart a signature into separate strings; that is entirely up to the tools that generate .dex files.

| Name | Format | Description |
| --- | --- | --- |
| value | String[] | the signature of this class or member, as an array of strings that is to be concatenated together |

## dalvik.annotation.Throws

*appears on methods*

A Throws annotation is attached to each method which is declared to throw one or more exception types.

| Name | Format | Description |
| --- | --- | --- |
| value | Class[] | the array of exception types thrown |

138dy

| Name | Format | Description |
|---|---|---|
| value | Class | the class which most closely lexically scopes this class |

## dalvik.annotation.EnclosingMethod

*appears on classes*

An `EnclosingMethod` annotation is attached to each class which is defined inside a method body. Every class that has this annotation must also have an `InnerClass` annotation. Additionally, a class may not have both an `EnclosingClass` and an `EnclosingMethod` annotation.

| Name | Format | Description |
|---|---|---|
| value | Method | the method which most closely lexically scopes this class |

## dalvik.annotation.InnerClass

*appears on classes*

An `InnerClass` annotation is attached to each class which is defined in the lexical scope of another class's definition. Any class which has this annotation must also have *either* an `EnclosingClass` annotation *or* an `EnclosingMethod` annotation.

| Name | Format | Description |
|---|---|---|
| name | String | the originally declared simple name of this class (not including any package prefix). If this class is anonymous, then the name is `null`. |
| accessFlags | int | the originally declared access flags of the class (which may differ from the effective flags because of a mismatch between the execution models of the source language and target virtual machine) |

## dalvik.annotation.MemberClasses

*appears on classes*

A `MemberClasses` annotation is attached to each class which declares member classes. (A member class is a direct inner class that has a name.)

138

# Implementation Techniques for Prolog

Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
andi@mips.complang.tuwien.ac.at

## Abstract

This paper is a short survey about currently used implementation techniques for Prolog. It gives an introduction to unification and resolution in Prolog and presents the memory model and a basic execution model. These models are expanded to the Vienna Abstract Machine (VAM) with its two versions, the $VAM_{2P}$ and the $VAM_{1P}$, and the most famous abstract machine, the Warren Abstract Machine (WAM). The continuation passing style model of Prolog, binary Prolog, leads to the BinWAM. Abstract interpretation can be applied to gather information about a program. This information is used in the generation of very specialized machine code and in optimizations like clause indexing and instruction scheduling on each kind of abstract machine.

## 1 Introduction

The implementation of Prolog has a long history [Col93]. Early systems were implemented by the group around Colmerauer in Marseille. The first system was an interpreter written in Algol by Phillip Roussel in 1972. With this experience a more efficient and usable system was developed by Gérard Battani, Henry Meloni and René Bazzoli [BM73]. It was a structure sharing interpreter and had essentially the same built-in predicates as modern Prolog systems. This system was reasonably efficient and convinced others of the usefulness of Prolog. Together with Fernande and Luis Pereira David Warren developed the DEC-10 Prolog, the first Prolog compiler [War77]. This compiler and the portable interpreter C-Prolog spread around the world and contributed to the success of Prolog. Further developments are described in [Roy94] and partly in this paper.

Section 2 presents a basic execution model for Prolog. This model helps to understand the Warren Abstract Machine described in section 3 and the Vienna Abstract Machine described in section 4. Section 5 gives on overview of optimizations.

## 2 A basic execution model

### 2.1 Introduction

The two basic parts of a Prolog interpreter are the unification part and the resolution part. The resolution is quite simple. It just implements a simplified SLD-resolution mechanism that searches the clauses top-down and evaluates the goals from left to right. This strategy immediately leads to the backtracking implementation and the usual layout of the data areas and stacks. Resolution handles stack frame allocation, calling of procedures, and backtracking.

Unification in Prolog is defined as follows:

- two constants unify if they are equal
- two structures unify if their functors (name and arity) are equal and all arguments unify
- two unbound variables unify and they are bound together
- an unbound variable and a constant or structure unify and the constant or structure is bound to the variable

This definition of unification determines the data representation. A thorough analysis of the recursive unification algorithm pays off because the interpreter spends most of the time in this part.

### 2.2 The representation of data

Since Prolog is not statically typed, the type and value of a variable can in general be determined only at run time. Therefore, a variable cell is divided into a value part and a tag part which determines the kind of the value. Fig. 1 shows a tagged value cell.

Basic data objects in Prolog are constants (atom and integer), structures and unbound variables. Since unbound variables can be bound together, there are

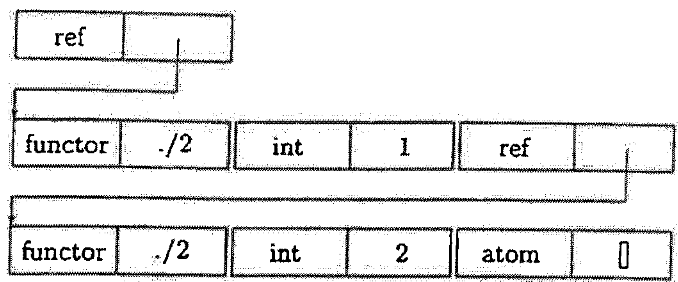



Figure 2: representation of X = 1.2.[]

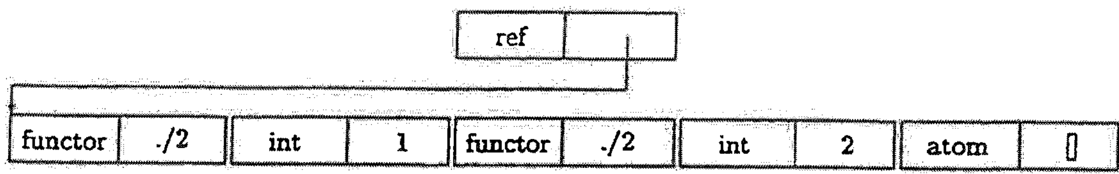



Figure 3: compacted representation of X = 1.2.[]

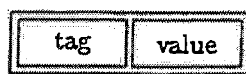



Figure 1: a tagged value cell

references between variables which are represented by pointers. To access a variable it can be necessary to follow the chain of references which is called dereferencing. Following tagged cells are needed in a Prolog system:

| | |
|---|---|
| atom | unique identifier of character string |
| integer | integer number |
| reference | pointer to another tagged cell |
| unbound | (self-reference pointer) |
| functor | name and arity of a structure followed by a tagged cell for each argument |

Most Prolog implementations do not use a separate tag for unbound variables. They represent unbound variables by a self reference. This can eliminate a tag check during unification of an unbound variable with another variable. The comparison can be replaced by an assignment. Since structures need more than one cell the variable cell contains a reference to the functor cell (see fig. 2). If the last cell of a structure is again a structure and the second structure is allocated directly after the first structure, the reference can be omitted (see fig. 3). This compact allocation can be obtained either at the first allocation or on garbage collection.

Another solution is a special reference tag for structures. The advantage of this method is that the type of a value cell can be determined without a memory access. Many implementations distinguish further between the empty list (nil) and other atoms, and between lists and other structures in order to allow more efficient implementations of lists. Big numbers and floating point numbers are represented as structures.

The tag field can be represented in different ways. It can be an additional memory cell of the standard word size, or it can be a small part of a memory cell. If the tag consists of some bits, the tag is either fixed-sized or variable-sized and uses the most or least significant part of the word.

Useful tag representations try to minimize the tag extraction and insertion overhead. An example is the use of zeroes in the least significant part of the word as an integer tag. Addition and subtraction can so be done without tag manipulation. Another example is to have the stack pointer displaced by the list tag, so that the allocation of list cells is free. A comprehensive study of tag representations can be found in [SH87].

Problems arise if a variable occurring inside a structure should be bound to this structure. In theorem provers in such a case the unification should fail. This test for occurrence of the variable in a structure, called occur check, is expensive. It is omitted in many unification algorithms employed by Prolog systems. If such a structure is assigned to a variable, a recursive structure is created. A simple unification algorithm would enter an infinite loop unifying two infinite structures. There exist linear time unification algorithms for infinite structures [Jaf84], but many Prolog systems do without it and create infinite structures, but cannot unify or print them.

141

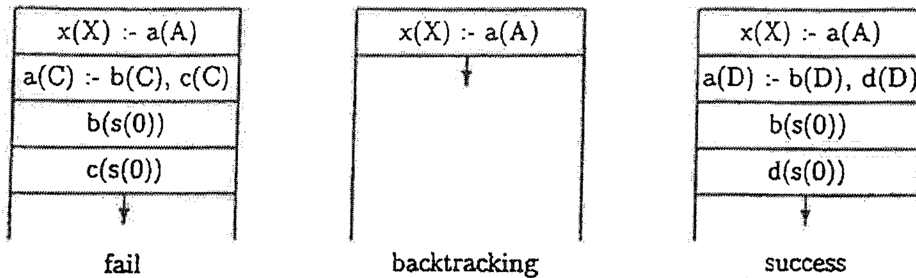



Figure 4: stacks

## 2.3 The data areas

Variables in a Prolog clause are stored in a stack frame similar to variables in a conventional programming language. The SLD-resolution was chosen as the resolution scheme for Prolog because of its simple stack implementation and efficient memory use. An early description of the memory management of Prolog can be found in [Bru82].

The clause

```
a(C) :- b(C), c(C).
```

can be represented by the tree in fig. 5.

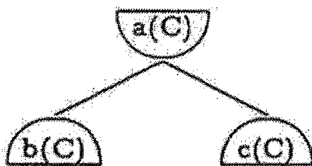



Figure 5: clause

Subtrees can be combined to a complete proof tree, also called AND-OR tree. As an example, take the following short Prolog program:

```
x(X) :- x(X).
a(C) :- b(C), c(C).
a(D) :- b(D), d(D).
b(s(0)).
c(s(0)).
d(s(0)).
```

The AND-OR tree is shown in fig. 6. The thick lines belong to the AND-tree of the last solution, the thin lines belong to the AND-tree of the first solution. The AND-tree represents the calls of the different goals of a clause. The OR-tree represents the alternative solutions.

The AND-OR-tree can be represented in linearized form by a stack (see fig. 4). Since we are only interested in one solution at a time, only an AND-tree

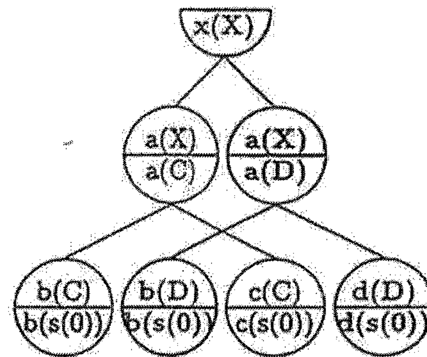



Figure 6: proof tree

is stored in the stack. The OR-tree corresponds to different contents of the stack between backtracking. Fig. 4 represents the AND-OR-tree at three different moments. The left part of the figure shows the first solution, the middle shows the stack after backtracking and the right part shows the second solution.

The cells of structures are allocated on a stack. In fig. 4 the cells for the structure `s(0)` would be allocated after the stack frame for `b(s(0))`. When the stack frame for `c(s(0))` is allocated, the stack frame for `b(s(0))` can be discarded if there are no references into the discarded stack frame and if there are no structure cells on the stack. In order to allow memory reuse the stack is divided into two parts. The environment (or local) stack holds the stack frames and the copy stack (global stack or heap) holds structure cells. The dangling reference problem can be solved if references within the environment stack are directed towards the bottom of the stack or to the heap.

In order to facilitate the removal of stack frames, there is a distinction between deterministic and indeterministic stack frames. A stack frame is deterministic if no alternative clauses are left for this procedure. An indeterministic stack frame is called choice point.

During unification variables in a stack frame may become bound. On backtracking they should be reset

to unbound. An additional stack, the trail, solves this problem. During unification the addresses of bound variables are pushed onto the trail. On backtracking these addresses are popped from the trail and the variables are reset to unbound. It is only necessary to trail the addresses of variables which are closer to the bottom of the stack than the last choice point. Testing this condition is called trail check.

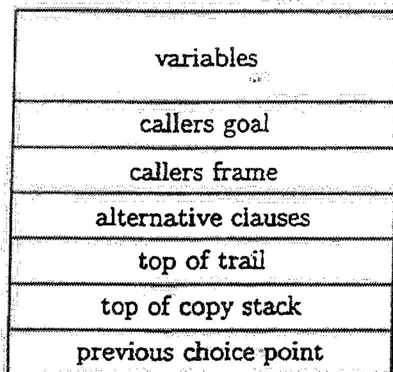



Figure 7: stack frame with choice point

Fig. 7 shows a stack frame with a choice point. A deterministic stack frame contains the cells for the variables, a pointer to the caller of this clause, comparable to the return address in a conventional stack frame, and a pointer to the stack frame of the caller. These two pointers are usually called continuation. A choice point additionally contains a pointer to the next alternative clause, a pointer to the top of trail and a pointer to the top of copy stack at the time the choice point was created, and a pointer to the previous choice point.

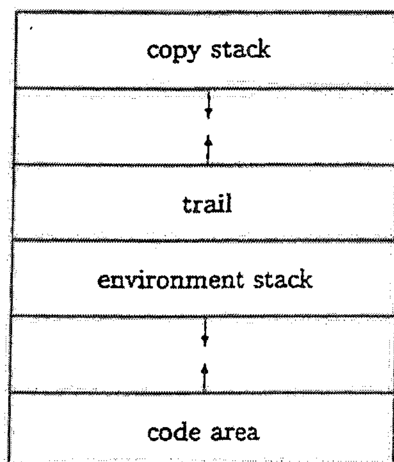



Figure 8: data areas

Fig. 8 shows the stacks and data areas in a Prolog system. The check for pointer directions is simplified if copy and environment stack grow in the same direction, and the copy stack grows towards the environment stack. The code area is needed to store the program and string representations of the atoms.

To enable fast unification, only unique identifiers of atoms are stored in variables. A hash table or search tree is constructed over these strings to enable fast searching when only the string representation is known. The same concept is applied to functors (name and arity of structures).

## 2.4 Simple Optimizations

### 2.4.1 The Representation of Terms

In the previous sections we have used a representation of structures known as structure copying. This technique was introduced by Maurice Bruynooghe [Bru82] and Christopher Mellish [Mel82]. Structure copying is now the standard implementation method because it is faster than the previously used structure sharing [BM72]. In general, structure copying also consumes less memory than structure sharing [Mel82].

Structure sharing is based on the assumption that a large part of a structure is constant and contains only few variables. A structure is here divided into the constant, part called skeleton, and a variable part, called environment. The skeleton contains the constants and the offsets into the environment, the environment contains the variables. The skeleton is stored in the code area, the environment in the global stack. A structure is represented by two pointers, one to the skeleton and one to the environment. Therefore, a variable cell has to hold a tag and two pointers. On modern machine architectures this means that a cell needs two words and spends much time in decoding skeletons. Thus only the first Prolog systems [BM73] and David Warrens first Prolog compiler [War77] used structure sharing. But in conjunction with binary Prolog (see section 3.3) structure sharing can gain in interest again.

### 2.4.2 Interpreters and Compilers

We did not yet address the problem of how to represent programs. A simple solution is to directly use the term representation of the clauses. The interpreter then has two instructions, the unification which operates on a whole goal and the head of the matching clause, and the resolution which pushes whole clauses onto the stack and does the backtracking. This simple model is called clause or goal stacking model. Using structure sharing for the goal level of the term leads to the classical interpreter model with two term pointers and two environment (frame) pointers.

Unification in general consists of assignments, conditional assignments and comparisons. So it is quite natural to break the unification up into its atomic parts. The program is analysed and instructions specialized for the argument types of the goals are generated. The resolution can be divided into stack allocation, clause indexing and calling instructions. The program is represented as a sequence of such instructions which can be either executed by an interpreter or compiled to machine code. Such an instruction set definition together with the memory model is called an abstract machine. Several abstract machines were defined, in this paper only the common Warren Abstract Machine (WAM) and the Vienna Abstract Machine (VAM) are dealt with.

#### 2.4.3 Variable Classification

In the simple execution model presented above it is assumed that during allocation of a stack frame all variable cells are initialized to unbound. Furthermore, for every variable occurring in a clause a cell is allocated.

Variables occurring only once in a clause, called void variables, can be bound only by a single instruction. The value bound to this variable will never be used. So it is not necessary to reserve space for such variables. Another case are variables which occur only within one subgoal. It is not necessary to reserve the space over different goals. Space for these temporary variables is not reserved in the stack frame but in an additional fixed area. To avoid dangling pointers, references must always point from temporary variables to the environment or copy stack.

The initialization of the stack frame and of temporary variables can be eliminated if the first occurrence and further occurrences of a variable are distinguished. The improvement comes not only from the elimination of some initializations but also from the elimination of a complex unification for the first occurrence.

#### 2.4.4 Clause Indexing

Indexing of Prolog clauses is an optimization whose aim is to reduce the number of clauses to be tried and to avoid the creation of choice points if possible. The results are better execution times and memory consumption.

The most trivial optimization done by every Prolog system is to try only the clauses of a procedure instead of all clauses of a program during the the search for a unifying clause. First argument indexing is more complicated: Only clauses which unify with the goal in the first argument are selected. For this purpose an indexing structure is built over the clauses which differentiates the clauses depending on their first arguments. This indexing structure is either a hash table or a search tree. The search tree has the advantage that it easily handles variables in the head of the clauses and allows dynamic clause insertion. Sophisticated clause indexing schemes are presented in section 5.2.

#### 2.4.5 Last-call Optimization

In section 2.3 we noticed that stack frames can be discarded after the subtree has been proved and no alternatives are left. This check is simple. The stack frame has to be the top-most frame. There can be no choice point left on the stack allocated later. A deterministic stack frame can be discarded not only after the call of the last subgoal, but also before this call. The general solution is to copy the stack frame of the called clause over the stack frame of the clause with the last call after the unification of the variables has been done (see fig. 9).

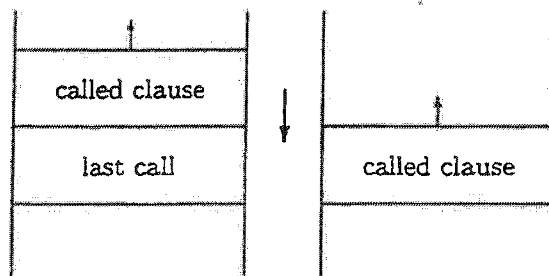



Figure 9: general last-call optimization

This frame moving is complicated by the fact that there could be references to the moved stack frame and references to unbound variables in the discarded stack frame. Therefore, the variables have to be checked and updated prior to the moving of the stack frame. Instead of updating the references, the variables can be globalized. That means that they are allocated on the global (copy) stack. The overhead of moving the stack frame can be avoided by copying the discarded stack frame to registers. The new stack frame then is directly created at the place of the discarded frame (WAM). An other solution is to create the new stack frame in registers and copy the registers to the place of the discarded frame (VAM).

Last-call optimization can be generalized for every call. A deterministic stack frame can be moved over this part of a stack frame which is not used at later calls. For that purpose the variables have to be ordered on their last occurrence. A simple stack trimming without the overhead of generalized last-call optimization can be achieved by discarding only variables which have their last occurrence before the call. Last-call optimization can reduce an infinite memory con-

144

sumption to a finite one. So it has to be implemented in every Prolog system. Specialized implementations also reduce the run time because unifications can be eliminated if variables occupy the same location.

#### 2.4.6 Garbage Collection

In Prolog unreferenced data (garbage) can be produced both in the code area and in the copy stack. But different kinds of garbage collection algorithms can be applied to these data areas. At least the copy stack needs a compacting collector which preserves the order of the cells. An algorithm which uses pointer reversal has the best space-time complexity. When the copy stack becomes compacted the trail must be updated too. Some Prolog garbage collectors collect only part of the stack due to wrong interpretations of uninitialized variables. Unused data in the code area is easily detected by the retract procedure. If the code is not moved, no updates of the environment stack are necessary.

## 3 The Warren Abstract Machine

Six years after the development of his successful compiler for the DEC-10 David Warren presented a new abstract Prolog instruction set [War83]. This New Prolog Engine has become very popular under the name Warren Abstract Machine (WAM). It has been the basis of nearly all Prolog systems developed after the year 1983. The aim of the WAM was to serve as a simple and efficient implementation model for byte code interpreters as well as machine code generating compilers. So the first implementation was a structure copying byte code emulator.

### 3.1 The Original Warren Abstract Machine

The WAM is closer to the execution model of imperative languages than all other implementation models. The main idea is the division of the unification into two parts, the copying of the arguments of the calling goal into argument registers and the unification of the argument registers with the arguments of the head of the called clause. This is very similar to the parameter passing in imperative languages like C. The first parameters are passed via registers. If the registers are exhausted, the stack can be used for additional parameters. The partitioning of the unification reduces the number of instruction pointers to one and the number of frame pointers to one, if all parameters can be kept in registers.

This parameter passing is mirrored in the instruction set. `put` instructions copy the arguments of the goal into the registers, `get` instructions unify the registers with the arguments of the head. `unify` instructions handle the unification of structure arguments. They can be executed in two modes. In write mode a new structure is created, in read mode the structure arguments are unified with the arguments of the head. *procedural* instructions manage the stack and execute procedure calls. *indexing* instructions build the indexing structure. The data areas are identical to the previously presented simple model (see fig. 10), but the choice point is quite different. The original WAM added a push down list used as a stack for the recursive unification procedure. But in a byte code emulator this push down list is hidden in the run time stack of the implementation language. In a machine code generating compiler the environment or the copy stack can be used for this purpose.

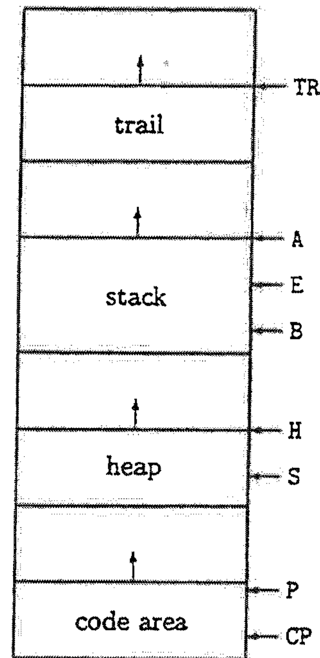



Figure 10: data areas of the WAM

Since all variables in the stack frame are copied into the argument registers before calling a procedure, last-call optimization is simplified. The stack frame of the called procedure can be created directly at the place of the stack frame of a deterministic caller. To avoid the overhead of recreating the argument registers on backtracking using put instructions, the argument registers are saved in the choice point. This permits last-call optimization also in these cases where the called procedure has alternative clauses. Furthermore, this leads to a relaxed definition of temporary variables. The head, the first subgoal and all builtin predicates between head and first subgoal count as one subgoal

for the classification of temporary variables. Unfortunately, the problem of dangling references is not solved. Therefore, there are special versions of put instructions which check if the last occurrence of a variable in the last subgoal has a reference to the discarded stack frame. Such variables are called unsafe variables and are allocated on the copy stack.

After this introduction we can present the machine registers of the WAM:

| | |
|---|---|
| P | program counter |
| CP | continuation program counter |
| E | current environment pointer |
| B | most recent choice point |
| A | top of stack (not strictly necessary) |
| TR | top of trail |
| H | top of heap |
| S | structure pointer |
| A1,A2,... | argument registers |
| X1,X2,... | temporary registers |

The continuation program counter is a register which caches the pointer to the continuation goal. It can be compared with the return address in an imperative language. Holding this value in a register speeds up the execution of the leaf procedures. The environment pointer is comparable to the frame pointer in an imperative language. The original WAM contained a HB register (heap backtrack point) which caches the top of heap corresponding to the most recent choice point. It is used to check if a variable has to be trailed. In general it is faster to take this value directly from the choice point than to update this register at every choice point creation and deallocation. The structure pointer S is used during the unification of the arguments of structures. Also named different, argument registers and temporary registers share the same pool of registers. Register allocation tries to use the registers in such an order that the number of instructions can be reduced.

The environment contains the local variables and the continuation code pointer CP' and a pointer to the previous environment E'. The choice point is shown in fig. 11. B', H', TR', CP', E' and the Ai' are copies of the values of the machine registers before the creation of the choice point. The value BP of the retry pointer is supplied by the instruction which creates the choice point and points to the code of the next alternative clause.

Fig. 12 shows the complete WAM instruction set. Vn describes either temporary or local variables. Ri designates the argument registers. C is a constant (integer or atom) in its internal representation and F is the functor of a structure which contains the name and the arity of the structure.

| | |
|---|---|
| B' | previous choice point |
| H' | top of heap |
| TR' | top of trail |
| BP | retry program pointer |
| CP' | continuation program pointer |
| E' | environment pointer |
| A1' | argument registers |
| ... | |
| An' | |

Figure 11: choice point in the WAM

## 3.2 Optimizing the basic WAM

In an interpreter the execution mode of unify instruction is hidden in the state of the interpreter. There are just two instruction decoding loops, one for the read mode and one for the write mode. In a machine code generating compiler the mode has to become explicit. The simple solution of a flag register, which is checked in every instruction, is not very efficient. The first step is to divide the unify instructions in write and read instructions. The optimal solution, which splits all paths for read and write mode, has exponential code size. Linear space is consumed if the mode flag is only tested once per structure. This scheme can be improved if write mode is propagated down a nested structure and read mode is propagated up. A more detailed description and further references can be found in [Roy94].

In the WAM it is very common that unbound variables are bound to a value shortly after their initialization. This happens e.g. if a variable has its first occurrence in the subgoal which calls a procedure with a constant argument. The variable has to be created in memory and needs to be dereferenced and trailed before being bound. Beer [Bee88] recognized that this is time consuming and additionally would require an occur check if implemented. He developed the idea of an uninitialized variable.

An uninitialized variable is defined to be an unbound variable that is unaliased, that means it is not shared with another variable. Such a variable gets a special reference tag. Creation of an uninitialized variable is simpler, it does not have to be dereferenced or trailed. Binding reduces to a single store operation. It is necessary to keep track of such variables at run time. If they remain uninitialized after the execution of the subgoal they have been created, they must be initialized to unbound.

### 3.3 Binary Prolog

The key idea of binary Prolog is the transformation of clauses to binary clauses using a continuation passing style. BinProlog, an efficient emulator for binary Prolog has been developed by Paul Tarau [Tar91][Tar92]. The implementation is based on the WAM which can be greatly simplified in that case.

In binary Prolog a clause has at most one subgoal. A clause can be transformed to a binary clause by representing the call of subgoals explicitely using continuations [App92]. For that purpose the first subgoal is given an additional argument containing the success continuation. The success continuation is the list of subgoals to be executed if the first subgoal is executed successfully. The head is given an additional argument which passes on the continuation. A fact is transformed to a clause, whose subgoal executes a meta-call of the continuation. For example, the following clauses

```
nrev([],[]).
nrev([H|T],R) :-
    nrev(T,L), append(L,[H],R).
```

are transformed into

```
nrev([],[],Cont) :- call(Cont).
nrev([H|T],R,Cont) :-
    nrev(T,L,append(L,[H],R,Cont)).
```

Compiling binary Prolog to the WAM it appears that the environment stack is superfluous since all variables are temporary. Therefore, all instruction dealing with local variables or managing the stack can be eliminated. So a small and efficient interpreter can be implemented. But this simplification has a big problem. The continuation, which contains also the variables previously contained in the stack frame, is stored on the copy stack. This means that there is no last-call optimization. So for a working BinWAM an efficient garbage collector is crucial. In some sense the BinWAM can be seen as mixture of a clause stacking model with the WAM.

## 4 The Vienna Abstract Machine

### 4.1 Introduction

The VAM has been developed at the TU Wien as an alternative to the WAM. The WAM divides the unification process into two steps. During the first step the arguments of the calling goal are copied into argument registers and during the second step the values in the argument registers are unified with the arguments of the head of the called predicate. The VAM eliminates the register interface by unifying goal and head arguments in one step. The VAM can be seen as a partial evaluation of the call. There are two variants of the VAM, the $VAM_{1P}$ and the $VAM_{2P}$.

A complete description of the $VAM_{2P}$ can be found in [KN90]. Here we give a short introduction to the $VAM_{2P}$ which helps to understand the $VAM_{1P}$ and the compilation method. The $VAM_{2P}$ (VAM with two instruction pointers) is well suited for an intermediate code interpreter implemented in C or in assembly language using direct threaded code [Bel73]. The goal instruction pointer points to the instructions of the calling goal, the head instruction pointer points to the instructions of the head of the called clause. During an inference the $VAM_{2P}$ fetches one instruction from the goal, one instruction from the head, combines them and executes the combined instruction. Because information about the calling goal and the called head is available at the same time, more optimizations than in the WAM are possible. The VAM features cheap backtracking, needs less dereferencing and trailing, has smaller stack sizes and implements a faster cut.

The $VAM_{1P}$ (VAM with one instruction pointer) uses one instruction pointer and is well suited for native code compilation. It combines instructions at compile time and supports additional optimizations like instruction elimination, resolving temporary variables during compile time, extended clause indexing, fast last-call optimization, and loop optimization.

### 4.2 The $VAM_{2P}$

Like the WAM, the $VAM_{2P}$ uses three stacks. Stack frames and choice points are allocated on the environment stack, structures and unbound variables are stored on the copy stack, and bindings of variables are marked on the trail. The intermediate code of the clauses is held in the code area. The machine registers are the goalptr and headptr (pointer to the code of the calling goal and of the called clause respectively), the goalframeptr and the headframeptr (frame pointer of the clause containing the calling goal and of the called clause respectively), the top of the environment stack, the top of the copy stack, the top of the trail, and the pointer to the last choice point.

Values are stored together with a tag in one machine word. We distinguish integers, atoms, nil, lists, structures, unbound variables and references. Unbound variables are allocated on the copy stack to avoid dangling references and the unsafe variables of the WAM. Furthermore it simplifies the check for the trailing of bindings. Structure copying is used for the representation of structures.

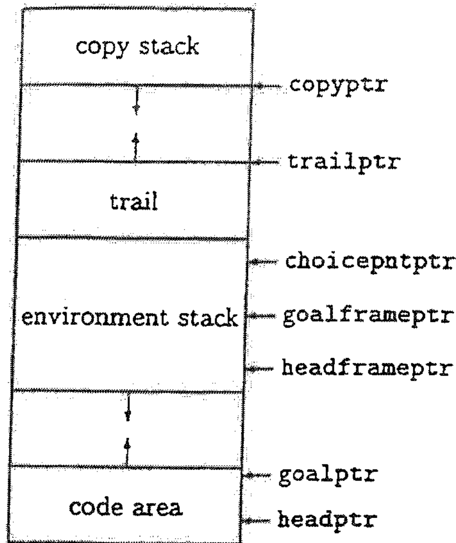



Figure 13: VAM data areas

| variables | local variables |
|---|---|
| goalptr' | continuation code pointer |
| goalframeptr' | continuation frame pointer |

Figure 15: stack frame

| trailptr' | copy of top of trail |
|---|---|
| copyptr' | copy of top of copy stack |
| headptr' | alternative clauses |
| goalptr' | restart code pointer ($VAM_{2P}$) |
| goalframeptr' | restart frame pointer |
| choicepntptr' | previous choice point |

Figure 16: choice point

Variables are classified into void, temporary and local variables. Void variables occur only once in a clause and need neither storage nor unification instructions. Different to the WAM, temporary variables occur only in the head or in one subgoal, counting a group of builtin predicates as one goal. The builtin predicates following the head are treated as belonging to the head. Temporary variables need storage only during one inference and can be held in registers. All other variables are local and are allocated on the environment stack. During an inference the variables of the head are held in registers. Prior to the call of the first subgoal the registers are stored in the stack frame. To avoid initialisation of variables we distinguish between their first occurrence and further occurrences.

The clauses are translated to the $VAM_{2P}$ abstract machine code (see fig. 14). This translation is simple due to the direct mapping between source code and $VAM_{2P}$ code. During run time a goal and a head instruction are fetched and the two instructions are combined. Unification instructions are combined with unification instructions and resolution instructions are combined with termination instructions. A different encoding is used for goal unification instructions and head unification instructions. To enable fast encoding the instruction combination is solved by adding the instruction codes and, therefore, the sum of two instruction codes must be unique.

## 4.3 The $VAM_{1P}$

The $VAM_{1P}$ has been designed for native code compilation. A complete description can be found in [KB92]. The main difference to the $VAM_{2P}$ is that instruction combination is done during compile time instead of run time. The representation of data, the stacks and stack frames (see fig. 15) are identical to the $VAM_{2P}$. The $VAM_{1P}$ has one machine register less than the $VAM_{2P}$. The two instruction pointers goalptr and headptr are replaced by one instruction pointer called codeptr. Therefore, the choice point (see fig. 16) is also smaller by one element since there is only one instruction pointer. The pointer to the alternative clauses now directly points to the code of the remaining matching clauses.

Due to instruction combination during compile time it is possible to eliminate instructions, to eliminate all temporary variables and to use an extended clause indexing, a fast last-call optimization and loop optimization. In WAM based compilers abstract interpretation is used to derive information about mode, type and reference chain length. Some of this information is locally available in the $VAM_{1P}$ due to the availability of the information of the calling goal.

All constants and functors are combined and evaluated to true or false. For a true result no code is emitted. All clauses which have an argument evaluated to false are removed from the list of alternatives. In general no code is emitted for a combination with a void variable. In a combination of a void variable with the first occurrence of a local variable the next occurrence of this variable is treated as the first occurrence.

Temporary variables are eliminated completely. The unification partner of the first occurrence of a temporary variable is unified directly with the unification partners of the further occurrences of the temporary variable. If the unification partners are constants, no code is emitted at all. Flattened code is generated for structures. The paths for unifying and copying structures is split and different code is generated for each path. This makes it possible to reference each argument of a structure as offset from the top of the copy stack or as offset from the base pointer of the struc-

ture. If a temporary variable is contained in more than one structure, combined unification or copying instructions are generated.

All necessary information for clause indexing is computed during compile time. Some alternatives are eliminated because of failing constant combinations. The remaining alternatives are indexed on the argument that contains the most constants or structures. For compatibility reasons with the $VAM_{2P}$ a balanced binary tree is used for clause selection.

The $VAM_{1P}$ implements two versions of last-call optimization. The first variant (we call it post-optimization) is identical to that of the $VAM_{2P}$. If the determinacy of a clause can be determined during run time, the registers containing the head variables are stored in the callers stack frame. Head variables which reside in the stack frame due to the lack of registers are copied from the head (callee's) stack frame to the goal (caller's) stack frame.

If the determinacy of a clause can be detected during compile time, the caller's and the callee's stack frames are equal. Now all unifications between variables with the same offset can be eliminated. If not all head variables are held in registers reading and writing variables must be done in the right order. We call this variant of last-call optimization pre-optimization.

Loop optimization is done for a determinate recursive call of the last and only subgoal. The restriction to a single subgoal is due to the use of registers for value passing and possible aliasing of variables. Unification between two structures is performed by unifying the arguments directly. The code for the unification of a variable and a structure is split into unification code and copy code.

# 5 Optimizations

## 5.1 Abstract Interpretation

Information about types, modes, trailing, reference chain length and aliasing of variables of a program can be inferred using abstract interpretation. Abstract interpretation is a technique of describing and implementing global flow analysis of programs. It was introduced by [CC77] for dataflow analysis of imperative languages. This work was the basis of much of the recent work in the field of logic programming [AH87] [Bru91] [Deb92] [Mel85] [RD92] [Tay89]. Abstract interpretation executes programs over an abstract domain. Recursion is handled by computing fixpoints. To guarantee the termination and completeness of the execution a suitable choice of the abstract domain is necessary. Completeness is achieved by iterating the interpretation until the computed information change. Termination is assured by bounding the size of the domain. The previous cited systems all are meta-interpreters written in Prolog and very slow.

A practical implementation of abstract interpretation has been done by Tan and Lin [TL92]. They modified a WAM emulator implemented in C to execute the abstract operations on the abstract domain. They used this abstract emulator to infer mode, type and alias information. They analysed a set of small benchmark programs in few milliseconds which is about 150 times faster than the previous systems.

## 5.2 Sophisticated Clause Indexing

The standard indexing method used in WAM-based Prolog systems can create two choice points. Therefore, this method has been called two-level indexing. Carlson [Car87] introduced one-level indexing by delaying the creation of a choice point as long as possible. By discriminating first on the type of the first argument and when appropriate on its principal functor, the set of potentially matching clauses is filtered out. A choice point is then needed only for non singleton sets. In the worst case the number of indexing instructions can be quadratic to the number of clauses. The $VAM_{2P}$ uses pointers instead of indexing instructions to avoid two-level indexing and to enable assert and retract [Kra88]. A similar strategy is used in [DMC89].

The use of field encoded and superimposed code words for clause indexing was proposed by Wise and Powers [WP84] and was refined by Colomb [CJ86] [Col91]. The method is based on content addressable memory (CAM). The CAM consists of an array of bit columns. Logical operations on columns and lines of the CAM can be computed in one cycle. The results of operations can be held in result columns or lines. The idea is to hold hash values for the arguments of a clause in the CAM. The encoding scheme is based on m–in–n coding which sets m bits in a word of size n to 1. Field encoding uses n/2–in–n coding and gives each argument some bits of a line. Superimposed coding uses m–in–n coding, where n is the size of a whole line and m so small that m times number of arguments is n/2. Variables are either represented by a special column or by hash values with all bits set to 1 or 0. The CAM is fast, but too special and expensive to be used in general purpose computer systems.

In [KS88] Kliger and Shapiro describe an algorithm for the compilation of an FCP(—,:,?) procedure into a control-flow decision tree that analyses the possible data states in a procedure call. This tree is translated to a header for the corresponding machine code of the predicate. At run time the generated instructions control the flow which finally reaches the jump instruction pointing to the correct clause. Redundant tests in a

process reduction attempt are eliminated and the candidate clause is found efficiently. The decision tree may need program space exponential in the number of clauses and argument positions. Consequently in [KS90] they choose decision graphs rather than decision trees to encode the possible traces of each predicate.

Hickey and Mudambi [HM89] were the first who applied decision trees as an indexing structure to Prolog. They compile a program as a whole and apply mode inference to determine which arguments are bound. The decision tree is compiled into switching instructions which can be combined with unification instructions and primitive tests. So equivalent unifications which occur in different clauses are evaluated only once. Reusing the result of such a unification requires a consistent register use. A complete indexing scheme generating algorithm is presented which takes into account effects of aliasing and gives a consistent register use. They also show that the size of the switching tree is exponential in the worst case and that finding an optimal switching tree is NP-complete. For cases where the size of the switching tree is a problem they also present a quadratic indexing algorithm. In general the size is no problem and the speedup is a factor of two.

Palmer and Naish [PN91] and Hans [Han92] also noticed the potential exponential size of decision trees. They compute the decision tree for each argument separately and store the set of applicable clauses for each argument. At run time the arguments are evaluated and the intersection of the applicable clause sets of each argument is computed. The disadvantage of this method is the high run time overhead. Furthermore the size of the clause sets is quadratic to the number of clauses, whereas decision trees are rarely exponential with respect to the number of arguments.

## 5.3 Stack Checking

Since a Prolog system has many stacks, the run time checking of stack overflow can be very time consuming. There are two methods to reduce this overhead. The more effective one uses the memory management unit of the processor to perform the stack check. A write protected page of memory is allocated between the stacks. Catching the trap of the operating system can be applied to promote a more meaningful error message to the user. A problem with this scheme occurs in combination with garbage collection. The trap can occur at a point in the program where the internal state of the system is unclear so that it is difficult to start garbage collection.

The second idea is to reduce the scattered overflow checks to one check per call. It is possible to compute at compile time the maximum number of cells allocated during a single call on the copy and the environment stack. If these stacks grow into one another (possible only if no references are on the environment stack) both stacks can be tested with a single overflow check. The maximum use of the trail during a call can not be determined at compile time.

## 5.4 Instruction Scheduling

Modern processors can issue instructions while preceding instructions are not yet finished and can issue several instructions in each cycle. It can happen that an instruction has to wait for the results of another instruction. Instruction scheduling tries to reorder instructions so that they can be executed in the shortest possible time.

The simplest instruction schedulers work on basic blocks. The most common technique is list scheduling [War90]. It is a heuristic method which yields nearly optimal results. It encompasses a class of algorithms that schedule operations one at a time from a list of operations to be scheduled, using priorization to resolve conflicts. If there is a conflict between instructions for a processor resource, this conflict is resolved in favour of the instruction which lies on the longest executing path to the end of the basic block. A problem with basic block scheduling is that in Prolog basic blocks are small due to tag checks and dereferencing. So instruction scheduling relies on global program analysis to eliminate conditional instructions and increase basic block sizes. Just as important is alias analysis. Loads and stores can be moved around freely only if they do not address the same memory location.

A technique called trace scheduling can be applied to schedule the instructions for a complete inference [Fis81]. A trace is a possible path through a section of code. In general it would be the path from the entry of a call to the exit of a call. Trace scheduling uses list scheduling, starting with the most frequent path and continuing with less frequent paths. During scheduling it can happen that an instruction has to be moved over a branch or join. In this case compensation code has to be inserted on the other path. In Prolog the less frequent path is often the branch to the backtracking code. In such cases it is often not necessary to compensate the moved instruction.

# Acknowledgement

We express our thanks to Alexander Forst, Franz Puntigam and Jian Wang for their comments on earlier drafts of this paper.

## References


[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Bee88] Joachim Beer. The occur-check problem revisited. *Journal of Logic programming*, 5(3), 1988.

[Bel73] James R. Bell. Threaded code. *CACM*, 16(6), June 1973.

[BM72] Roger S. Boyer and Jay S. Moore. The sharing of structure in theorem proving programs. In Melzer B. and Michie D., editors, *Machine Intelligence 7*. Edinburgh University Press, New York, 1972.

[BM73] Gérard Battani and Henry Meloni. Interpréteur du language PROLOG. Dea report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université de Aix-Marseille II, 1973.

[Bru82] Maurice Bruynooghe. The memory management of PROLOG implementations. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*. Academic Press, 1982.

[Bru91] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic programming*, 10(1), 1991.

[Car87] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Fourth International Conference on Logic Programming*, 1987.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symp. Priciples of Programming Languages*. ACM, 1977.

[CJ86] Robert M. Colomb and Jayasooriah. A clause indexing system for prolog based on superimposed coding. *Australian Computer Journal*, 18(1), 1986.

[Col91] Robert M. Colomb. Enhancing unification in Prolog through clause indexing. *Journal of Logic programming*, 10(1), 1991.

[Col93] Alain Colmerauer. The birth of Prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*, SIGPLAN Notices, pages 37–52. ACM, March 1993.

[Deb92] Saumya Debray. A simple code improvement scheme for Prolog. *Journal of Logic Programming*, 13(1), 1992.

[DMC89] Bart Demoen, Andre Marien, and Alain Callebaut. Indexing prolog clauses. In *North American Conference on Logic Programming*, 1989.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30, 1981.

[Han92] Werner Hans. A complete indexing scheme for WAM-based abstract machines. In *PLILP'92*, LNCS 631. Springer, 1992.

[HM89] Timothy Hickey and Shyam Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7(3), 1989.

[Jaf84] Joxan Jaffar. Efficient unification over infinite terms. *New Generation Computing*, 2, 1984.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a $VAM_{1P}$ based Prolog compiler. In *PLILP'92*, LNCS. Springer 631, 1992.

[KN90] Andreas Krall and Ulrich Neumerkel. The Vienna abstract machine. In *PLILP'90*, LNCS. Springer, 1990.

[Kra88] Andreas Krall. *Analyse und Implementierung von Prologsystemen*. PhD thesis, TU Wien, 1988.

[KS88] Shmuel Kliger and Ehud Shapiro. A decision tree compilation algorithm for FCP(—,:,?). In *Fifth International Conference and Symposium on Logic Programming*, Seattle, 1988.

[KS90] Shmuel Kliger and Ehud Shapiro. From decision trees to decision graphs. In *North American Conference on Logic Programming*, 1990.

[Mel82] Christopher S. Mellish. An alternative to structure sharing in the implementation of a Prolog interpreter. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*. Academic Press, 1982.

151


[Mel85] Christopher S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1), 1985.

[PN91] Doug Palmer and Lee Naish. NUA-Prolog: An extension to the WAM for parallel Andorra. In *Eighth International Conference on Logic Programming*, 1991.

[RD92] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1), 1992.

[Roy94] Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic programming*, 19/20, 1994.

[SH87] Peter Steenkiste and John Hennessy. Tags and type checking in LISP: Hardware and software approaches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM/IEEE, October 1987.

[Tar91] Paul Tarau. A compiler and a simplified abstract machine for the execution of binary metaprograms. In *Eighth International Conference on Logic Programming*, 1991.

[Tar92] Paul Tarau. WAM-optimizations in BinProlog: Towards a realistic continuation passing Prolog engine. Technical report, Université de Moncton, Canada, 1992.

[Tay89] Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, Lisbon, 1989.

[TL92] Jichang Tan and I-Peng Lin. Compiling dataflow analysis of logic programs. In *Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN*. ACM, 1992.

[War77] David H.D. Warren. *Applied Logic–Its Use and Implementation as a Programming Tool.* DAI Research Reports 39 & 40, University of Edingburgh, 1977.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[War90] Henry S. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1), 1990.

[WP84] Michael J. Wise and David M.W. Powers. Indexing Prolog clauses via superimposed code words and field encoded words. In *International Symposium on Logic Programming*, 1984.

| goal argument register loading instructions | |
|---|---|
| put_variable Vn,Ri | create a new variable, put reference into Vn and Ri |
| put_value Vn,Ri | move the content of Vn to Ri |
| put_unsafe_value Vn,Ri | move the content of Vn to Ri and globalize |
| put_constant C,Ri | move the constant C to Ri |
| put_nil Ri | move the constant nil to Ri |
| put_structure F,Ri | create functor F, put structure pointer into Ri |
| put_list Ri | put a list pointer into Ri |
| **head argument register unifying instructions** | |
| get_variable Vn,Ri | move the content of Ri to Vn |
| get_value Vn,Ri | unify Ri with Vn |
| get_constant C,Ri | unify Ri with the constant C |
| get_nil Ri | unify Ri with the constant nil |
| get_structure F,Ri | unify Ri with the functor F |
| get_list Ri | unify Ri with a list pointer |
| **structure argument unifying instructions** | |
| unify_variable Vn | move next structure argument to Vn |
| unify_value Vn | unify Vn with next structure argument |
| unify_constant C | unify the constant C with next structure argument |
| unify_nil Ri | unify the constant nil with next structure argument |
| unify_void N | skip next N structure arguments |
| **procedural instructions** | |
| call P,N | call procedure P, trim environment size to N |
| execute P | call procedure P (last subgoal) |
| proceed | return (last instruction of fact) |
| allocate | create an environment |
| deallocate | remove an enviroment |
| **indexing and backtracking instructions** | |
| switch_on_term V,C,L,S | four-way jump depending on the type of A1 |
| switch_on_constant N,T | hashed jump (table T with size N) on constant in A1 |
| switch_on_structure N,T | hashed jump (table T with size N) on structure in A1 |
| try_me_else L | create choice point to L, then fall through |
| retry_me_else L | change retry address to L, then fall through |
| trust_me_else_fail | remove choice point, then fall through |
| try L | create choice point, then jump to L |
| retry L | change retry address, then jump to L |
| trust L | remove choice point, then jump to L |

Figure 12: WAM instruction set

1-33

| unification instructions | |
|---|---|
| `const C` | integer or atom |
| `nil` | empty list |
| `list` | list (followed by its arguments) |
| `struct F` | structure (followed by its arguments) |
| `void` | void variable |
| `fsttmp Xn` | first occurrence of temporary variable |
| `nxttmp Xn` | subsequent occurrence of temporary variable |
| `fxtvar Vn` | first occurrence of local variable |
| `nxtvar Vn` | subsequent occurrence of local variable |
| **resolution instructions** | |
| `goal P` | subgoal (followed by arguments and call/lastcall) |
| `nogoal` | termination of a fact |
| `cut` | cut |
| `builtin I` | builtin predicate (followed by its arguments) |
| **termination instructions** | |
| `call` | termination of a goal |
| `lastcall` | termination of last goal |

Figure 14: $VAM_{2P}$ instruction set

J. LOGIC PROGRAMMING 1994:19, 20:1–679 1

# THE VIENNA ABSTRACT MACHINE

ANDREAS KRALL

▷ The goal of Prolog implementations is to achieve high overall efficiency. Many high-speed implementations sacrifice the performance of the compilation built-in predicates for expensive optimizations. The Vienna Abstract Machine (VAM) aims at both, fast compilation and fast execution. Different versions of the VAM are used for different purposes: the $VAM_{2P}$ is well suited for interpretation; the $VAM_{1P}$ has been designed for native code compilation. The $VAM_{2P}$ has been modified to the $VAM_{AI}$, an abstract machine for fast abstract interpretation. This article presents all three versions, explains their implementations and compares them with state of the art Prolog systems. ◁

## 1. INTRODUCTION

The implementation of Prolog has a long history. Early systems were implemented by the group around Colmerauer [11] in Marseille. The first system was an interpreter written in Algol by Phillip Roussel in 1972. With this experience a more efficient and usable system was developed by Gérard Battani, Henry Meloni and René Bazzoli [2]. It was a structure sharing interpreter and had essentially the same built-in predicates as modern Prolog systems. This system was reasonably efficient and convinced others of the usefulness of Prolog. Together with Fernando and Luis Pereira, David Warren developed the DEC-10 Prolog, the first Prolog compiler [34]. This compiler and the portable interpreter C-Prolog spread around the world and contributed to the success of Prolog.

*Address correspondence to* Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria, andi@complang.tuwien.ac.at.

*THE JOURNAL OF LOGIC PROGRAMMING*

© Elsevier Science Inc., 1994
655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

In 1983 David Warren presented the Warren Abstract Machine (WAM) [35] which has been the basis of nearly all later Prolog implementations. The WAM divides the unification process into two steps. In the first step the arguments of the calling goal are created in or copied into argument registers. In the second step the values in the argument registers are unified with the arguments of the head of the called predicate. This concept is as well-suited for intermediate code interpreters as for compilers.

### *1.1. The Vienna Abstract Machine*

The development of the Vienna Abstract Machine (VAM) started in 1985 as an alternative to the WAM. The aim was to eliminate the parameter passing bottleneck of the WAM. The first result of the development was an interpreter [18] which led to the $VAM_{2P}$ [24]. Partial evaluation of predicate calls led to the $VAM_{1P}$ which is well-suited for machine code compilation [21]. This compiler was enhanced by global analysis and modified to support incremental compilation [22]. Short analysis times were achieved through the $VAM_{AI}$, an abstract machine for abstract interpretation [23].

The $VAM_{2P}$ (VAM with two instruction pointers) is well-suited for an intermediate code interpreter implemented in C or in assembly language using direct threaded code [4]. The VAM eliminates the WAM register interface by performing the unification of each pair of a goal and a head argument in a single step. The goal instruction pointer refers to the instructions of the calling goal, the head instruction pointer to the instructions of the head of the called clause. For each argument an inference step of the $VAM_{2P}$ fetches one instruction from the goal and one instruction from the head, combines them and executes the combined instruction. Because information about both, the calling goal and the called head, is available simultaneously, more optimizations than in the WAM are possible. The VAM features cheap backtracking and cut, needs less dereferencing and trailing and has smaller stack sizes.

The $VAM_{1P}$ (VAM with one instruction pointer) uses only one instruction pointer and is well-suited for native code compilation. It combines goal and head instructions at compile time and supports additional optimizations like instruction elimination, resolving temporary variables during compile time, extended clause indexing, fast last-call optimization, and loop optimization.

A common solution for solving data flow analysis problems is abstract interpretation. To collect information about a program abstract interpretation executes the program over an abstract domain. Current abstract interpretation systems for Prolog were too slow for use in an optimizing Prolog compiler. So the $VAM_{AI}$ (VAM for abstract interpretation) has been designed. It is an order of magnitude faster than older abstract interpretation systems [23].

The article is structured as follows. Section 2 gives the details of the $VAM_{2P}$. Section 3 explains the principles of the $VAM_{1P}$. Section 4 describes the $VAM_{AI}$, a modified version of the $VAM_{2P}$ developed for abstract interpretation. Section 5 raises some implementation issues. Section 6 evaluates the performance of the VAM.

## 2. THE $VAM_{2P}$

The $VAM_{2P}$ execution model is very similar to the execution model of the classical Prolog interpreter described by Bruynooghe [6]. The main difference is that the unification has been broken up into a larger number of atomic parts, which leads to the definition of an instruction set and gives room for additional optimizations.

### *2.1. The $VAM_{2P}$ memory model*

The $VAM_{2P}$ uses three stacks and one heap (see fig. 2.1): stack frames and choice points are allocated on the environment stack; structures and unbound variables are stored on the copy stack (also called global stack or heap in the VAM); bindings of variables are marked on the trail; the intermediate code of clauses is held in the code area (organized as heap). Machine registers (see fig. 2.2) are the `goalptr` and `headptr` (pointer to the code of the calling goal and the called clause, respectively), the `goalframeptr` and the `headframeptr` (pointer to the stack frame of the clause containing the calling goal and the called clause, respectively), the top of the environment stack (`stackptr`), the top of the copy stack (`copyptr`), the top of the trail (`trailptr`), and the pointer to the last choice point (`choicepntptr`).

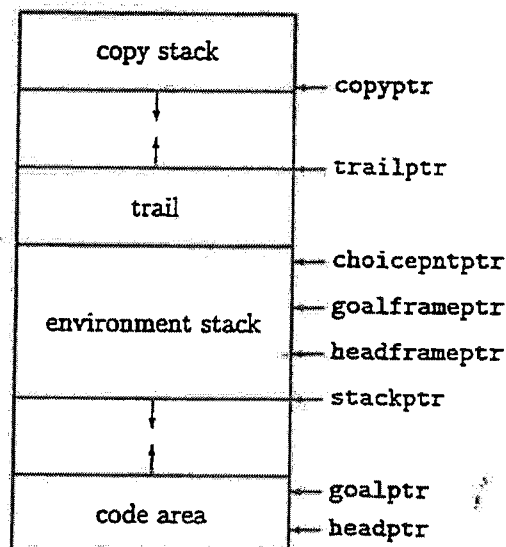



**FIGURE 2.1.** $VAM_{1P}$ data areas

Values are stored together with a tag in one machine word. The VAM distinguishes between integers, atoms, nil, lists, structures, unbound variables and references. Atoms contain a pointer to a character string. The technique of structure copying as described by Mellish [27] is used for the representation of structures. Lists and structures are represented as tagged pointers to a group of machine words. A list element uses two machine words, one for each argument. Structures use one

| register | usage |
|---|---|
| stackptr | top of environment stack |
| copyptr | top of copy stack |
| trailptr | top of trail |
| goalptr | pointer to instructions of calling goal |
| headptr | pointer to instructions of called clause |
| goalframeptr | frame pointer of calling goal |
| headframeptr | frame pointer of called clause |
| choicepntptr | previous choice point |

FIGURE 2.2. machine registers

machine word for the functor and one for each argument. The functor is a pointer to a location containing the arity and the atom. Long integers and floating point values are special structures. Unbound variables are represented as self references and are allocated on the copy stack in order to avoid dangling references and the unsafe variables of the WAM. Additionally the test for trailing of variables is simplified.

Variables are classified into void, temporary and local variables. Void variables occur only once in a clause and need neither storage nor unification instructions. Different to the WAM which treats the first subgoal as belonging to the head, temporary variables in the VAM occur only in the head or in one subgoal. A group of built-in predicates is counted as a single goal. The built-in predicates following the head are treated as if they belong to the head. Temporary variables need storage only during one inference and can be held in registers. All other variables are local and are allocated on the environment stack. During an inference the variables of the head are held in registers. These registers are stored in the stack frame prior to the call of the first subgoal. To avoid the initialization of variables, the first and further occurrences are distinguished.

## 2.2. *The $VAM_{2P}$ instruction set*

Prolog source code is translated to the $VAM_{2P}$ abstract machine code (see fig. 2.3). This translation is simple due to the direct mapping between source code and $VAM_{2P}$ code. The head arguments of a clause are translated to unification instructions. A fact is terminated by the instruction `nogoal`. Each subgoal is translated to the goal instruction, unification instructions for each argument and is terminated by the `call` instructions or the `lastcall` instruction if it is the last subgoal. The first subgoal of a clause has an additional operand, that specifies the number of local variables in the head. This number is needed for last-call optimization. Example 2.1 shows the $VAM_{2P}$ code for the append procedure.

*Example 2.1.*

```
append([],              nil
```

| unification instructions | |
|---|---|
| `const C` | integer or atom |
| `nil` | empty list |
| `list` | list (followed by its arguments) |
| `struct F` | structure (followed by its arguments) |
| `void` | void variable |
| `fsttmp Xn` | first occurrence of temporary variable |
| `nxttmp Xn` | subsequent occurrence of temporary variable |
| `fxtvar Vn` | first occurrence of local variable |
| `nxtvar Vn` | subsequent occurrence of local variable |
| **resolution instructions** | |
| `goal N, P` | first subgoal (followed by arguments and call/lastcall) |
| `goal P` | further subgoals (followed by arguments and call/lastcall) |
| `nogoal` | termination of a fact |
| `cut` | cut |
| `builtin I` | built-in predicate (followed by its arguments) |
| **termination instructions** | |
| `call` | termination of a goal |
| `lastcall` | termination of last goal |

**FIGURE 2.3.** $VAM_{2P}$ **instruction set**

```
       L,              fsttmp L
       L               nxttmp L
       ).              nogoal

append([               list
       H|              fsttmp H
       L1],            fstvar L1
       L2,             fstvar L2
       [               list
       H|              nxttmp H
       L3]) :-         fstvar L3
append(                goal 3,append
       L1,             nxtvar L1
       L2,             nxtvar L2
       L3              nxtvar L3
       ).              lastcall
```

The translation of terms into intermediate code is done in two passes using three steps (see fig. 2.4). The first pass scans the terms for variables and collects information about the variables in the var table. The second pass again scans the terms and generates the $VAM_{2P}$ instructions. Between these two passes the variable classes and offsets are determined.

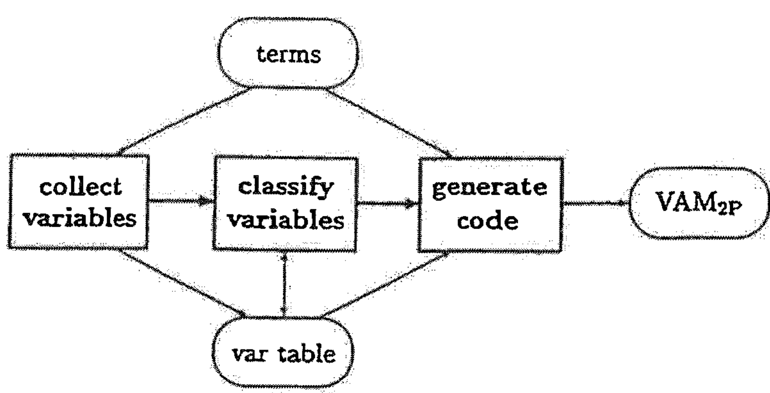



**FIGURE 2.4.** translator structure

## 2.3. *The $VAM_{2P}$ execution model*

A stack frame (see fig. 2.5) is allocated on the environment stack for each called procedure. For each non-deterministic procedure a choice point (see fig. 2.6) is pushed on the environment stack, too. The stack frame contains the variables and the continuation. The continuation is saved in the stack frame prior to the call of the first subgoal. The continuation contains the frame pointer of the calling clause (`goalframeptr'`) and the address of the instruction following the calling goal (`goalptr'`). If the calling goal was the last subgoal, the continuation of the caller is copied to the new stack frame.

| | |
|---|---|
| `goalptr'` | continuation code pointer |
| `goalframeptr'` | continuation frame pointer |
| $variable_0$ ... $variable_n$ | local variables |

**FIGURE 2.5.** stack frame

| | |
|---|---|
| `trailptr'` | copy of top of trail |
| `copyptr'` | copy of top of copy stack |
| `headptr'` | alternative clauses |
| `goalptr'` | restart code pointer |
| `goalframeptr'` | restart frame pointer |
| `choicepntptr'` | previous choice point |

**FIGURE 2.6.** choice point

18

During unification of a goal with the head of a called clause the next goal and head instructions are fetched, the two instructions are combined, and the combined instruction is executed. Goal unification instructions are combined with head unification instructions and resolution instructions with termination instructions. To enable fast decoding, goal and head unification instructions are encoded differently and the instruction combination is performed by adding the instruction codes. Therefore, the sum of each two instruction codes must be unique. The C statement

```
switch(*headptr++ + *goalptr++)
```

implements this instruction fetch and decoding. An assembly language implementation can use direct threaded code for faster execution [4]. Direct threaded code uses the address of the interpretation routine of an instruction as intermediate code. Portability of an assembly language implementation is achieved by macro expansion of a low level virtual machine code into assembly language.

As in the WAM, unification of structures is solved by executing the interpreter in read mode or write mode. In these modes, instructions are fetched using only one of the two instruction pointers. The proper interpreter for the mode is called recursively for each recursive structure. Unification of void variables with structures leads to skipping the argument of the structure. Thus the $VAM_{2P}$ interpreter is executed either in *combine*, *read (unify)*, *write (create)* or *skip* mode. Example 2.2 shows a code fragment of the interpreter with the parts for the four different modes (there exist three additional parts for skipping, writing and reading goal structures). A one page interpreter for ground Prolog is contained in [24].

*Example 2.2.*

```
combine:     switch(*headptr++ + *goalptr++) {
                 case (h_const+g_const):  ... goto combine;
                 case (h_list+g_void):    ... i = 2; goto head_skip;
                 case (h_list+g_fstvar):  ... i = 2; goto head_write;
                 case (h_struct+g_nxtvar):...
                     i = arity(headptr); goto head_read;
                 ...
             }
             goto combine;

head_skip:   while (--i >= 0)
              switch (*headptr++) {
              case h_const: ...
              ...
              }
             goto combine;

head_write: ...

head_read:  ...
```

### 2.4. Built-in predicate interface

The simple built-in predicate interface is identical to that of the WAM. The instructions are scanned and the corresponding values are put in the argument array for the call of the built-in predicate. This model simplifies the meta-call of built-in predicates but is not very efficient. To avoid this bottleneck, the arithmetic and some type checking built-in predicates scan the instructions on their own.

### 2.5. Meta-call

For a meta-call a special `metacall Vn` instruction is generated instead of the `goal` instruction. The operand `Vn` contains the offset of the meta-call variable in the stack frame. A meta-call is terminated by a `call` or `lastcall` instruction. At run time the variable contains an atom or a structure. The functor of a structure refers to code of the corresponding procedure. The `headptr` is set to the code of the first clause and the interpreter continues executing instructions in read mode. For backtracking of meta-calls it is necessary that the instruction combination of `metacall` with head unification and control instructions is unique.

### 2.6. Last-call optimization

The WAM implements last-call optimization by copying the variables of the caller's stack frame into argument registers. Then, the new stack frame is allocated in the place of the old one. The VAM first allocates the new stack frame and performs the unification between the caller and the callee. Afterwards, if the call of a procedure is deterministic, the stack frame of the called procedure is copied over the stack frame of the calling goal before the first subgoal is called. For this purpose the optional argument `N` of the first `goal` instruction contains the number of variables to be copied. In an assembly language implementation some head variables can be held in registers. Prior to the call of the first subgoal these registers can be stored in the place of the caller's stack frame. A more efficient version can be implemented with the $VAM_{1P}$ (see chapter 3.1).

### 2.7. Clause indexing

In contrast to the WAM the $VAM_{2P}$ does not translate indexing information into instructions but stores this information in indexing data structures. Since the $VAM_{2P}$ unifies the goal arguments from left to right, only first argument indexing can be supported. The current implementation uses a balanced binary tree for clause selection. The index values are either integers, atoms or the functors of structures. Each clause gets an additional header (see fig. 2.7) which contains a pointer to the clauses with smaller indices, a pointer to the clauses with larger indices, a pointer to

the clauses with the same index and a pointer for linear connection of the clauses. This example shows a procedure with six clauses with an integer constant or a variable as first argument.

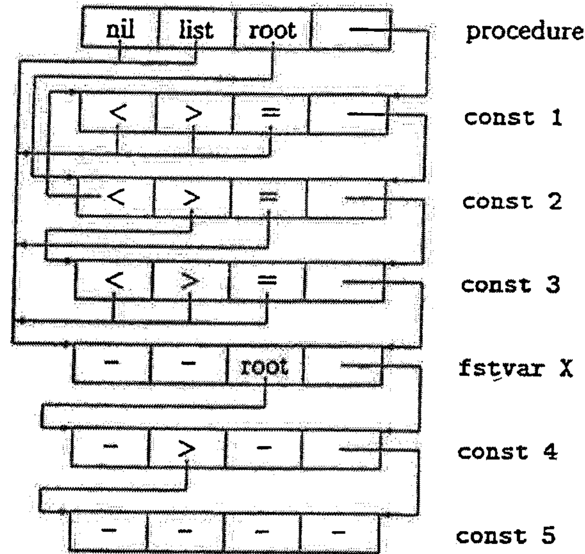



FIGURE 2.7. clause indexing

For each procedure there exists a header which contains a pointer to the clauses with nil as first argument, a pointer to the clauses with a list as first argument, a pointer to the root of the binary search tree and a pointer to the first clause. If there exists a clause with a variable as first argument, the leaf clauses of the binary search tree point to this clause and the header contains pointers to the nil clauses, the list clauses, the root of the next search tree and the linear connection.

To simplify the implementation of the indexing part of the interpreter a special case of the goal instruction is used. The instruction xgoal [N,]P,Vn is generated if the first argument of a subgoal is a variable. The operand Vn contains the offset of this variable in the stack frame. The choice point contains a flag which indicates if indexing was used for the call of this procedure.

The previously presented indexing scheme does not belong to the definition of the $VAM_{2P}$, but is an implementation model well-suited for this abstract machine. The balanced binary search tree could be replaced by a hash table. Since the number of clauses in a procedure is in general very small, the search tree is faster and better suited for incremental compilation. An indexing scheme as used by Carlson [8] or Demoen [14] for the WAM would be suitable, too.

### 2.8. Design alternatives

An important design decision is the allocation of unbound variables on the copy stack. This is a prerequisite for a fast last-call optimization and simplifies the check for the trailing of variables. An alternative solution would be to prohibit temporary variables in a subgoal (which eliminates references to the callee's stack frame) and use unsafe variables as the WAM does. The stack usage of both versions is very similar (the VAM creates only 36% of the unbound variables of the WAM [19]). Therefore, the method to store unbound variables on the copy stack is simpler and faster. Updating the references in the moved stack frame as proposed by Bruynooghe in [6] is too expensive.

It is difficult to decide if the choice point should be allocated before or after the variables in the stack frame. If it is allocated before the variables, fast shallow backtracking can be implemented by updating only the pointer to the alternative clauses. If the choice point is allocated after the variables, the space consumed by the choice point can be reclaimed easily when executing a cut. Since cuts occur more frequently than choice point updates [19], the current implementation allocates the choice point after the variables and goes for space instead of speed. Furthermore, the cut does stack trimming and removes the variables in the stack frame which are not used in later subgoals.

The size of the choice point could be reduced further by eliminating the restart code pointer and restart frame pointer by using the continuation instead. This would require an additional argument to the `goal` instruction which contains an offset to the next goal and a `nogoal` instruction after each lastcall. The disadvantage of this solution is that the continuations must be dereferenced if no last-call optimization is applicable.

## 3. THE $VAM_{1P}$

The $VAM_{1P}$ has been designed for native code compilation. A complete description can be found in [21]. The main difference to the $VAM_{2P}$ is that instructions are combined at compile time instead of run time. The $VAM_{1P}$ generates specialized code for each call of a procedure. Every subgoal of a clause is compiled to code which unifies the subgoal with each clause head of the called procedure. Since heads and goals are combined, only code for clause bodies is generated. For example

```
a([]).
a([_|L]) :- a(L).

:- a([3]).
```

is translated into $VAM_{1P}$ instructions represented as unifications in Prolog pseudocode. The variables got a suffix to identify the stack frame the variable belongs to.

```
a2:  :- (a(L_goal)=a([])) ; (a(L_goal)=a([_|L_head]), goto(a2).

     :- (a([3])=a([])) ; (a([3])=a([_|L_head]), goto(a2).
```

The representation of data, the stacks and the stack frames (see fig. 2.5) are identical to the $VAM_{2P}$. The two instruction pointers `goalptr` and `headptr` are replaced by one instruction pointer called `codeptr`. The choice point (see fig. 3.1) is smaller by one element. The pointer to the alternative clauses points directly to the code of the remaining matching clauses.

| | |
|---|---|
| `trailptr'` | copy of top of trail |
| `copyptr'` | copy of top of copy stack |
| `headptr'` | alternative clauses |
| `goalframeptr'` | restart frame pointer |
| `choicepntptr'` | previous choice point |

**FIGURE 3.1.** choice point

## *3.1. Basic optimizations*

Due to instruction combination at compile time it is possible to eliminate unnecessary instructions and all temporary variables, and to use an extended clause indexing scheme, a fast last-call optimization and loop optimization. In WAM based compilers, abstract interpretation is used in deriving information about mode, type and reference chain length. Some of this information is locally available in the $VAM_{1P}$ due to the availability of the information of the calling goal.

All constants and functors are combined and evaluated to true or false at compile time. No code is emitted for a true result. Clauses which contain an argument evaluated to false are removed from the list of alternatives. In general, no code is emitted for a combination with a void variable. In a combination of a void variable with the first occurrence of a local variable, the next occurrence of this variable is treated as the first occurrence.

Temporary variables are eliminated completely. The unification partner of the first occurrence of a temporary variable is unified directly with the unification partners of the further occurrences of the temporary variable. If the unification partners are constants, no code is emitted at all. Flattened code is generated for structures. The paths for unifying and creating structures is split and different code is generated for each path. This makes it possible to refer to each argument of a structure through an offset from the top of the copy stack or from the base pointer of the structure. If a temporary variable is contained in more than one structure, combined unification or copying instructions are generated. The two stream method as described e.g. by Meier [25] would be slower than the generation of flattened code, but would reduce the code size.

All necessary information for clause indexing is computed at compile time. Some alternatives may be eliminated because of failing constant combinations. The remaining alternatives are indexed on the argument that contains the most constants or structures. For compatibility reasons with the $VAM_{2P}$ a balanced binary tree

is used for clause selection. A version of the $VAM_{1P}$ [20] which uses a complete indexing as described by Hickey and Mudambi [17] has been implemented too.

The $VAM_{1P}$ implements two versions of last-call optimization. The first variant (called post-optimization) is identical to that of the $VAM_{2P}$. If a goal is deterministic at run time, the registers containing the head variables are stored in the caller's stack frame. Head variables which reside in the stack frame due to the lack of registers are copied from the head (callee's) stack frame to the goal (caller's) stack frame.

If the determinism of a clause can be detected at compile time, the space used by the caller's stack frame is used immediately by the callee. Therefore, all unifications between variables with the same offset can be eliminated. If not all head variables are held in registers, they have to be read and written in the right order. This optimization, called pre-optimization, can be seen as a generalization of recursion replacement by iteration to every last-call [26].

Loop optimization is done for a determinate recursive call of the last and only subgoal. The restriction to a single subgoal is due to the use of registers for value passing and possible aliasing of variables. Unification of two structures is performed by unifying the arguments directly. The code for the unification of a variable and a structure is split into unification code and copy code.

### 3.2. The $VAM_{1P}$ instructions set

The instruction set for the $VAM_{1P}$ is divided into general unification instructions, structure unification instructions, structure creation instructions, control instructions, indexing instructions and optimization instructions. In the following descriptions variables used as arguments are either registers or frame pointers with offset.

The unification instructions (see fig. 3.2) handle the unification of an argument of the calling goal and the related head argument of the called clause. If a structure is combined with the first occurrence of a variable (`fstvar_struct`) structure creation is started. The operand `Size` gives the size of the structure including all substructures. The operand `Reg` of the `nxtvar_struct` instruction is the register which holds the base pointer of the structure. The structure unification instructions follow immediately, the operand `CAddr` is the label of the structure creation instructions. Fig. 3.3 gives an example for the code generation of structures.

Structure unification (see fig. 3.4) and structure creation instructions (see fig. 3.5) are equivalent to the unify instructions of the WAM in read and write mode. Due to the splitting of the paths for unification and creation, the variable's offset from the top of the copy stack or the start of the structure is known at compile time.

The control instructions (see fig. 3.6) implement stack handling and the transfer of control: `alloc_stackframe` allocates the space for variables in a stack frame; `adjust` performs stack trimming; `store_cont_goto` saves the continuation in the stack frame and jumps to the procedure `ProcAddr`. For the last subgoal the instruction `copy_cont_goto` is used instead. `read_cont_goto` reads the continuation and executes the next subgoal. `store_var` saves the head variables residing in registers

186

| general unification |
|---|
| fstvar_const Var,Const |
| nxtvar_const Var,Const |
| fstvar_nil Var |
| nxtvar_nil Var |
| fstvar_list Var,Size |
| nxtvar_list Var,Reg,Size,CAddr |
| fstvar_struct Var,Functor,Size |
| nxtvar_struct Var,Functor,Reg,Size,CAddr |
| fstvar_fstvar Var,Var |
| nxtvar_fstvar Var,Var |
| nxtvar_nxtvar Var,Var |
| fstvar_void Var |

**FIGURE 3.2.** VAM$_{1P}$ general unification instructions

```
          nxtvar_struct  X,s/2,r0,5,label_2
          unify_const    1,1(r0)
          unify_struct   t/1,r1,2,label_3
          unify_const    2,1(r1)
label_1:  ...
label_2:  create_functor s/2,-5(copyptr)
          create_const   1,-4(copyptr)
          create_struct  -3(copyptr),-2(copyptr)
label_3:  create_functor t/1,-2(copyptr)
          create_const   2,-1(copyptr)
          goto           label_1
```

**FIGURE 3.3.** flattened structure unification code for X=s(1,t(2))

in the stackframe. If not all head variables can be held in registers, the instruction copy_var copies the variables from the callee's stack frame to the caller's stack frame during last-call optimization. push_choice_point creates a choice point with AltAddr used as the pointer to the code of the remaining clauses. cut deletes the choice points and performs stack trimming.

Depending on the type of the value contained in the variable Var, the indexing instruction index_var (see fig. 3.7) branches to one of the labels for lists, nil, constants and structures. The compare instruction cmp_const implements a binary search tree.

The optimization instructions (see fig. 3.8) support the unification of structures optimized by temporary variable elimination or loop optimization. create_undef initializes a cell on the copy stack to unbound. unify_create copies a value from the source structure to the destination structure. unify_unify implements

| structure unification |
| --- |
| unify_const Const,Offset |
| unify_nil Offset |
| unify_list Reg,Size,Offset,CAddr |
| unify_struct Functor,Reg,Size,Offset,CAddr |
| unify_fstvar Var,Offset |
| unify_nxtvar Var,Offset |

FIGURE 3.4. $VAM_{1P}$ structure unification instructions

| structure creation |
| --- |
| create_const Const,Offset |
| create_nil Offset |
| create_list Offset,Offset |
| create_struct Offset,Offset |
| create_functor FunctorOffset |
| create_fstvar Var,Offset |
| create_nxtvar Var,Offset |

FIGURE 3.5. $VAM_{1P}$ structure creation instructions

full unification between two arguments of structures. `create_ref` lets one argument of a structure reference the other. If the cell at `SAddr` contains a structure, `extract_struct` stores the address of the start of the structure in register `Reg`. Otherwise it continues execution at `ContAddr`. `construct_struct` creates a structure. `construct_extract_struct` is the combination of these two instructions.

### *3.3. Code size*

The size of the generated code can become quite large since for each call of a procedure specialized code is generated especially if there exist many calls of a procedure consisting of many clauses. But since many of these calls have the same calling pattern, the calls can share the same generated code. If this is not sufficient, a dummy call must be introduced between the call and the procedure.

## 4. THE $VAM_{AI}$

During the execution of a Prolog program a great amount of time is spent in useless type tests and dereferencing. This code can be eliminated if more information about variables is available to the compiler. Information about types, modes, trailing, reference chain lengths and aliasing of the variables of a program can be inferred using abstract interpretation.

| control |
| --- |
| alloc_stackframe VarCount |
| adjust VarCount |
| store_cont_goto ContAddr,ProcAddr |
| read_cont_goto |
| copy_cont_goto ProcAddr |
| goto Addr |
| store_var Reg,Offset |
| copy_var Offset |
| push_choice_point AltAddr |
| cut VarCount |
| call_bip BipId |
| inline_bip BipId |

**FIGURE 3.6.** VAM$_{1P}$ control instructions

| indexing |
| --- |
| index_var Var,LAddr,NAddr,CAddr,SAddr |
| cmp_const Const,LessAddr,GreaterAddr |

FIGURE 3.7. VAM$_{1P}$ indexing instructions

## *4.1. Abstract Interpretation*

Abstract interpretation is a technique for global data flow analysis of programs. It was introduced by the Cousots [12] for data flow analysis of imperative languages. This work was the basis of much of the recent work in the field of declarative and logic programming [1] [7] [10] [13] [16] [28] [30] [32]. Abstract interpretation executes programs over an abstract domain. Recursion is handled by computing fixpoints. To guarantee the termination and completeness of the execution a suitable choice of the abstract domain is necessary. Completeness is achieved by iterating the interpretation until the computed information reaches a fixpoint. Termination is assured by limiting the size of the domain. Most of the previously cited systems are meta-interpreters written in Prolog which are very slow.

A practical implementation of abstract interpretation has been done by Tan and Lin [31]. They modified a WAM emulator implemented in C to execute the abstract operations on the abstract domain. They used this abstract emulator to infer mode, type and alias information, and analysed a set of small benchmark programs in a few milliseconds. This is about 150 times faster than the previous systems.

| optimization |
| --- |
| create_undef Offset |
| unify_create Offset,Offset |
| unify_unify Offset,Offset |
| create_ref Offset,Offset |
| extract_list Reg,Laddr,ContAddr |
| construct_list Offset |
| construct_extract_list Reg,LAddr,UAddr |
| extract_struct Functor,Reg,Size,Saddr,ContAddr |
| construct_struct Functor,Size,Offset |
| construct_extract_struct Functor,Reg,Size,SAddr,UAddr |

**FIGURE 3.8.** $\text{VAM}_{1P}$ optimization instructions

## *4.2. A basic execution model of the $VAM_{AI}$*

The $\text{VAM}_{AI}$, an abstract machine for abstract interpretation, has been designed following the way of Tan and Lin. It has been developed on the basis of the $\text{VAM}_{2P}$ and benefits from the fast decoding mechanism of this machine. The inferred data flow information is stored directly in the intermediate code of the $\text{VAM}_{AI}$. The VAM was chosen as the basis of an abstract machine for abstract interpretation because it is much better suited than the WAM: The parameter passing of the WAM via registers and storing registers in backtrack points slow down the interpretation. Furthermore, in the WAM some instructions are eliminated so that the relation between argument registers and variables is sometimes difficult to determine. The translation to a $\text{VAM}_{2P}$-like intermediate code is much simpler and faster than WAM code generation. A $\text{VAM}_{2P}$-like interpreter enables the modeling of low level features of the VAM. Furthermore, the $\text{VAM}_{2P}$ intermediate code is needed for the generation of the $\text{VAM}_{1P}$ instructions.

A top-down approach is used for the analysis of the desired information. Different (static) calls to the same clause are handled separately to get more exact types. This is achieved by duplicating the clauses for each call of a procedure. So for each call of a goal there exists an own copy of the intermediate code of the called procedure. To save code size, only the heads of the clauses are copied. The bodies are shared. This duplication of the code gives a more precise analysis for the use in the $\text{VAM}_{1P}$ which generates specialized code for each call and simplifies many parts of the $\text{VAM}_{AI}$.

Recursive calls of a clause are computed until a fixpoint for the gathered information is reached. If there already exists information about a call and the new gathered information is more special than the previously derived one, i.e. the union of the old type and the new type is equal to the old type, a fixpoint has been reached and the interpretation of this call to the clause is terminated.

Abstract interpretation with the $\text{VAM}_{AI}$ is demonstrated by a short example. Fig. 4.1 shows a simple Prolog program part and a simplified view of its code duplication for the representation in the $\text{VAM}_{AI}$ intermediate code.

Prolog program:

$A_1$ :– $B^1$
$B_1$ :– $C^1$
$B_2$ :– $B^2, C^2$
$C_1$ :– *true*

Code representation:

$A_1^1$ :– $B^1$
$B_1^1$ :– $C^1$
$B_2^1$ :– $B^2, C^2$
$B_1^2$ :– $C^1$
$B_2^2$ :– $B^2, C^2$
$C_1^1$ :– *true*
$C_1^2$ :– *true*

**FIGURE 4.1.** Prolog program part and its representation in $VAM_{AI}$

The procedure $B$ has two clauses, the alternatives $B_1$ and $B_2$. The code for the procedures $B$ and $C$ is duplicated because both procedures are called twice in this program. Abstract interpretation starts at the beginning of the program with the clause $A_1^1$. The information of the variables in the subgoal $B^1$ are determined by the inferable data flow information from the two clauses $B_1^1$ and $B_2^1$. After the information for both clauses has been computed, abstract interpretation is finished because there is no further subgoal for the first clause $A_1$.

In the conservative scheme it has to be supposed that both $B_1^1$ and $B_2^1$ could be reached during program execution. Therefore, the union of the derived data flow information sets for the alternative clauses of procedure $B$ has to be formed. For $B_1^1$ only information from $C_1^1$ has to be derived because it is the only subgoal for $B_1^1$. For $B_2^1$ there exists a recursive call of $B$, named $B^2$. Recursion in abstract interpretation is handled by computing a fixpoint, i.e. the recursive call is interpreted as long as the derived data information changes. After the fixpoint has been reached, computation stops for the recursive call. The data flow information for the recursion is assigned to the clauses $B_1^2$ and $B_2^2$. After all inferable information has been computed for a clause, it is stored directly into the intermediate code. The entry pattern and success patterns are stored in the head variables information fields, the variables of a subgoal contain the the success patterns of the calls of subgoals at the left to the current subgoal. The same intermediate code is used efficiently in the next pass of the compiler that generates code.

## *4.3. The abstract domain*

The goal of the $VAM_{AI}$ is to gather information about mode, type, reference chain length and aliasing of variables. Reference chain lengths of 0, 1 and greater 1 are distinguished. The type of a variable is represented by a set comprised of following simple types:

| | |
|---|---|
| free | is an unbound variable and contains a reference to all aliased variables |
| list | is a non empty list (it contains the types of its arguments) |
| struct | is a term |
| nil | represents the empty list |
| atom | is the set of all atoms |
| integer | is the set of all integer numbers |

Possible infinite nesting of compound terms makes the handling of the types *list* and *struct* difficult. To gather useful information about recursive data structures a recursive list type is introduced which contains also the information about the termination type.

To represent the alias information, variables are collected in alias sets. Variables which could possibly be aliased are in the same set. The alias sets are represented as double linked sorted lists. In the intermediate code each set is identified by a unique number. Variables which are always aliased, can be represented by references like in ordinary Prolog interpreters. The intersection of this sets has to be stored in the intermediate code.

Efficient interpretation is achieved by using fixed-sized variable cells, which enables static stack frame size determination and the saving of the domains in intermediate code fields. The set of the domain values is represented as a bit field. Set operations like union or difference can be implemented using logical operations. The computation of the least upper bound of two domains is implemented by a *bitwise or* operation, the abstract unification by a *bitwise and*.

## *4.4. The $VAM_{AI}$ instruction set*

The representation of the arguments of a Prolog term is the same as that in the $VAM_{2P}$ (see fig. 4.2) with the following exceptions:

- Local variables have four additional information fields in their intermediate code: the actual domain of the variable, the reference chain length and two fields for alias information. These information fields replace the extension table of conventional abstract interpretation algorithms. Local variables of the head have split information fields because they store the information at both the entry of the clause and after a successful computation of this clause. This information is used for the handling of recursive calls.

- The argument of a temporary variable contains an offset which references this variable in a global table. The global table contains entries for the domain and reference chain length information or a pointer to a variable.

- The intermediate code `lastcall` has been removed because last-call optimization makes no sense in abstract interpretation. Instead, the intermediate code nogoal indicates the end of a clause. When this instruction is executed, the computation continues with the next alternative clause (artificial fail).

- The intermediate code goal got an additional argument: a pointer to the end of this goal. This eliminates the distinction between the continuation and the restart code pointer (see fig. 2.6).
- The instruction `const` has been split into `integer` and `atom`.

| unification instructions | |
|---|---|
| `int I` | integer |
| `atom A` | atom |
| `nil` | empty list |
| `list` | list (followed by its two arguments) |
| `struct F` | structure (functor)(followed by its arguments) |
| `void` | void variable |
| `fsttmp Xn` | first occurrence of temporary var (offset) |
| `nxttmp Xn` | further occurrence of temporary var (offset) |
| `fstvar Vn,D,R,Ai,Ac` | first occurrence of local var (offset, domain, ref. chain length, is aliased, can be aliased) |
| `nxtvar Vn,D,R,Ai,Ac` | further occurrence of local var (offset, domain, ref. chain length, is aliased, can be aliased) |
| **resolution instructions** | |
| `goal P,O` | subgoal (procedure pointer, end of goal) |
| `nogoal` | termination of a clause |
| `cut` | cut |
| `builtin I` | built-in predicate (built-in number) |
| **termination instructions** | |
| `call` | termination of a goal |

**FIGURE 4.2.** $VAM_{AI}$ instruction set

## *4.5. The $VAM_{AI}$ execution model*

Another significant difference to the $VAM_{2P}$ concerns the data areas: While the $VAM_{2P}$ needs three stacks, in $VAM_{AI}$ a modified environment stack and a trail are sufficient. Fig. 4.3 shows a stack frame for the environment stack of the $VAM_{AI}$. Note that every stack frame is a choice point because all alternatives for a call are considered to be the result of the computation. Similar to CLP systems the trail is implemented as a value trail. It contains both, the address of the variable and its content.

The stack frame contains the actual information for all local variables of a clause. The register `goalptr` points to the intermediate code of a goal. It allows to find the continuation after a goal has been computed. Register `clauseptr` points to the head of the next alternative clause for the called procedure, and `goalframeptr` points to the stack frame of the calling procedure.

1 P 3

| domain for variable n |
| --- |
| ⋮ |
| domain for variable 1 |
| goalptr |
| clauseptr |
| goalframeptr |
| trailptr |

**FIGURE 4.3.** structure of the stack frame

| reference | |
| --- | --- |
| domain | ref-len |
| alias-prev | alias-next |
| union-domain | union-ref-len |
| union-prev | union-next |

**FIGURE 4.4.** a local variable on the stack

Fig. 4.4 is a detailed description of the stack entry for a local variable. The fields *reference*, *domain*, *ref-len*, *alias-prev* and *alias-next* hold the information derived for a variable by analysing a single alternative of the current goal. The *union* fields get the union of all previously analysed alternatives.

The *reference* field connects the caller's variables with the callee's variables. Aliased variables are stored in a sorted list. The fields *alias-prev* and *alias-next* connect the variables of this list. The *domain* field contains all actual type information at each state of computation. Its contents may change at each occurrence of the variable in the intermediate code. The *ref-len* field contains the length of the reference chain. After analysing an alternative of a goal, the *union* fields contain the least upper bound of the information of all alternatives analysed so far.

## *4.6. Handling of recursion*

The information in the fields of local variables of a clause head is used for fixpoint computation. These fields hold information for these local variables at both, the entry of the clause and at the successful computation of the clause, i.e. the success pattern. When the interpreter reaches the last instruction of a clause (*nogoal*), the success pattern has to be updated. The success pattern fields of the clause's head variables are replaced with the least upper bound of their actual entries (the old success pattern) and the new variable domains. These new domains can be found on the stack after the computation of the clause.

During abstract unification of goal and head arguments the entry pattern for head variables is stored in the intermediate code of the head if this call is computed

the first time. If the intermediate code information already contains entry pattern information, the old information is replaced with the least upper bound of the new and the old information. If the information in the head's intermediate code fields do not change, i.e. the new entry pattern contains more special or equal information than patterns applied previously, there is no sense in a further recomputation of the clause. Instead, information about the clause's actual success pattern is gained from the actual intermediate code fields of the head. This information is then used in the variables occurring in the calling goal, and the interpreter computes the next alternative or the next subgoal of the calling clause if there are no more alternatives to compute. Whenever the success pattern of a clause changes, a flag is set in this clause and all of its calling clauses. The flag marks these clauses for recomputation. Interpretation is iterated until no success pattern changes any more.

### *4.7. Incremental abstract interpretation*

The $VAM_{AI}$ is well suited for incremental abstract interpretation. Incremental abstract interpretation is similar to recomputation if a success pattern has changed. Incremental abstract interpretation starts local analysis with all callers of the modified procedures and interprets the intermediate code of all dependent procedures. Interpretation is stopped when the derived domains are equal to the original domains (those derived by the previous analysis).

To make incremental abstract interpretation possible, pointers to the callers of each procedure are stored in the $VAM_{AI}$ code. They help in finding the top goal of the whole program. The pointer chain for a procedure is used in reconstructing the contents of the stack prior to the call of this procedure. Now, abstract interpretation can be executed as usual. In general, only a small part of the program is reinterpreted. In the worst case, incremental interpretation has to walk through the whole program.

## 5. IMPLEMENTATION ISSUES

### *5.1. A mixed interpreter system*

The advantage of the $VAM_{2P}$ is its compact intermediate code size. The advantage of the $VAM_{1P}$ is its fast execution. It is possible to build an interpreter using a combination of these two abstract machines. The idea is to use the $VAM_{1P}$ only at self recursive calls of the last subgoal and the $VAM_{2P}$ otherwise. The $VAM_{1P}$ code can be either translated to machine code or executed by an intermediate code interpreter. This speeds up the often used loops and uses the compact representation for the other parts of a program.

### *5.2. The incremental compiler*

The compilation of a Prolog program is carried out in five passes (see fig. 5.1). In the first pass a clause is read in by the built-in predicate `read` and transformed to term representation. The built-in predicate `assert` comprises the remaining passes:

- The compiler first translates the term representation into $VAM_{AI}$ intermediate code.
- Incremental abstract interpretation is executed on this intermediate code and the code is annotated with type, mode, alias and dereferencing information.
- The $VAM_{AI}$ intermediate code is traversed again, compiled to $VAM_{1P}$ code and expanded on the fly to machine code.
- The last step is instruction scheduling of the machine code and patching of branch offsets and address constants.

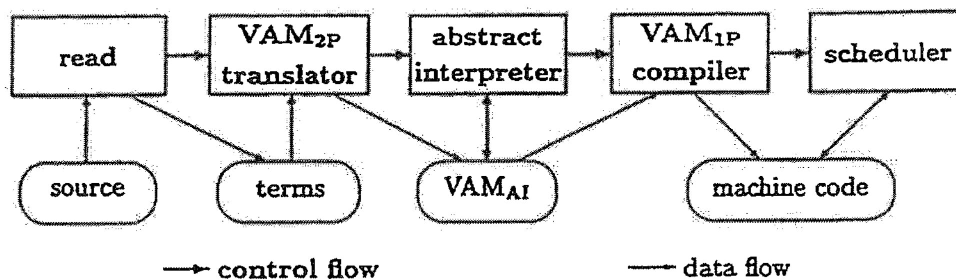



FIGURE 5.1. compiler passes

## 6. RESULTS

To evaluate the performance of the Vienna Abstract Machine we executed the well-known benchmarks described by Beer in [3] on a DECStation 5000/200 (25 MHz R3000) with 40 MB Memory. The following systems were benchmarked: a $VAM_{2P}$ intermediate code interpreter written in C; SICStus Prolog [9], a WAM based intermediate code interpreter implemented in C; the $VAM_{1P}$ compiler with global data flow analysis; the Aquarius compiler of Peter Van Roy [30] and the Parma system of Andrew Taylor [33]. We had no access to the Parma system, so the benchmark data reported in Van Roy's article [29] are used. Table 6.1 shows that the $VAM_{1P}$ compiler produces faster code than the Aquarius system.

Global analysis improved the execution time of the $VAM_{1P}$ only by about 25% [21]. The reason is that the $VAM_{1P}$ gains most of its performance by compile time instruction combination and therefore can use the information about the caller for optimizations. A similar idea is used by Koen de Bosschere and his colleagues in their call forwarding technique [5]. They copy some of the entry actions of each predicate into the call site and gain speedups similar to simple global analysis.

Due to limitations of our prototype system we did not compile large programs. But we expect that the compiler achieves similar performance on larger programs since all arithmetic and type checking and some other built-in predicates are compiled to machine code.

1 P6

| test | interpreters VAM$_{2P}$ ms | VAM$_{2P}$ scaled | SICStus scaled | compilers VAM$_{1P}$ scaled | Aquarius scaled | Parma scaled |
|---|---|---|---|---|---|---|
| det. append | 0.25 | 1 | 1.1 | 26.1 | 19.3 | - |
| naive reverse | 4.17 | 1 | 1.06 | 20.0 | 14.5 | 25.6 |
| quicksort | 6.00 | 1 | 1.1 | 18.1 | 14.9 | 28.0 |
| 8-queens | 65.4 | 1 | 1.1 | 13.5 | 15.4 | - |
| serialize | 3.90 | 1 | 0.83 | 6.84 | 4.26 | 16.4 |
| differentiate | 1.14 | 1 | 0.99 | 8.14 | 7.13 | 14.6 |
| query | 41.7 | 1 | 0.89 | 9.70 | 8.25 | 13.2 |
| bucket | 247 | 1 | 0.88 | 5.24 | 3.71 | - |
| permutation | 2660 | 1 | 0.70 | 6.48 | 6.96 | - |

**TABLE 6.1.** execution speedup, factor of improvement compared to the VAM$_{2P}$

Table 6.2 compares the compile times of the benchmark programs using the VAM$_{1P}$ compiler and the VAM$_{2P}$ and SICStus intermediate code translators. The optimizing VAM$_{1P}$ compiler is about ten times slower than the VAM$_{2P}$ intermediate code translator and about two times faster than the simple SICStus intermediate code translator. The Aquarius compile times (not included in the table) are slower than the VAM$_{2P}$ translator by a factor of about 2000. A direct comparison is not fair since the Aquarius compiler has three passes which communicate with the assembler and linker via files.

| test | intermediate code VAM$_{2P}$ ms | VAM$_{2P}$ scaled | SICStus scaled | native code VAM$_{1P}$ scaled |
|---|---|---|---|---|
| det. append | 5.78 | 1 | 21.5 | 11.43 |
| naive reverse | 7.31 | 1 | 19.3 | 10.5 |
| quicksort | 9.30 | 1 | 23.1 | 9.9 |
| 8-queens | 9.18 | 1 | 19.7 | 11.6 |
| serialize | 11.36 | 1 | 19.2 | 11.22 |
| differentiate | 13.71 | 1 | 30.3 | 11.41 |
| query | 21.05 | 1 | 13.4 | 7.5 |
| bucket | 15.59 | 1 | 12.7 | 7.25 |
| permutation | 4.88 | 1 | 18.1 | 8.88 |

**TABLE 6.2.** compile time, compared to the VAM$_{2P}$

The comparison of the VAM$_{2P}$ with the VAM$_{1P}$ shows that the size of the native code is about ten times larger than the intermediate code of the VAM$_{2P}$ (see table 6.3). The annotated VAM$_{AI}$ intermediate code is about three times larger than the simple VAM$_{2P}$ intermediate code. VAM$_{2P}$ intermediate code has about the same size as SICStus intermediate code.

| test | $VAM_{2P}$ bytes | $VAM_{2P}$ scaled | $VAM_{AI}$ scaled | $VAM_{1P}$ scaled |
|---|---|---|---|---|
| det. append | 288 | 1 | 3.63 | 9.96 |
| naive reverse | 380 | 1 | 3.59 | 11.3 |
| quicksort | 764 | 1 | 2.65 | 9.95 |
| 8-queens | 536 | 1 | 2.95 | 8.25 |
| serialize | 1044 | 1 | 3.33 | 15.7 |
| differentiate | 1064 | 1 | 8.37 | 28.4 |
| query | 2084 | 1 | 0.89 | 3.13 |
| bucket | 996 | 1 | 1.96 | 9.75 |
| permutation | 296 | 1 | 2.77 | 6.21 |

**TABLE 6.3. code size of intermediate representations**

There exists a SICStus native code compiler [15]. Its code executes two to three times faster than the SICStus emulator for small benchmarks and 1.5 to two times faster for large programs. Compilation with the SICStus native code compiler needs two times as long as with the SICStus intermediate code translator. SICStus native code is about twice as large as SICStus intermediate code.

## 7. CONCLUSION

We presented the Vienna Abstract Machine, an abstract machine for Prolog. Different versions are used as an interpreter, a compiler and a base for abstract interpretation. All three versions combine short translation times with fast execution.

## ACKNOWLEDGEMENT

I thank all the people who contributed to the success of the Vienna Abstract Machine: Professor Manfred Brockhaus initiated this project and made this work possible. Eva Kühn implemented the first $VAM_{2P}$ translator, a memory manager and some of the built-in predicates and had to cope with the frequent changes of the instruction set in the beginning. Martin Wais developed an Intel x86 emulator and Herbert Pohlai the portable Motorola 68k emulator. Franz Puntigam implemented a memory management system. Ulrich Neumerkel designed a garbage collector. Thomas Berger implemented the prototype $VAM_{1P}$ compiler with abstract interpretation. Dongyang Cheng helped to implement the $VAM_{AI}$ and the incremental compiler. I express my thanks to Thomas Berger, Manfred Brockhaus, Anton Ertl, Franz Puntigam and the anonymous referees for their comments on earlier drafts of this paper.

1P8

## REFERENCES


1. Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.
2. Gérard Battani and Henry Meloni. Interpréteur du language PROLOG. Dea report, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Université de Aix-Marseille II, 1973.
3. Joachim Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine.* Springer, 1989.
4. James R. Bell. Threaded code. *CACM*, 16(6), June 1973.
5. Koen De Bosschere, Saumya Debray, David Gudeman, and Sampath Kannan. Call forwarding: A simple interprocedural optimization technique for dynamically typed languages. In *POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles on Programming Languages.* ACM, 1994.
6. Maurice Bruynooghe. The memory management of PROLOG implementations. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming.* Academic Press, 1982.
7. Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic programming*, 10(1), 1991.
8. Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Fourth International Conference on Logic Programming.* MIT Press, 1987.
9. Mats Carlsson and J. Widen. SICStus Prolog user's manual. Research Report R88007C, SICS, 1990.
10. Baudouin Le Charlier and Pascal Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS*, 16(1), 1994.
11. Alain Colmerauer. The birth of Prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*, SIGPLAN Notices, pages 37–52. ACM, March 1993.
12. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symp. Priciples of Programming Languages.* ACM, 1977.
13. Saumya Debray. A simple code improvement scheme for Prolog. *Journal of Logic Programming*, 13(1), 1992.
14. Bart Demoen, Andre Marien, and Alain Callebaut. Indexing prolog clauses. In *North American Conference on Logic Programming.* MIT Press, 1989.
15. Ralph Clarke Haygood. Native code compilation in SICStus Prolog. In *Eleventh International Conference on Logic Programming.* MIT Press, 1994.
16. Manuel Hermenegildo, Richard Warren, and Saumya K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(2), 1992.
17. Timothy Hickey and Shyam Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7(3), 1989.
18. Andreas Krall. Implementation of a high-speed Prolog interpreter. In *Conf. on Interpreters and Interpretative Techniques*, volume 22(7) of *SIGPLAN*. ACM, 1987.
19. Andreas Krall. An empirical study of the Vienna Abstract Machine. Bericht TR 1851/90/1, Institut für Computersprachen, TU Wien, 1990.

1Pf


20. Andreas Krall. Clause indexing in VAM and WAM based compilers. In *Second International Workshop on Functional/Logic Programming*, Bericht 9311. Ludwig-Maximilians-Universität München, 1993.

21. Andreas Krall and Thomas Berger. Fast Prolog with a $VAM_{1P}$ based Prolog compiler. In *PLILP'92*, LNCS. Springer 631, 1992.

22. Andreas Krall and Thomas Berger. Incremental global compilation of Prolog with the Vienna Abstract Machine. In *Twelfth International Conference on Logic Programming*, Tokyo, 1995. MIT Press.

23. Andreas Krall and Thomas Berger. The $VAM_{AI}$ - an abstract machine for incremental global dataflow analysis of Prolog. In *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*. Science University of Tokyo, 1995.

24. Andreas Krall and Ulrich Neumerkel. The Vienna abstract machine. In *PLILP'90*, LNCS. Springer, 1990.

25. Micha Meier. Compilation of compound terms in Prolog. In *North American Conference on Logic Programming*, 1990.

26. Micha Meier. Recursion vs. iteration in Prolog. In *Eighth International Conference on Logic Programming*, Paris, 1991. MIT Press.

27. Christopher S. Mellish. An alternative to structure sharing in the implementation of a Prolog interpreter. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*. Academic Press, 1982.

28. Christopher S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1), 1985.

29. Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic programming*, 19/20, 1994.

30. Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1), 1992.

31. Jichang Tan and I-Peng Lin. Compiling dataflow analysis of logic programs. In *Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN*. ACM, 1992.

32. Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In *Sixth International Conference on Logic Programming*, Lisbon, 1989. MIT Press.

33. Andrew Taylor. LIPS on a MIPS. In *Seventh International Conference on Logic Programming*, Jerusalem, 1990. MIT Press.

34. David H.D. Warren. *Applied Logic–Its Use and Implementation as a Programming Tool*. DAI Research Reports 39 & 40, University of Edingburgh, 1977.

35. David H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

200

# CHAPTER 3

# SYMBOLIC EXPRESSIONS AND ABSTRACT PROGRAMS

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

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

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

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

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

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

## 3.1 SYMBOLIC EXPRESSIONS

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

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

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

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

### 3.1.1 Graphical Representation—Trees

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

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

- A ***terminal node, leaf node, atomic node***, or ***atom*** is one that has no further internal structure (namely, it is not a tuple, set, or other complex object). It typically has associated with it a constant, literal, character string, etc.
- A ***nonterminal node*** is one that does have some internal structure, and usually corresponds to a tuple. The internal structure can be represented as arcs that join it to its ***children nodes***, that is, other nonterminals or terminals. It is a ***parent*** to these child nodes. Each child corresponds to a component or element of the tuple represented by the parent.
- A ***root node*** is a nonterminal node that has no parents.



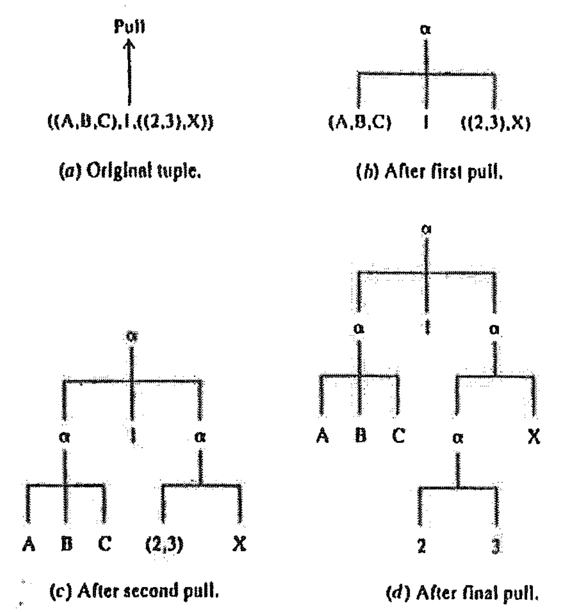



**FIGURE 3-1**
Growing a tree from a tuple.

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

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

The first rule simply guarantees that we are discussing a single object which is a tuple. Note that an object whose value is expressible as a simple constant *is not* a tuple, and is excluded by this rule. However, a

102

tuple of exactly one terminal is. The second rule simply prevents loops in the tree, and the third reinforces the relationship to the tuple form.

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

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

### 3.1.2 Historical s-Expression Notation

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

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

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

### 3.1.3 Dot Notation

A special notation, called ***dot notation***, exists for the case where a nonterminal has exactly two components. If all nonterminals in a tree

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

<terminal> := "appropriate character strings"

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

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

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

**FIGURE 3-2**
Partial BNF for lists and tuples.



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

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

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

### 3.1.4 Implementation of Dot Notation

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

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

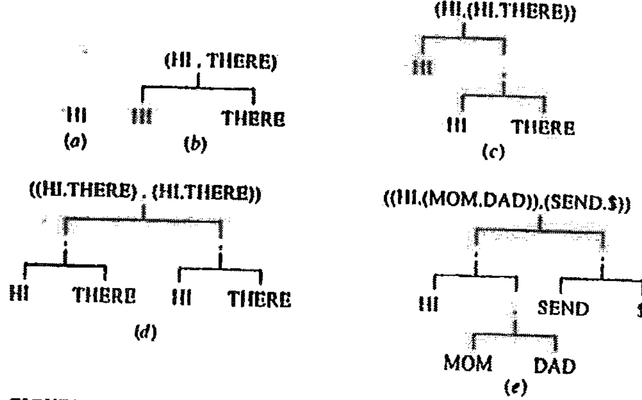



**FIGURE 3-3**
Sample use of dot notation.

262

FIGURE 3-4
Dot notation and its equivalent.

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

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

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

### 3.1.5 Shorthand Cell Drawing

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

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

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

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



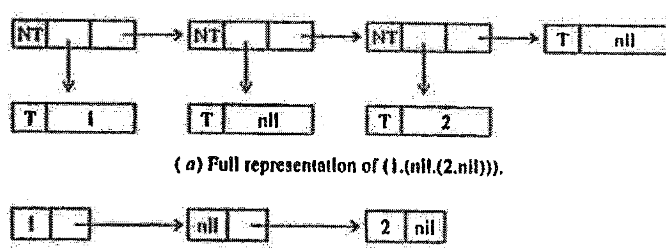



FIGURE 3-5
Drawing condensed s-expression cell diagrams.

### 3.1.6 Implementation of Lists with Dots

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

The most common approach is quite simple:

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

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

$$\text{"}(a_1\ a_2 \ldots a_n)\text{"} \quad \text{or} \quad \text{"}(a_1, a_2, \ldots, a_n)\text{"}$$

has an equivalent dot version as

$$\text{"}(a_1 . (a_2 . (\ldots(a_n\ .nil)\ldots)\text{"}$$

Note the cases for n=1 and 2:

$$(a_1) = (a_1 . nil)$$

$$(a_1\ a_2) = (a_1, a_2) = (a_1 . (a_2 . nil)) \quad \text{(THIS IS NOT } (a_1 . a_2))$$

203

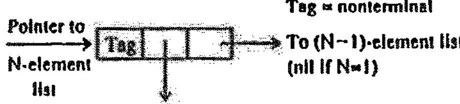


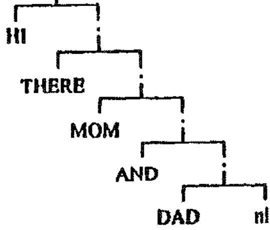

FIGURE 3-6
Linked-list equivalent of dot notation.

Tuple form: (HI,THERE, MOM,AND,DAD)

List form: (HI THERE MOM AND DAD)

Tree form:

Dot form: (HI.(THERE.(MOM.(AND.(DAD.nil)))))




Equivalent linked list:

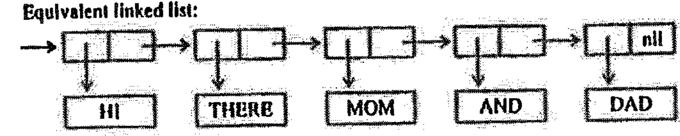



FIGURE 3-7
Multiple children in dot notation.

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

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

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



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

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

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

### 3.1.7 Common Functions and Predicates

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

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

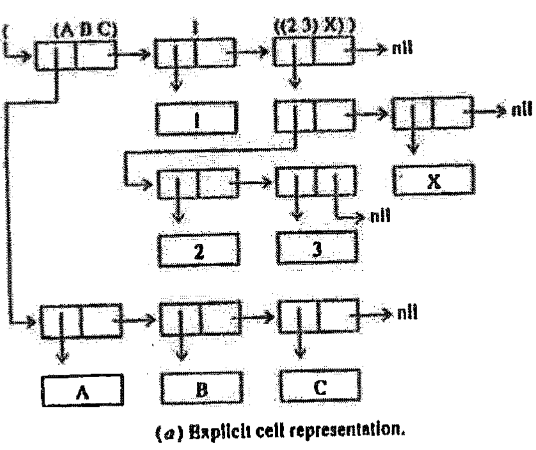
Because such compositions are so typical, a common convention eliminates from the written form all the middle "r(c"s and matching ")"s, leaving an initial "c," a final "r," and an intermediate string of "a"s and "d"s. Thus, car(cdr(cdr(*x*))) is shortened to caddr(*x*), and cdr(car(*y*)) reduces to cdar(*y*).

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

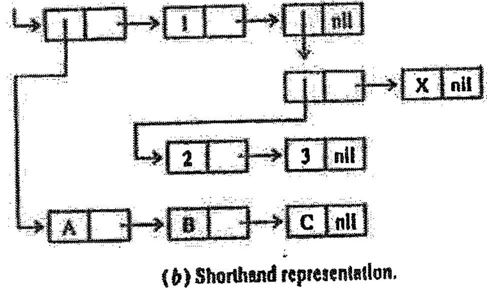
| (*a*) Equivalent representations. | (*b*) Different representations. |
|---|---|
| (1 2 3) = (1, 2, 3) | (1.(2.3)) |
| = (1.(2.(3.nil))) | ((1.2).3) |
| = (1.(2.(3))) | ((1.2) 3) |
| = (1.(2 3)) | ((1 2) 3) |
| = (1 2.(3.nil))) | (1 (2.3)) |
| = (1 2.(3)) | (1 2 3 nil) |
| = (1 2 3.nil) | (1 2 (3.nil))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

FIGURE 3-8
Three-element s-expressions.

402




(*a*) Explicit cell representation.




(*b*) Shorthand representation.

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

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

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

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

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

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

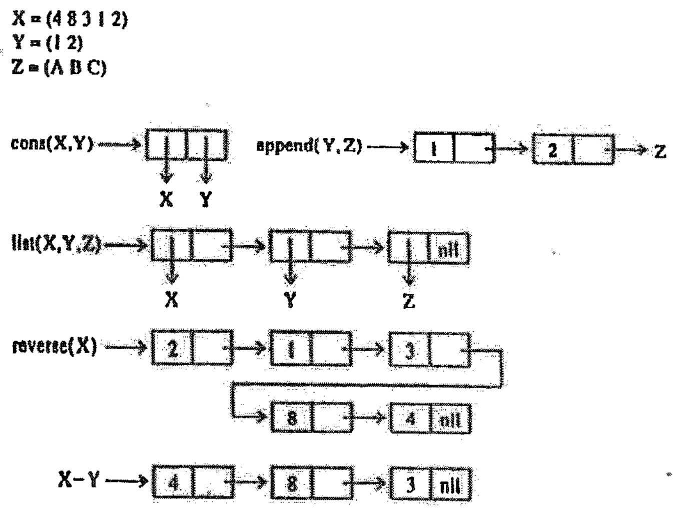
The general execution followed by this function involves making a copy of the list $x$, finding the last cell in the list, and replacing its cdr by a pointer to the root cell of $y$.

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



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

**FIGURE 3-10**
Sample list and dot expressions.




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

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

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

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

Also ***append*** $(x-y, y)=x$ and ***append*** $(x,y)-y=x$.

In addition to these functions there are several common predicates:

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

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

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

## 3.2 ABSTRACT PROGRAMS
(Henderson, 1980)

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

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

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

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

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

The final bit of syntax to be described here is the ***let expression***. This has two parts, the first of which (the definition part) looks like one or more assignment statements in a conventional programming language



```
<abstract-program> := <expr>
<expr> := " any normal mathematical expression"
<expr> := <if-expr> | <let-expr> | <where-expr>
<if-expr> := if <expr> then <expr> else <expr>
<let-expr> := let <definition> in <expr>
            | letrec <definition> in <expr>
<where-expr> := <expr> where <definition>
              | <expr> whererec <definition>
<definition> := <function-eqtn> | <arg> = <expr>
              | <definition> {and <definition>}*
<fcn-eqtn> := <function-name>(<arg>{,<arg>}*) = <expr>
<arg> := <identifier>
<function-name> := <identifier>
```

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

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

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

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

(*b*) Ackerman's function.

**FIGURE 3-13**
Sample abstract programs.

(coupled by the keyword ***and***), and the second (the body) which is a conventional expression. Each definition either defines a function or equates a variable name with some expression. Typically these function definitions are ones that are used inside the expression following in. Each function is defined by giving it a name and appending to it a tuple of formal argument names. These argument names are placeholders for the components of a single argument tuple that the function would accept as input.

To the right of this name and list is an "=" followed by an expres-

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

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

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

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

let $x=3$ in let $x=x+1$ in let $x=x\times 2$ in $7-x$

the only $x$ to take on the value 3 is the one in "$x+1$," which means that the $x$ in "$x\times 2$" takes on the value 4, and the $x$ in "$7-x$" the value 8 (yielding a value of $-1$ for the whole expression).

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

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

Finally, the typical abstract program will often consist of several relatively high-level function definitions followed by a call to one of them



as the ultimate expression to be evaluated. In such cases we will invoke a standard bit of shorthand by deleting the outermost let, and matching ands and in. The function definitions and final expression will be written independent of each other and on separate lines.

## 3.3 RECURSION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Many recursive definitions, such as that for factorial, can be written

factorial(n) = n×(n−1)×(n−2)...×1

Basis test: Does n=0?

Basis case: Value is 1 if n=0

Recursion rule: n×factorial(n−1) if n>1

Thus, as an abstract program definition:
factorial(n) = If n=0 then 1 else n×factorial(n−1)

E.g., factorial(3) → 3×factorial(3−1)
→ 3×(2×factorial(2−1))
→ 3×(2×(1×(factorial(1−1)))
→ 3×(2×(1×1))
→ 6

FIGURE 3-14
Recursive factorial.



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

factorial($n$) = fact($n$,1)
whererec fact($n$,$z$) = If $n$=1 then $z$ else fact($n$−1,$n$×$z$)

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

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

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

fib($n$) = If $n$<2 then 1 else fib($n$−1)+fib($n$−2)

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

### 3.3.1 Common Examples

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

### 3.3.2 Accumulating Parameters

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

202

From prior sections:

```
factorial(x) = if x=0 then 1 else x×factorial(x-1)

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

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

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

member(x,s) = if null(s) then F
                  else if eq(x,car(s))
                      then T
                      else member(x,cdr(x))
```

Other interesting functions:

```
count(x) counts the number of non-nil atoms in x.
count(x) = if atom(x)
               then if null(x) then 0 else 1
               else count(car(x))+count(cdr(x))
E.g., count((A (B C) D)) = 4

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

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

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

FIGURE 3-15
Some common recursive functions.



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

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

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

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

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

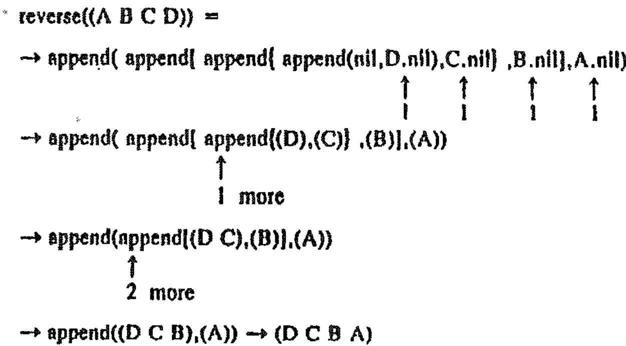



FIGURE 3-16
Counting cons operations in reverse.

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

(*a*) An imperative reverse.

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

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

**FIGURE 3-17**
More efficient reverses.

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

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

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

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

### 3.3.3 The Towers of Hanoi

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

The goal of the problem is to move all disks to the R peg, so that the



result is in the same order as initially. During the solution only one disk can be moved at a time, and no disk can be placed on top of a smaller one. Any of the pegs can be used at intermediate steps.

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

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

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

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

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

Sample reduction sequence:

```
towers-of-hanoi(2) ⟶ reverse(toh(2,Left,Right,Middle,nil))
  ⟶ reverse(toh(1, Middle, Right, Left,
                 (Left.Right).toh(1,Left,Middle,Right,nil)))
  ⟶ reverse(toh(1, Middle, Right, Left,
                 ((Left.Right) (Left.Middle)))
  ⟶ reverse(((Middle.Right) (Left.Right) (Left.Middle)))
  ⟶ ((Left.Middle) (Left.Right) (Middle.Right))
```

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

270

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

## 3.4 ABSTRACT SYNTAX

(McCarthy et al., 1965)

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

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

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

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

1. Test arguments (fragments of program text) to see if they fall in some syntatic category; for example, "Is $x$ a let expression?"
2. Break a fragment of program text of known syntatic type into pieces; for example, "Get the definition text from the let expression $x$."
3. Put pieces back together again; for example, "Create a character string corresponding to the number 3.1415."



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

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

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

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

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

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

272

Predicate functions:

is-a-number(x)—true if x has syntax of a number

is-paren(x)—true if x has outer ( )

is-a-×(x)—true if x is the × operator

is-a-+(x), is-a-/(x),...—similar

Selector functions:

get-value(x)—returns numeric value of x (assuming x is a number)

get-body(x)—returns x without outermost ( )

get-operator(x)—returns outermost ×, /, +, −

get-left(x)—returns left term from outermost function

get-right(x)—returns right term from outermost function

Creator functions:

create-number(x)—converts x into syntatically valid number string

create-infix(left, operator, right)—creates an infix expression

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

**FIGURE 3-19**
Abstract syntax functions for integer expressions.

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

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



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

**FIGURE 3-20**
Abstract interpreter for integer expressions.

## 3.5 PROBLEMS

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

   (((1.(2.(3.nil))).(1.(2.nil))) . nil)
   ((1.2).(2.3))
   ((1 2 (3.nil).(4 5 6)))

2. Draw a picture of the cells needed to implement the s-expressions of Figure 3-7(*b*), showing all cells.
3. How many memory cells are needed for each of the following? Draw a picture (use condensed form).

   ((1 2)(3 4)(5 6))
   ((1.2)(3.4)(5.6))
   (((1.2).(3.4))(5.6))
   (((1.2).(3.4)).(5.6))
   ((1.2).((3.4).(5.6)))
   ((1 2).((3 4).(5 6)))
   (( ) ( ) ( ).(( ) ( )))

4. Find the car, cdr, cadr, cdar, and caddr (if possible) of each of the expressions from the preceding problem.
5. Evaluate Ackerman's function A(2,1) showing all reductions (see Figure 3-13).
6. Write an abstract program that returns the difference of two lists.

212

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

# CHAPTER 4

# LAMBDA CALCULUS

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

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

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

The following sections address the syntax and semantics of lambda calculus, with particular emphasis on the rules for formal parameters and



273

their replacement under substitution. Also addressed is what is possible when an expression has several different internal function-argument pairs that could be applied at the same step. These discussions will lead in later chapters to opportunities for parallelism that are not found in conventional computing.

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

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

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

## 4.1 SYNTAX OF LAMBDA CALCULUS

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

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

The alphabet for this language consists of the set of lowercase letters, plus the characters "(," ")," "|," and "λ" (the Greek character *lambda*). The nonterminal "<expression>" describes all valid lambda calculus expressions, and comes in one of three forms: an application, a function, and an identifier. An *application* expression consists of the concatenation of two other expressions, surrounded by a set of parentheses.

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

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

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

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

**FIGURE 4-1**
BNF syntax for lambda calculus.



Syntatically, the leftmost of such expressions (the one to the immediate right of a "(") represents a function object for which the expression is to be used as its actual and sole argument in a functional evaluation.

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

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

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

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

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

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

## 4.2 GENERAL MODEL OF COMPUTATION

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

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

274

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

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

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

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

Given: ((λx|(xi)) ((λz|((λq|q)z))h))

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

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

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

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

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

FIGURE 4-2
Sample computation.



## 4.3 STANDARD SIMPLIFICATIONS

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

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

$$(\ldots((E_1E_2)E_3)\ldots E_n)$$

we will feel free to write it as

$$(E_1E_2E_3\ldots E_n)$$

Note that the second expression can be interpreted in only one way. The only application that can be made is that which treats $E_1$ as a function and $E_2$ as its argument. Only after this application is performed can we consider $E_3$ as an argument to the function that results from the first application. *Under no conditions* can we consider any other $E_k$ as a function object in an application until it has been absorbed by the left-to-right process and appears in a true function position.

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

$$(\lambda x|(\lambda y|(\lambda z|M)))=(\lambda x|\lambda y|\lambda z|M)$$

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

$$(\lambda xyz|M)E_1E_2E_3$$

still means that the function is (λx|λyz|((*zy*)*x*))=(λx|(λy|(λz|*zyx*))), and takes only the single argument $E_1$ for the formal *x*. The result will be a function (λyz|[$E_1$/x]M), which in turn will process $E_2$, and so on. (A later section will, however, address a multiple simultaneous substitution convention).

Another simplification is to use standard notations for numbers and infix arithmetic expressions whenever it makes sense. This will be justi-

72

fied in a later section where we give exact lambda equivalents to standard integer arithmetic.

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

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

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

## 4.4 IDENTIFIERS

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

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

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

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

Although no single instance can have more than one value, it is possible for multiple instances of the same symbol in the same expression to have different values. This occurs when multiple function definitions are embedded within each other, all using some common identifier character



in their list of formal arguments. Each of these function definitions controls a different, nonoverlapping part of the expression. Figure 4-3 gives an example of such an expression and the values taken on by different instances of the identifier $x$.

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

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

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

1. E=$x$.
2. E=($\lambda y$|A), $y \neq x$, and $x$ occurs free in A.
3. E=(AB) and $x$ occurs free in either A or B.

(λx|(λx|(λx|xxx)(bx)x)(ax)) c
→ [c/x](λx|(λx|xxx)(bx)x)(ax)
→ (λx|(λx|xxx)(bx)x)(ac)
→ [(ac)/x]((λx| xxx)(bx)x)
→ ((λx|xxx)(b(ac))(ac))
→ [(b(ac))/x](xxx)(ac)
→ (b(ac))(b(ac))(b(ac))(ac)

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

**FIGURE 4-3**
Sample nested functions with same identifier.

216

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

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

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

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

## 4.5 SUBSTITUTION RULES

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

The difference is in the first substitution of the second sequence. If we blindly apply the function (λ$q$|(λ$p$|$p$($pq$))(λ$r$|+ $p$ $r$)) to its argument (+ $p$ 4), we substitute the latter for $q$ inside the function's body. However,

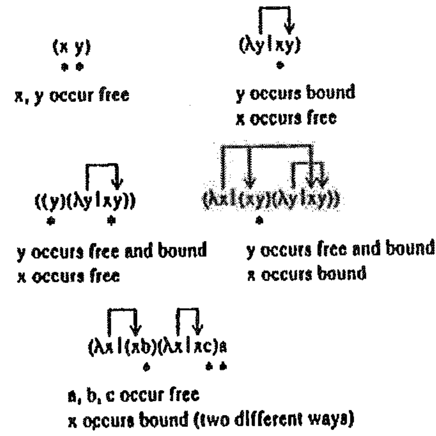



FIGURE 4-4
Sample identifier occurrence patterns.



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

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

(*a*) A valid reduction sequence.

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

(*b*) An improper substitution.

FIGURE 4-5
Example of a potential substitution problem.

the $p$ in (+ $p$ 4) is free in that expression, while the $p$'s in the function body subexpression (λ$p$|$p$($pq$)) are bound. A simple substitution will change this formerly free $p$ to a bound one, changing the meaning of the expression. In a proper substitution, the free $p$ should remain free even after the application, because its "value" should remain unchanged.

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

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

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

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

For (λx|E)A → [A/x]E → E'

Rules for substitution [A/x]E are:

1. If E is an identifier y
   then if y=x, then E' = A
   else (in this case, y≠x) E' = E.
   Examples:
   - (λx|x)M → [M/x]x → M
   - (λx|y)M → [M/x]y → y
2. If E = (BC) for some expressions B and C
   then E' = (([A/x]B)([A/x]C)).
   Example: (λx|xyx)M → (([M/x](xy))([M/x]x)) → MyM
3. If E = (λy|C) and C some expression
   a. and y=x, then E' = E.
      Example: (λx|(λx|xN))M → (λx|xN)
   b. and y≠x and y does not occur free in A
      then E' = (λy|[A/x]C)
      Example: (λx|(λy|xyN))M
      → [M/x](λy|xyN))
      → (λy|[M/x](xyN)))
      → (λyMyN))
      → (λ|MyN)
   c. (RENAMING RULE) otherwise (i.e., y occurs free in A)
      E' = (λz|[A/x]([z/y]C)), where z is a new symbol never used before (or at least not free in A)
      Example: (λx|(λy|xyN))(ay)
      → [(ay)/x](λy|xyN) (note y occurs free in (ay))
      → (λz|[(ay)/x][z/y](xyN))
      → (λz|[(ay)/x](xzN))
      → (λz|(ay)zN)

FIGURE 4-6
Substitution rules for lambda calculus.

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

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



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

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

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

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

## 4.6 CONVERSION RULES, REDUCTION, AND NORMAL ORDER

(Stoy, 1977, pp. 64–67)

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

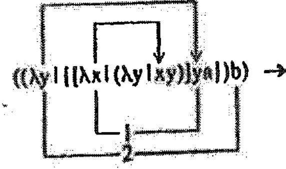
In this context it becomes relevant to discuss under what conditions we know that two separate expressions are in fact the same, and what kinds of standard forms there are that simplify such comparisons.

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

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

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

FIGURE 4-7
A proper reduction sequence.




- If Application 2 done first,
  → [b/y]{[λx|(λy|xy)]ya} = {[λx|(λy|xy)]ba}
  → ([b/x](λy|xy))a → (λy|by)a → ba
- If Application 1 done first AND NO RENAMING (an error),
  → ((λy|{([y/x](λy|xy))a})b) = ((λy|{(λy|yy)a})b)
  "name conflict"
  ... → aa
- If Application 1 done first AND renaming used correctly:
  → ((λy|{([y/x](λy|xy))a})b) → ((λy|{(λz|[y/x][z/y](xy))a})b)
  Change name to "z"
  → ((λy|{(λz|[y/x](xz))a})b) → ((λy|{(λz|(yz))a})b)
  → ((λy|ya)b) → ba

**FIGURE 4-8**
Sample renaming substitution.

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

The ***alpha conversion*** rule corresponds to a simple and safe "renaming" of formal identifiers. Two expressions are the same if they differ

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

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

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

**FIGURE 4-9**
Reduction rules for expression equivalence.



(λx|+ x 4)3 → [3/x](+ x 4) → (+ 3 4) → 7
(λx|+ x 4)(+ 2 z) → [(+ 2 z)/x](+ x 4) → (+(+ 2 z)4)
(λx|+ x y) 3 → (+ 3 y)
(λx|λy|+ x y) 3 → [3/x](λy|+ x y) → (λy|+ 3 y)
(λx|λx|+ x 3) 4 → [4/x](λx|+ x 3) → (λx|+ x 3)
(λx|λy|+ y 3) 4 → [4/x](λy|+ y 3) → (λy|+ y 3)
(λx|λy|+ y x)(+ z 4) → λy|[(+ z 4)/x](+ y x)
  → (λy|+ y(+ z 4))
(λx|λy|+ y x)(+ y 4) → λz|[(+ y 4)/x][z/y](+ y x)
  → λz|(+z(+ y 4))
(λq|((λp|p(pq))(λr|+ p r))) (+ p 4)
  → [(+ p 4)/q]((λp|p(pq))(λr|+ p r))
  → ([(+ p 4)/q](λp|p(pq))) ([(+ p 4)/q](λr|+ p r))   RULE 2
  → ([(+ p 4)/q](λp|p(pq))) (λr|+ p r)   RULE 3b
  → {λs|[(+ p 4)/q][s/p] (p(pq))}(λr|+ p r)   RULE 3c
  → (λs|[(+ p 4)/q](s(sq))) (λr|+ p r)   AFTER RENAME
  → (λs|s(s(+ p 4)))(λr|+ p r)
(λx|xx)(λx|xx) → [(λx|xx)/x](xx)
  → (λx| xx)(λx| xx)
(λx|xxx)(λx|xxx)
  → (λx|xxx)(λx|xxx)(λx|xxx)
  → (λx|xxx)(λx|xxx)(λx|xxx)(λx|xxx)
  → . . . . an infinitely long concatenation of (λx|xxx)

**FIGURE 4-10**
Sample equivalent expressions.

only in the symbol names they give to their function objects' formal parameters.

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

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

(λx|Ex)A with no free x's in E

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

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

Using eta conversion first,

$$(\lambda x|Ex)A \rightarrow EA$$

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

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

$$E_n \rightarrow (\lambda x_n|(\lambda x_{n-1}|(\ldots(\lambda x_1|Ex_1)\ x_2)\ldots)x_n) \text{ for any } n$$

where no $x_k$ appears free in E. Note that because of the ")" between $x_k$ and $x_{k+1}$, this is *not* the same as $(\lambda x_1 \ldots x_n|(\ldots(Ex_1)x_2)\ldots x_n)\ x_2)\ldots x_n)$.

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

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

## 4.7 THE CHURCH-ROSSER THEOREM
(Stoy, 1977, p. 67)

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

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

The second theorem, named the ***Church-Rosser Theorem II***, states



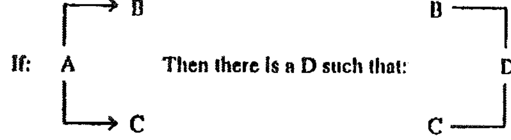



**FIGURE 4-11**
The Church-Rosser theorem.

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

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

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

## 4.8 ORDER OF EVALUATION

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

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

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

**FIGURE 4-12**
Normal-order reduction.

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

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

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

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

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

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

This example is indicative of the general results about these two evaluation orders. Normal-order evaluation guarantees a reduced expression if one exists, but at the expense of potential duplication of evaluations. Applicative order will not give a different answer. If it terminates, it will yield an expression equivalent to that from the normal-order sequence, and it will do so without duplicate computation. Further, reducing the function body and the argument can be done in parallel, offering

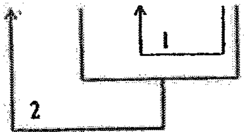



**FIGURE 4-13**
Multiple applications.



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

(*a*) Normal order reduction.

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

(*b*) Applicative-order reduction.

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

(*c*) A mixed order of evaluation.

**FIGURE 4-14**
Different reduction orders.

the possibility of even more time-efficient evaluation. However, it is possible that the applicative-order reduction will never terminate, but loop forever.

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

## 4.9 MULTIPLE ARGUMENTS, CURRYING, AND NAMING FUNCTIONS

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

221

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

(λxyz|E)ABC
→ (λx|(λy|(λz|E)))ABC
→ ([A/x](λy|(λz|E)))BC
→ ([B/y]([A/x](λz|E)))C
→ [C/z]([B/y]([A/x]E))

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

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

$$((\lambda x_1 \ldots x_n | E)\ A_1 \ldots A_m)$$

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

$$[A_1/x_1, \ldots, A_m/x_m]E$$

In either case, if any of the symbols $x_{m+1}$ through $x_n$ appear free in $A_1$ through $A_m$, then a renaming of them is needed before the substitution can go forward. However, if the multiple substitution truly is done simultaneously, then there are no renaming problems within the first m x's.

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

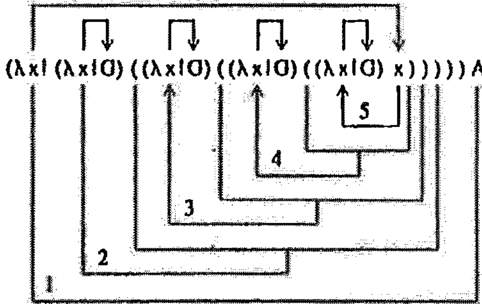
A very common example in lambda calculus where this multiple substitution is useful, even when there is only one argument, is in function definitions that employ within them several applications involving the same subfunction. (This will be particularly true for recursive definitions, where this subfunction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

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

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

Assume function f desired, where:
f(x) = g(g(g(g(x))))  (e.g., taking x to the 16th power)
g(x) = (λx|O)  (e.g., squaring x)

Brute-force approach:




Using function naming:

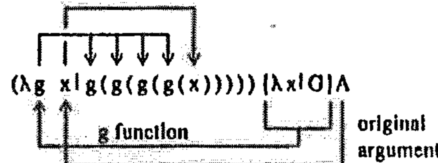



**FIGURE 4-15**
Function naming example.

## 4.10 BASIC ARITHMETIC IN LAMBDA CALCULUS

(Seldin and Hindley, 1980; Stoy, 1977)

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

Other standard integer operations can be defined similarly.

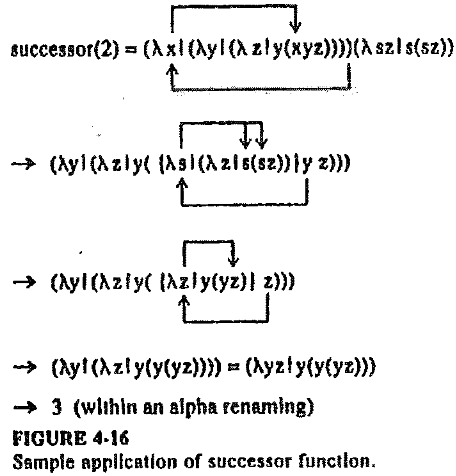



**FIGURE 4-16**
Sample application of successor function.

## 4.11 BOOLEAN OPERATIONS IN LAMBDA CALCULUS

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

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

$$\text{true} = T = (\lambda xy|x)$$
$$\text{false} = F = (\lambda xy|y)$$

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

$$\text{If } P=T \text{ then } PQR = T\ Q\ R = [Q/x,R/y]x = Q$$
$$\text{If } P=F \text{ then } PQR = F\ Q\ R = [Q/x,R/y]y = R$$

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

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

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



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

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

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

**FIGURE 4-17**
**Standard logical connectives.**

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

Examples:

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

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

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

**FIGURE 4-18**
**A lambda zero test predicate.**

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

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

422

f(a) = if zero(a) then 0 else a + 1
= (λa|zero a 0 (successor a))
= (λa|(λx|x (λn|z|z) (λw|w(λxy|y)(λxy|x)) (λn|z|z))
a
(λn|z|z)
((λxyz|y(xyz)) a))

FIGURE 4-19
A complex lambda function.

## 4.12 RECURSION IN LAMBDA CALCULUS

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

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

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

$((\lambda x|xx)(\lambda x|xx))$
$\rightarrow [(\lambda x|xx)/x](xx)$
$\rightarrow ((\lambda x|xx)(\lambda x|xx))$

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

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

$((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx)))$
$\rightarrow [(\lambda x|\mathbf{R}(xx))/x](\mathbf{R}(xx))$
$\rightarrow \mathbf{R}\ ((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx)))$
$\rightarrow \mathbf{R}\ (\mathbf{R}\ ((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx))))$
$\rightarrow \ldots$
$\rightarrow \mathbf{R}\ (\mathbf{R}\ (\mathbf{R}\ (\ldots((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx))))\ldots)$

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



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

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

$\mathbf{YR} \rightarrow [\mathbf{R}/y]((\lambda x|y(xx))(\lambda x|y(xx)))$
$\rightarrow ((\lambda x|\mathbf{R}(xx))(\lambda x|\mathbf{R}(xx)))$
$\rightarrow \mathbf{R}(\mathbf{YR})$
$\rightarrow \ldots$
$\rightarrow \mathbf{R}(\mathbf{R}(..\mathbf{R}(\mathbf{YR})..).$

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

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

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

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

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

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

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

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

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

225

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

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

FIGURE 4-20
Reduction of 4

## 4.13 PROBLEMS

1. Compute all the variations in application sequences possible for Figure 4-2 and show that they yield the same answer.
2. In the following lambda expressions, which identifiers are free, and which are bound, and to which lambda.
   a. ((λy|yxx)y x)
   b. ((λx|(λx|(λx|x)x)x)x)
   c. (λx|(xy(λy|((xy)(λx|wxyz))(λz|wyz))))
   d. (λx|y(λy|x(λx|xz(λy|yx))))
3. Evaluate, showing each reduction: (λf|+(f 3)(f 4))(λx|×xx).
4. Show that the lambda expression not, and, or work (show what happens when presented with all combinations of T and F).
5. Define a lambda expression for xor (*exclusive or*). Show both in terms on T and F, and when the body of the function has been fully reduced to normal order.
6. Show that ($Z_k$ (λxyz|y(xyz)) 0) → $Z_k$. How does this reinforce the notion that $Z_k$ is a reasonable representation of the integer k?
7. Add 3 and 2 using the lambda definitions of add and integers. Show all steps.
8. Multiply 3 and 2 using the lambda definitions of multiply and integers.
9. Assuming that $Z_i$ and $Z_j$ represent lambda expressions for integers as defined in the text, evaluate each for $Z_i$=2 and $Z_j$=3. What exactly is the second function?

   (λ*mnf*|*m*(*nf*)) $Z_i$ $Z_j$

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



226

# CHAPTER 5

# A FORMAL BASIS FOR ABSTRACT PROGRAMMING

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

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

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

The following sections give a BNF description for the syntax of an





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

## 5.1 A BASIC SYNTAX

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

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

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

<function-name> := <identifier>

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

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

<body>:=<expression>

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

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

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

**FIGURE 5-1**
BNF for abstract programs.

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

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

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

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

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

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

## 5.2 CONSTANTS

Semantically, a ***constant*** is any object whose name denotes its value directly. In abstract programming we will assume that at a minimum we

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

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

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

**FIGURE 5-2**
Sample equivalent definitions.



have syntax for constants of types boolean, integer, and character string, all of which can be described by their normal written form, and all of which translate directly into equivalent lambda expressions. These lambda expression equivalents represent their semantic meaning.

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

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

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

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

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

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

## 5.3 FUNCTION APPLICATIONS

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

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

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

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

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

As before, we give a meaning to such an application by giving its lambda equivalent, namely, "$f(A_1,\dots,A_n)$" is equivalent to "$(F B_1 \dots B_n)$," where F is the lambda-expression form of f [i.e., something like $(\lambda x_1 \dots x_n|E)$] and $B_k$ is the lambda equivalent of $A_k$.

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

## 5.4 CONDITIONAL EXPRESSIONS

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

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

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

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

if $P_1$ then $E_1$ else if $P_2$ then $E_2$ else...if $P_n$ then $E_n$ else $E_{n+1}$

The meaning of this is that we find the first $P_k$ that is true, and return as its value $E_k$. If none of the Ps are true, return $E_{n+1}$.

The translation of this to lambda form is direct:

$(P_1\ E_1\ (P_2\ E_2\ (\dots(P_n\ E_n\ E_{n+1})\dots)$

For simplicity, the keyword elseif will be used for such combinations.



## 5.5 LET EXPRESSIONS—LOCAL DEFINITIONS

Many conventional programming languages offer a *macroexpansion* capability whereby for a certain region of text in the program (called the *scope*) all occurrences of a certain identifier are to be replaced by some predefined text string. In other languages, where the passing of arguments to a procedure is by *call-by-name* semantics, the occurrence of a formal argument in the body of the procedure actually means replacement of that argument by the result obtained by evaluating the text corresponding to the actual argument.

Both of these mechanisms have two things in common. First, they simplify the writing of expressions by letting some identifier "stand for" some other expression. Second, the use of this identifier is very controlled; throughout the entire scope the identifier's value as an externally defined expression (almost) never changes.

Abstract programming has a notation very akin to a cleaned-up combination of macros and call-by-name. The simplest form of this notation (called a *let expression*) is

let (identifier)=(expression) in (body)

where (body) is itself an arbitrary expression.

This whole expression is equivalent in value to a copy of the body where every free occurrence of the identifier in the definition part is replaced by the expression in the definition. In terms of its lambda equivalent (from which we get its exact meaning), a let expression of the form "let $x$=A in E" is the same as "$(\lambda x|E)A$," and means that we must perform the substitution $[A/x]E$. The let expression is totally equivalent to an application of an anonymous function to an argument, where the formal parameter for the function is the identifier in the definition, the body of the function is the body of the let, and the argument is the right-hand side of the application.

As discussed earlier, this conversion process brings with it a precise

228

definition of which occurrences of $x$ in E are targets of the substitution, and what happens if a part of E that contains a free $x$ also binds a symbol that is free in A (we must "rename" the former binding variable). Figure 5-3 diagrams a sample let demonstrating all these possibilities.

### 5.5.1 Local Nonrecursive Functions

This definition of let places no constraints on the type of expression of the right-hand side of the "=" in the definition. In particular, it may be an arbitrary lambda function, in which case the identifier in the definition becomes a ***local function*** to the body of the let, as in

let *square* = (λx|(x×x)) in *square*(*square*(2))

This is such a handy notation that our definition of abstract programming includes a special form of the let definition that eliminates the λ from the definition's body, and transfers the formal argument from the body's lambda to be inside some parentheses next to the function's name. Thus we have as a form of let:

let ⟨function-name⟩(⟨id⟩ {,⟨id⟩}$^*$) = ⟨expression⟩ in ⟨body⟩

where this is totally equivalent to

let ⟨function-name⟩ = (λ⟨id⟩{⟨id⟩}$^*$ |⟨expression⟩) in ⟨body⟩

or

(λ⟨function-name⟩|⟨body⟩) (λ⟨id⟩⟨id⟩$^*$|⟨expression⟩)

Thus the above example is equivalent to the more readable

let square($x$) = $x$×$x$ in square(square(2))

Either one is equivalent to (λs|s(s 2)) (λx|x×x).

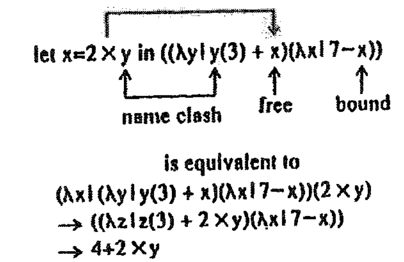



FIGURE 5-3
A sample let expression.



Figure 5-4 diagrams another example in detail. In both cases, the new notation is quite easy to read, and agrees closely with notation found in many conventional languages. However, the reader should take care to study these examples closely and verify exactly how each part of the new let form migrates into the multiple functions in the lambda equivalent.

### 5.5.2 Nested Definitions

Given that a let expression is itself an expression, it is also possible to nest one inside another, particularly inside the other's body. Again the meaning of this is an exact extension of the basic let equivalence. Figure 5-5 diagrams a triply nested let and its lambda equivalent.

As before, it is important to apply the rules for identifying free instances and performing the resulting substitutions properly. The second example in Figure 5-5 diagrams such a case where the middle let has two different $x$'s, the $x$ in the definition expression that is receiving the value 7 from the outermost let, and the $x$ that is equivalent to the value 8 and is being substituted into the third let.

### 5.5.3 Block Definitions

One important consequence of the nesting rules for let expression is that identifiers that appear free in the expressions for definitions of inner lets are perfectly valid candidates for replacement by definitions in more outer lets. Thus, in "let $x$=A in let $y$=...$x$...in...," the free $x$ in the $y$ definition is replaced by A.

There are many cases, however, where what is desired is a simultaneous replacement of several definitions into a let body, with no cross-substitutions among the definitions. In our abstract programming language the ***and*** form of definitions permits this simultaneity. An expression of the form

let $x$=A and $y$=B and $z$=C in E

let f = (λxy|x+3×y+1) in f(2+x,7)

or

let f(x,y) = x+3×y+1 in f(2+x,7)

are both equivalent to:

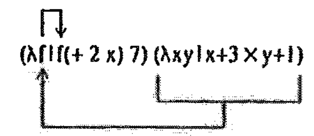



FIGURE 5-4
Ways to define local functions.

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien

230

```
let x = A in
   let y = B in
      let z = C in E

(λx| (λy| (λz|E) C) B) A

Substitution eqvt: [A/x]([B/y]([C/z]E))

Sample: let x=7 in
            let x=x+1 in
               let f(y)=y+2×x in x×f(3)

  → let x=8 in let f(y)=y+2×x in x×f(3)
  → let f(y)=y+16 in 8×f(3)
  → 8×(3+16) → 152

Equivalent: (λx|[λx|(λf|x×f(3))(λy|y+2×x)](x+1))7
  ≡[7/x]([x+1/x]([(λy|y+2×x)/f](x×f(3))))
  →[7/x]([x+1/x](x×(λy|y+2×x)(3)))
  →[7/x]([x+1/x](x×(3+2×x)))
  →[7/x((x+1)×(3+2×(x+1)))
  →((7+1)×(3+2×(7+1)))
  →152
```

FIGURE 5-5
Nested lets.

means that $x$, $y$, and $z$ should be simultaneously replaced in E by A, B, and C, respectively, and that any free $x$, $y$, and $z$'s in A, B, or C should remain free and should not be replaced by A, B, or C as happens in the nested form. (Note that an $x$ that is free in A stays free anyway.)

In terms of pure lambda equivalents, such and forms correspond to creating a function application where the new anonymous function has multiple arguments, namely:

(λ$xyz$|E) A B C

Evaluating such an expression results in either the triply nested substitution

[C/z]([B/y]([A/$x$]E))

or the equivalent simultaneous substitution [C/z,B/y,A/$x$]E. Note that when the nested substitution is performed inside out and one at a time, the renaming rule will guarantee that any free $z$'s in A or B and any free



variables in the arguments will stay free and unchanged. Figure 5-6 gives an example of this.

### 5.5.4 Where Expressions

Finally, there are cases where understanding of an expression is enhanced by placing the definitions after the expression they affect. Our version of abstract programming includes the where expression to permit this syntatic reversal; i.e.,

(body) where (definition)

is totally equivalent to

let (definition) in (body)

Further, the where expression may be expanded in an and form, again with exactly the same meaning as the let version.

## 5.6 RECURSIVE DEFINITIONS

One limitation of the let and where expressions is that they do not permit recursion in a function definition. An expression of the form

let $f(n)$=if zero($n$) then 1 else $n\times f(n-1)$ in $f(4)$

does not result in the intended effect of having $f$ call itself recursively when $n \neq 0$. Instead, the $f$ in the definition's expression is replaced by whatever definitions of $f$ exist in surrounding expressions.

The brute-force solution to this is to use the ***fixed-point combinator***

```
let x=(y x 1) and y=(zyx 2) and z=(yxz 3) in xyz
≡ (λxyz|xyz) (yx 1) (zyx2) (yxz 3)
→ {[(yx 1)/x](λyz|xyz)}(zyx2) (yxz 3)
→ {(λqz|(yx 1)qz)} (zyx2) (yxz 3)—"y renamed to q"
→ {[(zyx2)/q](λz|(yx 1)qz)} (yxz 3)
→ {(λr|(yx 1)(zyx2)r)} (yxz 3)—"z renamed to r"
→ (yx 1) (zyx2) (yxz 3)
```

FIGURE 5-6
Sample and form of let.

231

Y, defined earlier, at carefully selected spots in the let expression. Unfortunately, the result is not terribly readable.

The alternative used in abstract programming is the letrec expression, a recursive form of the standard let. Here the major difference is that any free occurrence of the definition's identifier in the definition's expression is replaced by the expression itself. Thus, in

letrec *f*(*n*)=if zero(*n*) then 1 else *n*×*f*(*n*−1) in *f*(4)

the free occurrence *f* in *f*(*n*−1) is replaced (recursively) by the whole expression

if zero(*n*) then 1 else *n*×*f*(*n*−1)

All uses of *f* in the letrec's body now are replaced by this whole recursively defined expression.

The conversion from this notation to a pure lambda equivalent is direct. Given "letrec *f*=A in E," we form an application where the function is an anonymous function whose formal parameter is *f* and whose body is E. This is just as with let. The actual argument to this function is, however, different. Instead of just using A, we create the object (λ*f*|A) and use that as an argument to the function Y. The resulting application is the recursive function we want and is then used as the argument to the outermost application.

In total, the lambda equivalent for

letrec *f* = A in E

is

(λ*f*|E) (Y (λ*f*|A)) = (λ*f*|E) ((λy|(λx|y(xx))(λx|y(xx))) (λ*f*|A))

A letrec expression is really useful only for defining local functions (try out your patience on "letrec *x*=*x*+1 in *x*"). Consequently, one would expect the "A" in "letrec *f*=A in E" to be a lambda function itself (of the form (λ*x*|B)). As with let, this is permissible, but a more human-readable form is possible if we permit listing the arguments of *f* next to *f*, with only the function body on the right-hand side.

In summary, the conversion process for an expression of the form "letrec *f*(*x*)=B in E" involves making a lambda function (λ*x*|B) and creating a nested set of applications from it as defined above. To verify this, consider:

letrec *f*(*n*)=if zero(*n*) then 1 else *n*×*f*(*n*−1) in *f*(4)

Its pure lambda equivalent is only one reduction away from that detailed at the end of Chapter 4:



(λ*f*|*f* 4) (Y (λ*f*n|zero *n* 1 (×*n* (*f* (−*n* 1)))))

The nesting of letrecs is also permissible, and mirrors nested lets directly. Here the definition of any recursive function can reference any recursive function defined in surrounding letrecs, but cannot use definitions in deeper letrecs.

### 5.6.1 Mutually Recursive Functions

In contrast to let, however, there are some subtle differences between multiple definitions in the ***and form*** of a letrec versus a let. In the latter there is absolutely no connection between the expressions used in the definitions, even though they may use some of the same identifiers being given definitions. In the letrec form, however, the expression in each anded definition has complete access to every other definition. The result is a set of ***mutually recursive functions*** that are defined together.

This mutual dependency makes conversion of a multiple-definition letrec somewhat more challenging than anything we have done yet. Consider, for example, the expression

letrec *f*(*x*)=A and *g*(*x*,*y*)=B in E

where the expressions A and B both involve applications using *f* and *g*.

As before, this conversion consists of an application of the form (λ*fg*|E)FG, where F and G are expressions for the functions *f* and *g*. As with a single recursion, these F and G contain objects for *f* and *g* that accept as arguments what will be copies of both *f* and *g*, along with their natural arguments. They are of the form (λ*fgx*|A) for *f* and (λ*fgxy*|B) for *g*.

Define combinators:

$Y_1$ = (λfg|RRS)

$Y_2$ = (λfg|SRS)

where:

R = (λrs|f(rrs)(srs))

S = (λrs|g(rrs)(srs))

Properties:

$Y_1$ F G = F ($Y_1$ F G) ($Y_2$ F G)

$Y_2$ F G = G ($Y_1$ F G) ($Y_2$ F G)

**FIGURE 5-7**
Pairwise mutually recursive combinators.

```
letrec f(x) = A and g(x,y) = B in B
is equivalent to
(λfg|B) (Y₁(λfgx|A)(λfgxy|B)) (Y₂(λfgx|A)(λfgxy|B))
```

FIGURE 5-8
Conversion of a dual-definition letrec.

The hard part with this is completing the expansion of F and G. We have to use something like the Y expression used above, but Y by itself will not work. It involves recursion over only one function, and has no way of introducing dependencies on another. The Y equivalents that do work are shown in Figure 5-7. Each one accepts two arguments (the nonrecursive forms of the two functions) and creates an expression consisting of one of the nonrecursive forms applied to two arguments which represent the recursive forms of both *f* and *g*. When this application is evaluated, it gives us a curried function whose remaining arguments are the natural arguments for the function. This is exactly what we want to substitute into E.

Figure 5-8 gives a complete conversion. It is left as an exercise to the reader to expand the approach to handle the case where there are three or more mutually recursive functions defined as anded terms in a single letrec.

As with let, the whererec expression is identical to the letrec, but with the definitions given after the body rather than before.

## 5.7 GLOBAL DEFINITIONS

One final syntatic simplification deals with the common habit of describing different but related functions in different places in a document. Strictly speaking all such definitions should be collected in a single expression, with the end problem to be solved written as an expression involving these definitions as the body of the outermost let or letrec. This should include all the "built-in" functions which we all know can be written but do not want to spend the time or paper describing (e.g., square root or car, cdr, cons,...).

The solution to be assumed in this text is that all expressions that are not obviously self-contained are assumed to be lumped as and definitions in a single large letrec, with other and definitions assumed to cover whatever functions are missing. Figure 5-9 gives an example of this.

## 5.8 HIGHER-ORDER FUNCTIONS

As has been said before, in pure lambda calculus everything is a function. In abstract programming we have simplified much of the notation to hide



```
Assume the following are defined at different places in a document:
  f(x) = ...
  g(x,y) = ...
  h(z) = ...
  s = 0
  msg = "Hi There"
  basevalue = f(s)
with g(msg,basevalue) as the desired program output.
This is equivalent to the abstract program:
  letrec f(x) = ...
  and g(x,y) = ...
  and h(z) = ...
  and s = 3
  and msg = "Hi There"
  and basevalue = f(s)
  and...
  in g(msg,basevalue)
```

FIGURE 5-9
Handling of global definitions.

many common function equivalents (such as integers or booleans), but not deleted the capability. It is still perfectly acceptable to either pass a function as an argument or get one as a result. To distinguish such functions from the more mundane ones, we call them ***higher-order functions***.

Figure 5-10 gives some particularly useful higher-order functions, with an example for each. They are so useful in practice that many real functional languages build them in. The first (and most famous) one, *map*, takes as its arguments some arbitrary function and some other arbitrary list of objects. The result is a new list of exactly the same length as the list argument, with the k-th element equaling the result of applying the function input to the k-th element of the input list. *Map2* is identical, except that it expects two equal length lists, and applies matching elements of both to the function argument.

A slightly different higher-order function is *reduce*. Here there are three arguments: a function, an arbitrary object, and a list. If the list is not empty, the value returned is the result of applying the function argument to the first element of the list and to the object resulting from recursively applying reduce to the rest of the list. When the list finally empties, the value returned is the second argument. As an example, using "+" as the function input results in adding up all elements of the list, plus the original input for *t*.

A function very similar to *map* is *vector*. Instead of applying a single function argument to all elements of a list of objects, this function applies each element of a list of functions to a single object and collects the results in a list.

223

```
map(f,x) = "apply function f to each element of list x"
         = if null(x) then nil
           else cons(f(car(x)), map(f,cdr(x)))
           e.g., map((λz|× 3 z),(3 5 7)) = (9 15 21)

map2(f,x,y) = "f applied to matching elements of lists x and y
            = if null( x) then nil
              else cons( f(car( x),car( y)), map2( f,cdr( x),cdr( y)))
              e.g., map2((λxy|x+ y),(1 2 3),(4 5 6)) = (5 7 9)

reduce(f,t,x) = "recursively apply f to each x and prior result"
              = if null(x) then t
                else f(car(x),reduce(f,t,cdr (x)))
                e.g., reduce( + ,0,(1 2 3)) = 1+(2+(3 +0)) = 6

vector(f,x) = "apply list of functions f to x"
            = if null(f) then nil
              else cons((car(f)) x, vector(cdr(f), x))
              e.g., vector(((λx|2× x) (λz|z−6) (λx|x)),3) = (6 −3 3)

while(p,f,x) = "while loop"
             = if p(x) then while(p,f,f(x)) else x
               e.g., cdr(while({λx|car(x)>0},
                              {λx|cons(car(x)−1, car(x)×cdr(x)},
                              (4.1))) = 41

compose(f,g) = composition of functions f and g = (λx|f(gx))
  e.g., compose((λx|3×x), (λx|(4+x)) → (λx|3×(4+x))
```

FIGURE 5-10
Some higher-order functions.

An entirely different kind of higher-order function is *while*. This function takes two function arguments and a third object as an accumu-
lating parameter. It operates much like a *while loop* in a conventional pro-
gramming language. As long as the application of the first function argu-
ment (the loop's iteration test) to the accumulating parameter returns T,
while recursively calls itself with the accumulating parameter modified by
applying it to the second function argument (the loop body). When the
result is F, the accumulating parameter is returned as the value. The ex-
ample in the figure computes the factorial of a number.

The final higher-order function is *compose*. This function takes as its arguments two other functions, and produces as a result a new function which is the composition of the two.

## 5.9 AN EXAMPLE—SYMBOLIC DIFFERENTIATION

As an example of many of the above capabilities, we consider here the problem of writing a function that can take the symbolic derivative of an



```
dx/dx = 1
dy/dx = 0 (y ≠ x)
d(A+B+C+..)/dx = dA/dx + dB/dx + dC/dx + ...
d(A−B)/dx = dA/dx − dB/dx
d(A×B)/dx = (dA/dx)×B + A×dB/dx
d(A/B)/dx = ...
```

FIGURE 5-11
Rules for differentiation.

arbitrary mathematical expression. In addition to demonstrating techniques of expression writing, this example also serves as a good example of an expression where the input and output both could conceivably be treated as "code" of some sort.

Figure 5-11 gives some of the basic rules for differentiating an expression with reference to some "mathematical" variable. The only difference from standard definitions is in the case of "+," which has been extended naturally to cover multiple operands. This is done deliberately to demonstrate the use of a higher-order function.

Although it is possible to describe a function that takes arbitrary infix notation expressions and produces infix outputs, things get quite messy quite fast (we would need to consider parentheses, precedence, etc.). Instead, we use prefix notation and s-expressions to produce a notation that is still readable but much easier to process. Figure 5-12 gives some syntax rules for this format; the meaning should be obvious.

This prefix s-expression format will show up again in the programming language LISP.

An abstract program for such differentiation is given in Figure 5-13. There are two recursive functions, map2s and dif. The latter accepts two arguments: the s-expression *e* to be differentiated, and the symbol *x* for

```
<symbolic-expression> := <number> | <variable>
    | ( <binary> <symbolic-expression> <symbolic-expression> )
    | ( + <symbolic-expression>+ )
    | ( <unary> <symbolic-expression> )
  <binary> := −|×|\|...
  <unary> := −|¬|sqrt|...
Example: (−x + sqrt(2×y) + x×y)/22 becomes
(/ (+ (− x) (sqrt(× 2 y)) (× x y)) 22)
```

FIGURE 5-12
BNF for s-expression form of integer expressions.

457

```
letrec map2s(f,x,y) = list of elements f(z,y) for each element z of list x"
  = if null(x) then nil
    else cons(f(car(x),y), map2s(f, cdr(x), y))

and dif(e,x) =
  if atom(e)
  then if e=x then 1 else 0
  else let op=car(e) in
    if op="+"
    then cons("+", map2s(dif, cdr(e), x))
    else let left=cadr(e) and right=caddr(e) in
      if op="-"
      then triple(op,dif(left,x),dif(right,x))
      else if op="*"
        then triple("+",triple("×",dif(left,x),right),
                        triple("×",left,dif(right,x)))
        else ... expressions for other operators
     where triple(x,y,z) = cons(x,cons(y,cons(z,nil)))

in dif(...)
```

Example:
Input: (infix notation) 3×x + x×y + (4−x)×z
Input: (s-expression notation) (+ (× 3 x) (× x y) (× (− 4 x) z))
Output: (s-expression notation) (+ (+ (× 0 x) (× 3 1)) (+ (× 1 y) (× x 0)) (+ (× (− 0 1) z) ((− 4 x) 0))
Output: (infix notation) (0×x + 3×1) + (1×y + x×0) + ((0 − 1)×z + (4−x)×0)

FIGURE 5-13
Abstract program for symbolic differentiation.

the variable to drive the differentiation. The main function is recursive and has one local function (triple) of its own.

The main body of dif is a set of nested if-then-elses that determine what kind of expression the argument is. As usual, the first leg of the tests is the basis case (is $e$ an atom?) whose T result is a nested test of whether or not that atom is the same as $x$. The else leg of this test covers all the recursive cases where $e$ is not an atom, i.e., an expression with subexpressions that will have to be differentiated in turn. It starts with a let expression that picks off the first element of $e$, which in prefix notation must be the operator name. A series of tests then cover the different operators individually.

The first of these cases is for "+." Here the rule says to add up the derivatives of all the terms of the original sum. The approach chosen uses the recursive higher-order function map2s, which operates like map2 except that the second argument for the nested calls is a *scalar* object that does not change from call to call. The actual arguments given to map2s include dif itself, the list of sum terms to be differentiated, and the derivative symbol. The result is a list to which we add a "+" to recreate the whole result.

After "+," the operators involve exactly two operand expressions. A dual let expressions picks these out of $e$ before proceeding. Then, in each case, the actual expression for the derivative uses the simple nonrecursive local function triple to create the appropriate results from derivatives of the subexpressions.



## 5.10 PROBLEMS

1. Write as an abstract program the predicate free($x$,$e$), which returns true if the identifier $x$ occurs free in the lambda expression $e$.
2. Write an abstract definition for a function subs($y$,$x$,$m$), where $y$ and $m$ are any valid s-expressions and $x$ is an atom representing a variable name. The result of the function should be an s-expression equivalent to $[y/x]m$. Assume that you have available a function newvar(), which at each call returns a new variable name that has not been used before. Your definition should detect and perform renaming properly.
3. Translate any of the abstract programs for reverse into pure lambda notation. Assume only the primitive functions null, cons, car, and cdr.
4. (Project) Write an abstract program that converts an abstract program back to pure lambda calculus. Use abstract syntax functions as required.
5. (Hard Project) Go the other way—from pure lambda calculus to some readable form of abstract programming.
6. What are the values bound to $x$, $y$, and $z$ at each level of the following:

   let $x$=4 in
     let $y$=$x$+2
     and $z$=$x$−3 in
       let $x$=$y$+$x$
       and $y$=$z$+3
       and $z$=$y$+6 in $x$+$y$+$z$

7. Show where to put the Y expression if one wanted to define a single recursive function using just the let expression.
8. Show that Figure 5-7 is correct.
9. In analogy to Figure 5-7 and Figure 5-8, show how to translate to pure lambda a letrec expression of the form "letrec f($x$)=A and g($x$,$y$)=B and $h$($x$,$y$,$z$)=C in E." Discuss how this generalizes to an arbitrary number of mutually recursive functions.
10. Convert the abstract program in Figure 5-13 to pure lambda calculus form. (You need not translate car, cdr, cons, atom, =, or character representations for "+," "-," "*,"....)
11. Write an abstract program to optimize arithmetic expressions constructed according to Figure 5-12. Use optimization rules of the form:

237

"(× 0 A)" is 0
"(× A 1)" is A
"(+ A 0)" is 0
An expression involving only numbers can be reduced to a number.
You may assume that the predicate number($x$) returns true if $x$ is a number and false otherwise.

12. Rewrite reduce to use an accumulating parameter. You may redefine reduce to "parenthesize" the other way if you wish.
13. Define and write in abstract syntax an until function modeled after the while function described in the text.
14. Complete Figure 5-13 to include division and the unary operators "−" and "sqrt."

# CHAPTER 6

# SELF-INTERPRETATION

Lambda calculus is powerful enough to express any computable function, and is thus as powerful as any other computing language. Given this, it is interesting to ask if the execution model of lambda calculus is itself expressible as a computable function. If it is, then we have the possibility of writing in lambda calculus an interpreter for itself. Such an interpreter would express the semantics of lambda calculus in itself (and thus for any language that lambda calculus supports—such as abstract programming).

As hinted at several times, and perhaps partially demonstrated by the differentiator evaluator of the last chapter, the answer to this is a resounding yes. We can write a lambda calculus function, usually called *eval*, which when given any arbitrary lambda expression E, reduces it as far as possible, and returns the result. In particular, since eval(E) (before reduction) is itself a valid lambda expression, we are perfectly free to try eval(eval(E)), that is, have eval determine what happens when eval itself is turned loose on an expression. The answer returned from eval(eval(E)) is the same as that from eval(E), and both are equivalent to the reduced form of E. Eval thus forms a completely valid *interpreter* for lambda calculus, expressible in lambda calculus.

The first few sections in this chapter describe eval using abstract syntax functions for the syntax parsing of generic lambda expressions. Two versions are given, one which performs normal-order evaluations, and one which performs applicative-order evaluations.

After this we will introduce a particular concrete syntax for lambda calculus using s-expressions and prefix notation similar to that for the in-



236

put to the differentiator example of the last chapter. The abstract syntax functions in eval will be rewritten in terms of the appropriate s-expression operators, and eval will be modified as necessary.

There are good reasons for this approach. First, it approaches the syntax of the most popular function-based language, LISP. More important, however, the notation is close enough to something that is implementable on conventional computers to consider using it as a *bootstrap* process to bring up a real function-based programming system. Having once written a machine-language version of this eval function, we can then write a modified eval on top of it which has all sorts of extra features (abstract programming syntax, built-in functions, input/output, error handling, etc.), which in turn could support evals for languages with even more powerful programming features.

Such languages are often called *metalanguages*, and the evals which support them, *metainterpreters*. When given a program in the highest such language, the operation of such a system corresponds to the lowest-level interpreter simulating the execution of the first metainterpreter as it simulates the next metainterpreter simulating the one above it, etc., until the final metainterpreter simulating the given program is reached. Although obviously this is potentially a very slow process, this approach is used frequently in the computer science community to quickly investigate the potentials of new programming languages.

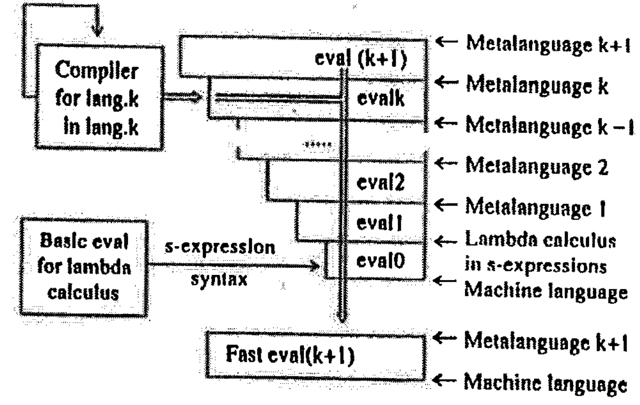
We should note that this cascading of interpreters can be sped up quite easily by picking some particular level of metalanguage and writing a compiler for it in itself. As pictured in Figure 6-1, this compiler can then be fed to the metainterpreter for that language, with a copy of itself used as input to the compiler. The result is a compiled version of the compiler, which now can be used to compile the next-higher metainterpreter (or even a compiler for that metalanguage).

Another reason for investing time in an s-expression form of eval is that it gives us an excellent vehicle for describing orders of evaluation which are combinations and extensions of both pure applicative- and pure normal-order reduction. Such approaches offer some extremely interesting capabilities, and will be investigated in a later chapter.

A related topic in this chapter discusses a variation to the design of such interpreters that packages the information needed for substitutions into *association lists*, and defers actual substitution until the last possible moment. Because of obvious efficiency gains and the match of such structures to objects that can be built into conventional computers, this style of interpreter is used in many real systems for real functional languages, and will be the basis for discussions in later chapters.

Finally, this chapter also discusses how to "package" the process of evaluating an expression in midstream, freeze it, pass it around as an ordinary object, and then restart it at some later time. The package is called a *closure* and forms the basis of quite a bit of the advanced language techniques discussed later in this book. Landin (1963) is one of the earliest references to this subject area. Henderson (1980) is another good reference, particularly Chapter 4.






Let "$P_n$" be a program written in metalanguage n:

$eval_0(eval_1(eval_2(\ldots eval_n(P_n)\ldots) \rightarrow$ result

Assume $compiler_k$ = compiler program for metalanguage k written in metalanguage k.

$eval_0(eval_1(\ldots eval_k(compiler_k(compiler_k)\ldots) = {}^*compiler_k$

Now ${}^*compiler_k(eval_{k+1}) \rightarrow {}^*eval_{k+1}$

.... A machine-language-speed interpreter for metalanguage k+1

**FIGURE 6-1**
Metainterpreters, languages, and compilers.

## 6.1 ABSTRACT INTERPRETERS

The first version of eval interprets pure lambda expressions as defined by the original syntax of Chapter 5. This is the syntax without any of the "simplifications" of multiple arguments, reduced "( )," etc. To make this eval simple, we will use the ***abstract syntax*** functions listed in Figure 6-2. It should be obvious that such functions could be written in many different programming languages without much difficulty but are largely uninteresting.

For historical reasons, our first version of this interpreter will actually consist of three mutually recursive functions: ***eval***, ***apply***, and ***subs***. Eval(E) does as advertised: It reduces the arbitrary lambda expression E as far as possible. Apply($f$,$a$) takes two lambda expressions, $f$ and $a$, and treats $f$ as a function to which $a$ should be given as its argument. Subs($a$,$x$,$e$) performs the substitution of $a$ for all free instances of the identifier $x$ in the expression $e$, (i.e., $[a/x]e$).

Figure 6-3 lists these functions. They are all mutually recursive. As

```
Syntax summary:
  <expression> := <identifier> | <function> | <application>
  <function> := (λ<identifier>"|" <expression>)
  <application> := (<expression> <expression>)

Abstract predicates: Assume E an arbitrary lambda expression
• is-id(E): true if E an identifier
• is-function(E): true if E a lambda function
• is-application(E): true if E an application

Abstract selectors: Assume E a lambda expression
• get-function(E): get function from application E
• get-argument(E): get argument from application E
• get-id(E): get identifier from function E
• get-body(E): get body expression from function E

Abstract creators: Create an expression
• create-function(x,E): create (λx|E)
• create-application(A,B): create (A B)
• new-id( ): return a guaranteed unique identifier symbol

Examples:
• is-id(((λx|(xy))A)) → F
• is-function(((λx|(xy))A)) → F
• is-application(((λx|(xy))A)) → T
• get-argument(get-body(get-function(((λx|(xy))a))))
  → get-argument(get-body((λx|(xy))))
  → get-argument((xy))
  → y
• let z = new-id( ) in
  create-application(create-function(x,((xy)y)), z)
  → create-application((λx|((xy)y)), z)
  → ((λx|((xy)y))z
```

**FIGURE 6-2**
Abstract syntax functions for eval.

discussed previously, there is an implied letrec at the beginning, and ands coupling them.

Eval has three cases corresponding to the three forms of a lambda expression. The first handles identifiers by simply leaving them alone. The second handles expressions that are pure functions by recreating the function, but with its body fully reduced. The final case handles applications by selecting out the function and argument subexpressions and passing them to the function apply.

Apply handles the reduction of applications. As with eval, this function divides into three subcases corresponding to the internal structure of the expression passed as the function. If it is a simple identifier, the only reductions possible are to the argument, and what is returned is the original application put back together again (by the create-application func-



```
eval(e) = "evaluate expression e as far as we can" =
        if is-id(e)
        then e
        else "either a function or application"
            if is-function(e)
            then create-function(get-id(e), eval(get-body(e)))
            else apply(get-function(e), get-argument(e))

apply(f,a) = "normal-order application" =
        if is-id(f)
        then create-application(f,eval(a))
        else "f an application itself or a function"
            if is-application(f)
            then apply(eval(f),a)
            else eval(subs(a, get-id(f),get-body(f)))

apply(f,a) = "applicative-order application" =
        if is-id(f)
        then create-application(f,eval(a))
        else let b=eval(a) in "evaluate argument first"
            if is-application(f)
            then apply(eval(f),b)
            else eval(subs(b, get-id(f), get-body(f)))

subs(a,x,e) = "substitute a for x in e" =
    if is-id(e); see if Rule 1
    then if e=x then a else e
    else if is-application(e); see if Rule 2
        then create-application(subs(a,x, get-function(e)),
                                subs(a,x,get-argument(e)))
        else let y=get-id(e) and c=get-body(e) in
            if y=x then e; Rule 3a
            else; always Rule 3c—rename binding variable
                let z=new-id() in
                    create-function(z, subs(a,x,
                    subs(z,y,c)))
```

**FIGURE 6-3**
Abstract interpreter.

tion), but with the argument evaluated. If the function part $f$ is itself an application, it is evaluated first before performing the application to the argument $a$. Finally, if $f$ is a pure lambda function, then the application is performed by substituting the argument $a$ for the binding variable for $f$ into the body of $f$.

There are two versions given for apply, one for *normal-order reduction* and one for *applicative-order reduction*. The former substitutes the

unevaluated argument into the function body; the latter evaluates the argument first. Note also that in the latter case we could, if we wished, reduce the function body before the substitution.

Figure 6-4 gives an example of a normal-order evaluation by this interpreter of the expression ((T *a*) *b*).

There is one subtle problem in the above version of apply. If *f* is an application of the form (*xy*), where *x* is a simple identifier, then apply will call itself with *f* replaced by eval(*xy*), which will end up returning (after another call to apply) (*xy*) unmodified. Apply will be caught in an infinite loop. It is left up to the reader to describe the relatively simple fixes needed to avoid this.

Finally, the function subs performs the substitution process spelled out in the previous chapter, with one exception. When the body *e* is itself a function and the binding identifier *y* of this nested function is different from *x*, the substitution rule calls for a check if *y* occurs free in *a*, and if so, the renaming rule is invoked. For simplicity, the subs function given here always renames *y*; the function new-id provides a unique new identifier each time it is called. It is left as another exercise to the reader to correct it to rename only when necessary.

```
eval((((λx|(λy|x)) a) b))
→ apply(get-function((((λx|(λy|x)) a) b)),
        get-argument((((λx|(λy|x)) a) b)))
→ apply(((λx|(λy|x)) a), b)
→ apply(eval(((λx|(λy|x)) a)), b)
→ apply(apply((λx|(λy|x)), a), b)
→ apply(eval(subs(a, get-id((λx|(λy|x))), get-body((λx|(λy|x))))), b)
→ apply(eval(subs(a, x, (λy|x))), b)
→ apply(eval((λz|a)), b)
→ apply(create-function(get-id((λz|a)), eval(get-body((λz|a))), b)
→ apply(create-function(z, eval(a)), b)
→ apply(create-function(z, a), b)
→ apply((λz|a), b)
→ eval(subs(b, get-id((λz|a)), get-body((λz|a))))
→ eval(subs(b, z, a))
→ eval(a)
→ a
```

FIGURE 6-4
Example of normal-order eval.



## 6.2 LAMBDA EXPRESSIONS AS S-EXPRESSIONS

The introduction to s-expressions in Chapter 2 emphasized their use for data structures. The symbolic differentiation example earlier in this chapter introduced the idea of using s-expressions in a prefix notation. This section takes the idea one further, and gives lambda calculus a ***concrete syntax*** by showing a conversion between its original syntax and s-expressions. The result is a notation for which we can describe precisely the abstract syntax functions of Figure 6-2 and which begins to approach the actual syntax and interpretative model of LISP.

### 6.2.1 Pure Lambda Calculus and s-Expression Form

Figure 6-5 gives the conversion between the BNF forms of lambda expressions and equivalent s-expressions. It also includes a diagram of the cell representation for a simple example. The result is a notation where a single atom by itself is an identifier, and an expression in parentheses is either a function or an application. In the former case the car of the expression is the keyword lambda; in the latter case it is the expression to be treated as a function, with the cadr of the total expression its argu-

Lambda expression <l-expr> ⟹ s-expression <s-expr>:

<identifier> ⟹ <identifier>

(λ<identifier>|<l-expr>) ⟹ (lambda (<identifier>) <s-expr>)

(<l-expr><l-expr>) ⟹ (<s-expr> <s-expr>)

Example: (((λx| (λy| x)) a) b)
⟶ (((lambda (x) (lambda (y) x)) a) b)

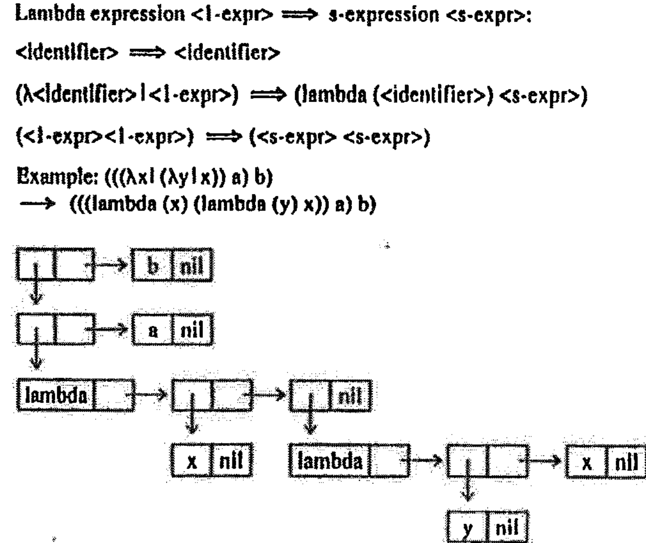



FIGURE 6-5
Lambda expression translation into s-expressions.

ment expression. In either case, looking at the car of the expression tells us immediately what kind of expression it is. This looks like a *prefix notation*, considerably simplifying a real interpreter specification. In fact, replacing the abstract syntax functions of Figure 6-3 by those listed in Figure 6-6(*a*) gives a complete set of functions for a lambda language in s-expressions. Figure 6-6(*b*) gives the converted form of the subs function.

### 6.2.2 Multiple Arguments, Constants, and Built-Ins

The particular conversion process shown above was chosen because it supports cleanly several of the notational simplifications discussed earlier for the original lambda calculus format. First, multiple arguments can be handled both in function definitions and in applications. In the former, the list object following lambda can include as many variable names as there are arguments, as in (λxy|y(xx)) ⇒ (lambda (*x y*) (*y* (*x x*))). Note that each such variable is a separate element in the embedded list (*x y*).

In conjunction with this, an s-expression application can be ex-

```
Abstract function → s-expression equivalent
  is-id(e) → atom(e)
  is-function(e) → eq(car(e),lambda)
  is-application(e) → not(atom(e)) and not(is-function(e))
  get-function(e) → car(e)
  get-argument(e) → cadr(e)
  get-id(e) → caadr(e)
  get-body(e) → caddr(e)
  create-function(id,body) → list(lambda, list(id), body)
  create-application(f,a) → list(f, a)
```

(*a*) s-Expression function equivalents.

```
subs(a,x,e) = "substitute a for x in e—all s-expressions"
  if atom(e); see if Rule 1
  then if eq(e,x) then a else e
  else if not(atom(e)) and not(is-function(e)); see if Rule 2
       then list (subs(a,x, car(e)),
                  subs(a,x,cadr(e)))
       else let y = caadr(e) and c = caddr(e) in
            if y = x then e; Rule 3a
            else; always Rule 3c—rename binding variable
                 let z = new-id() in
                     list(lambda, list(z), subs(a,x, subs(z,y,c)))
```

(*b*) Substitution of s-expressions.

**FIGURE 6-6**
Abstract syntax functions for s-expression notation.



tended to have more than two list elements, with the first representing the function as before and the rest of the list representing the available argument values. The order of the argument expressions in the list matches the order of identifiers in the formal argument list of the function. For example:

(λxy|y(xx)) *w z* ⇒ ((lambda (*x y*) (*y* (*x x*))) *w z*)

Note that a cell representation of this is different from that of Figure 6-5.

Next, numbers, booleans, character strings, etc., can all be used as ***constants*** to convey their standard meaning. In this form, they need only be written in their normal textual form without expansion to the more complex pure lambda expression form.

To go with this, we can expand the allowable types of expressions that can be used in the first element of an application to include common functions such as "+," "×," "and," "or," "car," "cdr," "cons," etc. Then, either eval or apply can be expanded to include specific tests for these ***built-in functions***, and perform the appropriate operations on the argument values. Thus (cons (+ 3 4) nil) should evaluate to the list (7).

Note that in such cases we probably want some sort of applicative-order evaluation to reduce the arguments before applying the function. Consider the complexities resulting if we did not, and were asked to evaluate (× (+ 3 4) (− 8 6)) The code for "×" would have to handle unevaluated applications as arguments—a large impact on performance.

### 6.2.3 Other Special Forms

Just as lambda in the car of a list serves as a keyword indicating a lambda function, we can extend our s-expression form of the language to cover many of the other convenient notations from abstract programming, such as let-and-in, if-then-else, etc. When expressed as an s-expression with a special keyword in the car position of the list, such notation are called ***special forms***. Figure 6-7 summarizes this s-expression syntax, and Figure 6-8 gives a sample expression that includes many of the features.

**let AND letrec FORMS** The let and letrec notations of abstract programming translate to s-expressions by a list of the form:

(let ((identifier)*) ((s-expression)*) (s-expression))

Thus let $x$=A and $y$=B and $z$=C in f($x$, $y$, $z$) is equivalent to

(let (*x y z*) (A B C) (f *x y z*))

The second list element (accessed by cadr on the whole list) follows the multiple-argument notation introduced earlier. The names of all iden-

```
<id-list> := (<identifier>*)

<expr-list> := (<expr>*)

<builtin> := + | - | × | / | car | cdr | cons | atom | null ...

<expr> := <number> | <nil> | <identifier>
        | <expr-list>
        | (lambda <id-list> <expr>)
        | <special-form>

<special-form> := (<builtin> <expr>*)
               | (let <id-list> <expr-list> <expr>)
               | (letrec <id-list> <expr-list> <expr>)
               | (if <expr> <expr> <expr>)
               | (cond (<expr> <expr>)*)
               | (quote <expr>)
```

**FIGURE 6-7**
s-Expression form of abstract program syntax.

```
letrec fact(n) =
  if n=0
  then 1
  else let m=n-1 in n×fact(m)
and sum(n,a) =
  if n=0
  then a
  else sum(n-1,n+a)
in sum(fact(3),0)
```

(*a*) Abstract form.

```
(letrec (fact sum)
      ( (lambda
         (n)
         (cond ((= n 0) 1)
                (T (let (m) ((- n 1)) (× n (fact m))))))
        (lambda
         (n a)
         (if (zero n )
           a
           (sum (- n 1) (+ n a))))
 (sum (fact 3 0)))
```

(*b*) s-Expression form.

**FIGURE 6-8**
Sample conversion to s-expression form.



tifiers being bound to values are included here as elements of a list. Thus, the first element in this sublist is the variable name after the let; the following identifiers follow any coupled ands.

The third list element here (accessed by a caddr) is the list of expressions to be substituted for the list of identifiers (cadr). The length of the list of expressions and identifiers should match. The expressions themselves are the transformed right-hand sides of the let definitions.

The final element of the original list (accessed by cadddr) is the expression into which all these variables are substituted. This corresponds to the body expression following the in in the abstract program form.

To handle this special form, either eval or apply (usually eval in this case) can be modified. It will test the car of a list for let and perform the appropriate substitutions. Either applicative- or normal-order evaluations of the arguments are possible.

**if-then-else AND cond FORMS** Abstract program forms of the if-then-else sort can be included as s-expressions by special forms of the form

(if ⟨s-expression⟩ ⟨s-expression⟩ ⟨s-expression⟩)

where if is a keyword detected by eval, and the first expression is evaluated for a true or false value. In the former case, the second expression (caddr) is then passed to eval. In the latter, the third (cadddr) is so handled.

Although either applicative- or normal-order evaluation is possible here, normal order makes more sense. We evaluate the then or else expressions only after evaluating and testing the expression following the if.

A generalization of this form neatly handles nested if expressions of the form if...then...elseif...then...elseif...then...else.... Here the car is cond (for *conditional*), and the rest of the list is two-element lists:

(cond (⟨s-expression⟩ ⟨s-expression⟩)*)

The interpretation of this is that when eval finds an expression with cond as its car, the evaluation process involves taking the first element of the remaining list and evaluating the car of this list. If the result is true, then the cadr of this element is evaluated and returned. If the result is false, the next element of the original expression is evaluated.

If none of the pairs has a car that evaluates to T, we will assume that nil is returned. To avoid this default, the last pair of expressions should have its first expression equaling T to force evaluation of the second. This corresponds to the last else case in the chain.

An alternative form of the cond that is sometimes used deletes the T in the last expression, leaving something of the form (⟨expression⟩). If none of the pairs in front of it is true, the embedded expression in such a last term is always evaluated and its value returned.

242

**STOPPING INTERPRETATION—QUOTE** There are times when a programmer wants to prevent an expression from being evaluated. The expression is to be treated as data, and not as a piece of program text to be reduced. This is particularly common in languages where the syntax for programs looks like s-expressions, with little or no distinction between program and data.

A typical example of this is in the writing of one of the interpreters described in this chapter in real code. In such programs there are many tests for equality between some evaluated expression and the symbolic form of program keywords, such as in if *e*=lambda, or in expressions to test if a function is a "built in," as in member(*e*, (cons car cdr...)). Evaluating the latter with a direct extension of the interpreters to date might reduce (cons car cdr...) to (car cdr) or worse. This is not at all what is intended.

The popular solution to such problems is to introduce a special function *quote*, which when evaluated returns its single argument totally unevaluated (i.e., neither normal nor applicative evaluation). Thus (if (eq *e (quote lambda))...) or* (member *x* (quote (cons car cdr...) would work as desired.

## 6.3 AN EXPANDED INTERPRETER

Figures 6-9 and 6-10 give a version of eval that assumes the syntax of Figure 6-7 and thus has all of the following features:

- Interpreted language is in s-expression format.
- Functions may have multiple arguments.
- Support for "built-in" functions such as "+," car, etc.
- A mix of normal- and applicative-order reduction.
- Support for special forms such as let, letrec, if, cond, etc.

The result is that this new eval supports a function-based language with semantics that is very close to pure lambda calculus, but with the syntatic sugar of abstract programming, a representation that permits easy implementation on conventional computers, and efficiency hooks that speed up certain standard calculations.

Both normal- and applicative-order reduction are used in this interpreter. The typical processing of lambda expressions is via normal order. This greatly simplifies the handling of recursion. Applicative order, however, is used in several places, particularly when dealing with "built-in" functions such as "+," "car,"....The arguments to such functions are reduced fully before reaching the function. While this increases performance, it does mean that we cannot curry built-in functions—all arguments to them must reduce to the appropriate type of object.

The s-expression notation used here is the same as that described in the last section, with one limitation. For simplicity, expressions with prefix of letrec are permitted to make exactly one local definition. This is to



```
eval(e) = "reduce s-expression e"
  if atom(e)
  then e "Nonlists are fully reduced"
  else let fcn=car(e) and args=cdr(e) in
       if atom(fcn)
       then "function term is built-in or special form"
            if fcn=quote then car(args)
            elseif member(fcn, builtins)
            then apply-builtin(fcn, args)
            elseif fcn=lambda "look for special forms"
            then list(lambda, car(args), eval(cadr(args)))
            elseif fcn=if then "see which expr to eval"
                   if eval(car(args))=T
                   then eval(cadr(args))
                   else eval(caddr(args))
            elseif fcn=cond then eval-cond(args)
            elseif (fcn=let) or (fcn=letrec)
            then let ids=car(args) "list of identifiers"
                 and vals=cadr(args) "matching value list"
                 and body=caddr(args) in
                 if fcn="let"
                 then subs2(vals, ids, body)
                 else "a letrec expression"
                   apply(list(lambda,ids,body),
                         list(Y, list(lambda, ids, vals))
                   where Y=(lambda (y) ((lambda (x) (y(xx)))
                                        (lambda (x) (y(xx))))
            else "must be an identifier"
              apply(fcn, args)
       else apply(fcn ,args) "function is a list"

apply(f,a) = "a normal-order application"
  if atom(f)
  then cons(f,map(eval,a))
  elseif car(f)=lambda
  then "f is a lambda expr—see if curry"
       let formal=length(cadr(f))
       and actual=length(a) in
           if formal=actual
           then eval(subs2(a, cadr(f),caddr(f)))
           else list(lambda,
                     drop(actual, cadr(f)),
                     eval(subs2(a, cadr(f),caddr(f))))
                whererec drop(count, x) =
                if count=0 then x else drop(count-1,cdr(x))
  else apply(f, a)
```

**FIGURE 6-9**
An expanded interpreter.

242

```
apply-builtin(f,a) = "an applicative-order evaluation"
  let args = map(eval, a) in "args = list of evaluated arguments"
     if f = "car" then caar(args)
     elseif f = "cdr" then cdar(args)
     elseif f = "+" then car(args) + cadr(args)
     elseif ...

eval-cond(a) = "special evaluation for cond"
  if null(a) then nil
  else "iterate thru the list of pairs"
       let z = car(a) in
           if eval(car(z)) = T
           then eval(cadr(z))
           else eval-cond(cdr(a))

subs2(v,n,e) = "substitute v's for n's in e"
  if atom(e) "if e variable, replace"
  then lookup(e,n,v)
  elseif car(e) = lambda
  then "substitute into a lambda expression—rename"
       let old = cadr(e) in "cadr = id list"
           let new = map(new-id,old) in
           list(lambda,new, subs2(v, n, subs2(new,old,caddr(e))))
  else "e is an application—substitute in all parts"
  map((λz|subs2(v,n,z)), e)

lookup(z,n,v) = "find z in n and replace by value in v"
  if null(n) or null(v)
  then z
  elseif car(n) = z then car(v)
  else lookup(z,cdr(n),cdr(v))
```

FIGURE 6-10
Support functions for expanded interpretations.

avoid for now the complications described earlier for mutually recursive functions.

The major differences between Figure 6-9 and Figure 6-3 are in the handling of multiple arguments and in applications involving *special forms* (where "keywords" are prefixes). Syntatically, multiple actual arguments for a standard lambda application are represented as multiple elements in an s-expression list following the prefix element [a function of the form "(lambda (...) (...))"]. They are handled by expanding the subs function into subs2, which "simultaneously" replaces all the occurrences of identifiers from the lambda function by matching value expressions from the application list. Internally, subs2 recursively processes the lambda function's body expression one element at a time, using the function *lookup* each time an element which is a variable is found.



Currying is detected when the number of actual arguments is less that the number of formal identifiers in the lambda expression. In this case, only those identifiers with matching variables are replaced, and the rest are left as identifiers for a new lambda expression.

Special forms are handled by special if-then-elseif tests in eval. "Built-in" functions are handled as described above. All the arguments are evaluated, and then code corresponding to the function is executed.

For the keyword prefix lambda, the evaluation process simply evaluates the body of the function. Note that we are evaluating this body only if a request was made to evaluate the function to begin with. This is in tune with the rules of normal-order reduction.

For if, the evaluation procedure takes the first argument, the test expression, and evaluates it. On the basis of the resulting value, either the second or third argument is evaluated. This is a mix of applicative- and normal-order evaluation.

Evaluation of cond is handled similarly. The arguments in this case are pairs of expressions, the car of which are evaluated sequentially until one is found whose value is true. Then its matching cdr is evaluated. As with built-ins and if, these car tests must be fully evaluatable.

Expressions containing let are translated exactly as the abstract-language version was translated earlier in this chapter. A new function expression is created, with arguments represented by the list in the car of the let expression. This function is passed to apply along with the second element of the argument list, which itself is a list of argument values for the specified identifiers.

Evaluation of letrec is similar but more complex. For simplicity, the version shown in Figure 6-9 handles properly letrecs with at most one identifier given a recursive definition. It also assumes that this definition is itself a lambda expression (as converted into s-expression notation). As with let, the processing generates a new function expression (again in s-expression notation) and passes it to apply. In this case, however, the argument expression for apply is taken apart and put back together with an extra identifier. This is the name for the recursive function. It is then made into an application with a prefixed Y expression. Again, this is exactly as described for abstract programs.

Note that if we had wanted to, we could have added Y as a special form to eval which performed the same operations without the substitutions.

Figure 6-11 gives a partial trace of the evaluation of the factorial function. Most of the major parts of eval and apply are executed here.

## 6.4 ASSOCIATION LISTS

All the lambda interpreters discussed up to now have employed brute-force substitution to handle applications. The argument expression (either evaluated or unevaluated) is substituted in total for every free occurrence of the binding variable in the function's body. This requires

242

```
eval(((lambda (x y) (+ x (× 4 x))) (+ 3 5) 6))
  → apply((lambda (x y) (+ x (× y x))), ((+ 3 5) 6))
  → eval(subs2(((+ 3 5) 6), (x y), (+ x (× y x))))
  → eval(map((λz|subs2(((+ 3 5) 6), (x y), z), (+ x (× y x))))
  → eval((+ (+ 3 5) (× 6 (+ 3 5))))
  → apply-builtin(+, ((+ 3 5) (× 6 (+ 3 5))))
  → let args = map(eval,((+ 3 5) (× 6 (+ 3 5)))) in ...
  → let args = list(eval((+ 3 5)), eval((× 6 (+ 3 5)))) in ...
  → ...
  → let args = (8 48) in ...
  → car((8 48)) + cadr((8 48))
  → 8 + 48 → 56
```

(*a*) A simple example.

```
eval( (letrec (f) {lambda (f) (lambda (n) (cond (zero n) (T (× n (f (− n 1))))))}) (f 3)))
  → apply( (lambda (f) (f 3)), (Y (lambda (f n) (cond ...))))
  → eval(subs2((f 3), (f), (Y ...)))
  → eval(((Y ...) 3))
  → ...
  → eval(({lambda (f n) (cond ...)} {Y...} 3))
  → apply(eval({lambda (f n) (cond ...)}), ({Y...} 3))
  → apply({lambda (f n) (cond ...)}, ({Y...} 3))
  → eval(subs2((cond...), (f n), ({Y...} 3)))
  → eval((cond ((zero 3) 1) (T ...)))
  → eval-cond({((zero 3) 1) (T ...)})
  → if apply-builtin(zero 3) then 1 else ...
  → ...
```

(*b*) A version of factorial.

**FIGURE 6-11**
Sample partial interpretations.

essentially scanning a function's body twice, once to find all the free occurrences and do the substitutions, and once to find the next application to attempt.

While conceptually simple, such an approach suffers from obvious inefficiencies when the function's body contains (as most do) one or more nested if-then-else structures, or when there are multiple arguments. More passes may be needed (up to two per argument), and part, if not most, of the resulting substitutions may be wasted effort (due to then-else cases that are not used).

One alternative to such substitutions is simply to remember at the time of an application which identifiers are to get which values, and then do the substitution on an identifier by identifier basis when the function's body is scanned for the next application. For example, in the application

(λ*wxyz*|E) W X Y Z



we wish to remember the four-way simultaneous substitution [W/*w*,X/*x*,Y/*y*,Z/*z*]. This pairing is often called the ***context***, ***binding***, or ***environment*** under which the expression E is to be evaluated or reduced.

The easiest data structure in which to record such a substitution is an ***association list***, or ***alist***, which is created at the time an application is encountered and which, in some way, pairs up identifier names and the values assigned to them via applications. Such a data structure is passed as required between apply and eval when various parts of the function's body are actually evaluated. Finding an identifier in such an expression should cause eval to look up the identifier in the alist and replace it by the identifier's matching value. Thus both eval and apply must have their definitions extended to include extra arguments that pass these alists.

There are two major forms that an alist for a single application typically takes. First, it can be a single s-expression list whose elements are consed pairs of names and matching values. Note that the order of each pair is backward from that of the "[/]" notation; the reason is efficiency in many real implementations.

For the previous example, the s-expression form of the alist would be of the form ((*w*.W) (*x*.X) (*y*.Y) (*z*.Z)).

Second, the alist can be two lists of the same length, one for the identifier names and one for the matching argument values, such as (*w x y z*) and (W X Y Z), respectively. This resembles usage of the *n* and *v* arguments in earlier applys. The former list is often called the ***name list*** and the latter, the ***value list***. Figure 6-12 diagrams examples of both notations.

Although conceptually simple, the use of either form does require solutions to several problems, including:

1. Multiple formal arguments in a lambda function
2. Nesting of applications
3. Recursion
4. Free variables in either the function body or the arguments
5. Handling of name clashes and renaming
6. Normal-order reduction versus applicative-order reduction

Assume: (let (x y z) (1 2 (/ 9 3)) (× (+ x y) z))

When evaluating (× (+ x y) z):

- alist = ((x.1) (y.2) (z.3))
- name list = (x y z)
- value list = (1 2 3)

**FIGURE 6-12**
Sample notations for association lists.

332

Solving the first problem was discussed above. Multiple name-value pairs in the alist match precisely the multiple simultaneous substitutions that correspond to a function applied to multiple arguments.

The second problem, nesting of multiple applications, can also be handled fairly easily by augmenting the alist. Again two approaches are possible. The first simply appends the new pairs onto the front of the prior alist. Searching for a substitution involves scanning down this list until a pair is found where the name matches the identifier in the expression. While the approach is simple, it does lose insight into which application generated which substitution.

The second approach makes the alist a list of lists, each of which in turn corresponds to the substitutions called out by a single application. Each nested application then stacks a new list onto the front of the old one. Locating a value thus involves looking backward through this list of lists, one sublist at a time. The first occurrence of the desired identifier gives the appropriate value.

### 6.4.1 A Simple Associative List-Based Interpreter

The other problems are somewhat more difficult to understand and are best demonstrated by poking holes in a simple interpreter that uses alists as described above. Figure 6-13 diagrams such an eval and apply. They assume only the pure lambda calculus (in s-expression format) as inputs; only one argument (no multiple arguments), no built-in functions, constants, or special forms are assumed. Both functions resemble those from Figure 6-3 but do not use the subs function or its equivalent. Instead they both have an extra argument representing pairs of names and values for all the applications still in force at the current point in the evaluation process. Thus when eval encounters an identifier, it calls a function *assoc*, which will look through the alist for the identifier. If the identifier is not the car of any pair, this function returns nil. If it is there, it returns the pair of name and value. (This permits us to distinguish between an identifier that is not in an alist and one that is, but bound to nil).

When eval receives a match from assoc, it takes the value (the cdr part of the pair) and reevaluates it. This is to take care of circumstances where there is a chain of substitutions. (Consider, for example, the alist ((*z*.1) (*q*.2) (*y*.(+ *z q*)) (*x*.(× *y z*))) when the value of *x* is desired).

Apply evaluates applications by building a new (name.value) pair, appending it to the front of the current alist, and then passing the body of the function back to eval with the augmented list as the new context.

There are some severe problems with this simple-looking set of functions, many of which revolve around situations where the body of a function itself contains a function.

The first example of this is the simple expression

((lambda (*y*) (lambda (*x*) (+ *x y*)) 4)



```
<lambda-expr> := <identifier>
               | (lambda (<identifier>) <lambda-expr>)
               | (<lambda-expr> <lambda-expr>)

eval(e,alist) = "evaluate e with context alist"
  if atom(e)
  then let z=assoc(e,alist) in
          if null(z)
          then e
          else eval(cdr(z), alist)
  elseif car(e)=lambda
  then e
  else apply(car(e),eval(cadr(e),alist),alist)

apply(f,a,alist) = "apply f to argument a with context alist"
  if atom(f) "replace a "variable" function by its value"
  then let z=assoc(a,alist) in
          if null(z)
          then cons(f, a)
          else apply(cdr(z),a,alist)
  elseif car(f)=lambda
  then eval(caddr(f), ((caadr(f).a).alist)
  else apply(eval(f,alist),a,alist )

assoc(a,alist) = "find (a."value") in alist"
  if null(alist) then nil
  elseif caar(alist)=a then car(alist)
  else assoc(a,cdr(alist))
```

**FIGURE 6-13**
A simple interpreter using association lists.

The result should be (lambda (*x*) (+ *x* 4)). Eval, however, will pass control to apply, which in turn will call

eval((lambda (*x*) (+ *x y*)), ((*y*.4)))

This look good, but eval, as stated above, will return as its result its first argument unmodified (the argument is a lambda expression). The context ((*y*.4)) is lost.

One solution would be to replace the elseif line of eval by

elseif car(*e*)=lambda then list(lambda, cadr(*e*), (eval(caddr(*e*),*alist*))

This evaluates the body of the function, and would replace the *y* above by the appropriate 4, but would have its own problems. The expression eval(((lambda (*y*) (lambda (*y*) (*y y*))) 4), nil) would result in (lambda (*y*) (4 4)). This time we should have avoided the substitution because of the name clash.

132

Again there is a direct solution to this, namely, rename the inner function's binding variable. This renaming can be done by augmenting the alist used to evaluate the body of a lambda by a new pair that substitutes a unique identifier for the binding variable. Figure 6-14 diagrams such an approach, again for a case where the renaming is always done.

The problem with this approach is that we have essentially done much of the work we were trying to avoid by creating an association list; we end up going through an expression and doing brute-force substitution. A later section will introduce a solution to both this problem and several others. The reader should, however, remember the source of the problem, because it will show up in somewhat different circumstances later as the *funarg problem*.

### 6.4.2 Multiple Arguments

Multiple arguments are a feature that we definitely want to handle in real systems. A solution mentioned above involves making an association list into a list of lists, where each list element is actually the substitution list for each function application. Thus it is the list of binding variables and their matching actual arguments. The changes to make this happen in the above interpreter are small. First, the last line of eval becomes:

**else apply(car($e$), map(($\lambda x$|eval($x$,*alist*)),cdr($e$))**

This applies the function eval to all the argument expressions in the original s-expression other than the first, with the same context represented by *alist*, and gathered into a new list. This list is then passed on to apply as the arguments to the function.

The second change is to the second-to-last line of apply:

**then eval(caddr($f$), cons(map2(cons,cadr($f$),$a$),*alist*))**

Here the list of formal argument names (cadr($f$)) is paired, via a map2

```
eval(e,alist) =
  if atom(e)
  then let z = assoc(e,alist) in
        if null(z) then e else eval(cdr(z), alist)
  elseif car(e) = lambda
  then let z = newid( ) in
        list(lambda,list(z),
             eval(caddr(e),((cadr(e).z) .alist))
  else apply(car(e),eval(cadr(e),alist),alist)
```

**FIGURE 6-14**
A revised eval.



function, with the actual argument values and appended to the front of the *alist*. The result is then fed back to eval as the new *alist*.

### 6.4.3 Recursion

Now consider what happens when this interpreter is used on an application with recursive properties. If the function is some sort of "built-in" function where the implementation handles the recursion directly, then the process described above works well and in fact mirrors the frame-building process used in many conventional languages for recursive procedures. The problem, however, comes up when the recursion is explicit, as in YR, where Y is the *fixed-point combinator* discussed earlier and R is some recursive function. The applicative-order evaluation shown here blows up, and wanders off to infinity computing $R(YR) \Rightarrow R(R\ldots(YR)) \Rightarrow \ldots$.

Matters get even worse when a set of mutually recursive functions are desired, as in

**letrec $f$=F and $g$=G and $h$=H in E**

Here the context (association list) for E is of the form

**((($f$.F1) ($g$.G1) ($h$.H1))...)**

However, F1 (the value for $f$) (and likewise for G1 and H1) requires knowing the values for $f$, $g$, and $h$ before they are computed. But this is the context for E. We need to know the association list for E before we can compute the association list for E.

The solution for this problem involves new mechanisms to be discussed in the next section.

## 6.5 CLOSURES

Most of the problems of the last section come from trying to improve the "efficiency" of a basic interpreter through a combination of applicative-order reduction and the use of simple association lists in place of immediate substitutions. These proved not to provide the full efficiency gain hoped for (examples where name clashes might occur), and were relatively inadequate for general recursion.

A better solution to both problems involves "packaging" an expression with its environment into a single unit which can be passed around at will, but still be unpackaged and evaluated when needed. Such a package is called a *closure* and is something that not only makes it possible to consider using alists in interpreter descriptions, but also introduces some very novel ideas that permit opportunities for parallelism and for the easy expression of essentially infinite objects. Later sections ad-

dress the latter opportunities; the rest of this section and the next address the former.

Consider an application (λx|E)A (or the equivalent, let $x$=A in E). Its evaluation involves the substitution [A/$x$]E. Assuming for the time being that A has no free identifiers, let us suspend this evaluation just before the substitution takes place. What we have is the expression E and the ***environment*** $x$=A [or association list (($x$.A))]. This combination is called a ***closure*** and for this text will be written as

⟨closure⟩ := [⟨expression⟩,⟨environment⟩]

where the ⟨environment⟩ is expressed as an s-expression association list, and the expression is in whatever format seems convenient at the time. The term ***context*** is often used as an alternative for environment.

For example, expressing "let $x$=3 and $y$=5 in 2×$x$+$y$" as a closure results in [(2×$x$+$y$), (($x$.3)($y$.5))].

Evaluating a closure consists of restarting the substitution, that is, using the environment part as a source for values for the free variables in the expression part.

Nesting of closures is not only possible, but necessary. Consider, for example, the expression

let $x$=A in let $x$=$x$+1 in E

This is equivalent to the nested closure [[E,(($x$.$x$+1))], (($x$.A))]. In such cases the evaluation of a closure must itself be recursive. To evaluate the outer closure we must first evaluate the inner closure. This evaluation involves replacing all the free occurrences of $x$ in E by $x$+1, where the new $x$ is the one in the outer closure's environment. In general, if an expression inside a closure is itself a closure, the outer one must be suspended while the inner one is worked.

The modifications to eval to handle closures are direct. Eval has two arguments as before, the expression to be evaluated and the alist. If the expression is an identifier, we look it up in the alist. If it is a closure, we may call eval recursively, as in eval(eval(get-expression($e$), get-environment($e$)),*alist*). If it is an application, several alternatives are possible depending on the type of reduction sequence desired. In many cases (to be discussed later) we may simply want to form and return a new closure. In other cases we may evaluate it.

Figure 6-15 diagrams a simple case where we will always build a closure upon finding an application, and expand the closure only when its value is needed. In this case a single evaluation of an application returns a closure, and a second evaluation is needed to unpack the closure. To make this fully evaluate an expression we need an outer function which will apply eval repeatedly until an expression has no closures in it of value to a user.



```
eval(e,alist) =
  if is-a-identifier(e)
  then let z = assoc(e,alist) in
        if null(z) then e else cdr(z)
  elseif is-a-closure(e) "evaluate closure here ****"
  then let e1 = get-expression(e)
        and alist1 = get-environment(e) in
        eval(eval(e1,alist1),alist) (nest evaluations)
  else ... as before...
apply(f,a) = ... as before...
  elseif is-a-function(f)
  then create-closure(get-body(f),
                      create-alist(get-identifier(f),a))
  else ... as before..
Example: let x=3 and y=2 in let x=x+y in x×y
= eval([[x×y,((x.x+y))],((x.3)(y.2))], nil)
→ eval(eval([x×y,((x.x+y))],((x.3)(y.2))),nil)
→ eval(eval(eval(x×y,((x. x+y))), ((x.3)(y.2))),nil)
→ eval(eval((x+y)×y, ((x.3)(y.2))),nil)
→ eval((3+2)×2, nil) = 10
```

**FIGURE 6-15**
Basic closure evaluation.

The key point about this process is that, if formed correctly, a closure is entirely equivalent to the original expression, and it can be evaluated (unsuspended) at any time and still get the equivalent normal-form answer. In a sense it is a closed universe, complete in itself, whose ultimate value never changes. In prior terminology, it is ***referentially transparent***, since its value is the same whenever it is evaluated. This will permit systems described in later chapters to do the minimal processing necessary to return something to a user but still be capable of recreating the full answer at any time. This will be particularly useful when dealing with very large, even infinite, lists, where the embedded expression describes how to compute the next element.

## 6.6 RECURSIVE CLOSURES

The above implementation of closures handles nested applications, free variables, and normal- and applicative-order reductions relatively easily. It does not, however, handle recursion. Consider, for example, the differences between "let $f$=A in E" and "letrec $f$=A in E." In the former, any free instances of $f$ in A refer to $f$'s bound in expressions surrounding this one. In the latter, any free instances of $f$ in A should refer to the whole value of A as is. Even worse, those references to $f$ in the A being substituted for

E72

*f* in A must also be replaced, as must the references to *f* in that replacement. There is nothing to stop this substitution from going on forever.

In terms of the substitution notation we have used to date, what we want is something like:

$$[A/f]E=[([A/f]A)/f]E=[([([\ldots)/f]A)/f]A)/f]E$$

The beauty of using a closure to handle this is that it can delay any required substitution, particularly these recursive ones, until they are absolutely needed, and then perform only a minimal amount of substitution necessary to satisfy the immediate evaluation. Keeping all or part of the closure around will permit later recursions as necessary.

The problem with expressing this as a closure is getting an alist entry which has some sort of internal references to itself. As a first cut, try which has some sort of internal references to itself. As a first cut, what we want is a closure of the form [E,((*f*."value for *f*"))], where the "value for *f*" is itself a closure of the form [A,((*f*."value for *f*"))]. This "value for *f*" is a nontrivial object since it requires some sort of reference to itself in itself. That is, if we let α stand for the "value for *f*," then

$$\alpha = [A,((f.\text{"value for } f\text{"}))]=[A,((f.\alpha))]$$

The context for the closure α includes the complete closure α itself! Figure 6-16 diagrams this self-reference.

The expansion to and forms of letrec is also worth discussing. Consider the expression:

letrec $f_1=A_1$ and...and $f_n=A_n$ in E

When converted to pure lambda expressions, the equivalent of this statement gets quite complex. However, if expressed in terms of closures and a self-interpreter, the process is much more understandable. The closure for the whole expression is of the form:

[E, (($f_1$."closure for $f_1$")...($f_n$."closure for $f_n$"))]

where the "closure for $f_j$" is a closure,

[$A_j$, (($f_1$."closure for $f_1$")...($f_n$."closure for $f_n$"))]

Expression: letrec f=A in E

Closure for f = α = [A, ((f.α))]

Entire closure = [E, ((f.α))]

**FIGURE 6-16**
Value in a recursive closure.



The contexts for all these internal closures is the same. If we denote this context by β, we get that the closure for the whole expression is

[E,β] where β=(...($f_j$."context for $f_j$")...)=(...($f_j$.β)...)

Figure 6-17 diagrams this arrangement.

If we implement such data structures as s-expressions, we find that the cdrs of cells containing such contexts point back to the cars of that cell or some earlier cell linked to it. This will be discussed in more detail for the SECD Machine.

## 6.7 CLOSING THE LOOP—READ-EVAL-PRINT

If we were being totally rigorous in our description of these interpreters, they would take the form:

letrec eval=...and apply=...and...in eval(E)

where E is some expression that we want interpreted. While fine for one-of-a-kind uses, this is hardly of general-purpose utility. We must rewrite at least part of this overall expression each time we want to interpret a new expression.

A more useful approach is to change the final "in eval(E)" into something which repeatedly:

1. Reads in an expression to be evaluated from some input device, such as a terminal
2. Evaluates the expression
3. Writes the expression out, for example, back to the terminal's screen
4. Repeats the process for a new input expression

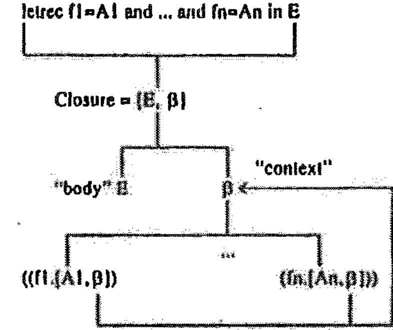



**FIGURE 6-17**
A fully recursive context.

Most real functional-language interpreters work this way. The code that controls this loop is often termed a ***read-eval-print loop***, a ***listener***, or a ***waiter***, and looks something like:

**In listener($e$) whererec listener($e$)=listener(write(eval(read())))**

where **read** is a function (of no arguments) which returns an s-expression to be evaluated from the input device, and **write** prints out its argument (again an s-expression) on some output device. For convenience, we can assume that **write**($e$) returns $e$ as its functional output.

If our interpreter will not handle functions of no arguments, we can add a dummy argument to **read** as shown above and then simply ignore it.

Note also that **listener** is tail-recursive. This means that an efficient implementation of it could avoid stacking arguments or return information from any kind of stack. This is important if we try to implement it on a machine with finite memory.

Note that the above definition of **listener** never completes—as soon as **write(eval(read()))** completes, **listener** starts up another go-around. The argument for **listener** is simply there to provide a place to put **write(eval(read()))**; the value it retains is never used.

## 6.8 PROBLEMS

1. Trace out the evaluation of the following expressions using the interpreter of Figure 6-3. Use both normal- and applicative-order apply. Show the arguments to all calls (except that you may skip internally recursive calls to **subs** or calls to functions whose values are obvious, like **get-identifier**(λx|...)).
   a. (((λx|(λy|(yx)))a)b)
   b. (λx|(λy|(λz|(y((xy)z)))))(λz|z)
   c. ((λx|(λy|y))) ((λx|(xx))(λx|(xx)))) w
2. Convert each of the above to an s-expression form.
3. Repair Figure 6-4 so that it does not get caught in an infinite recursion when called from something like **eval**(($xy$)).
4. Modify **subs** in Figure 6-4 to rename lambda variables only when necessary.
5. Draw out the cell representation for the s-expression (lambda ($x$ $y$) ($y$ ($x$ $x$))) $w$ $z$) from Section 6.2.2.
6. Convert the pure lambda expressions for addition and multiplication to the s-expression form of Figure 6-5.
7. Convert the following version of **member** to the s-expression form of Figure 6-5. Assume built-in functions **null**, **eq**, **car**, **cdr**.

   **member**($x$,$s$)=**if null**($s$) **then** F **else if eq**($x$,**car**($s$)) **then** T **else member**($x$,**cdr**($s$))
8. Assume the following syntax for a certain class of s-expressions:

```
⟨logic-expr⟩ := ⟨id⟩ | T | F
              | (AND ⟨logic-expr⟩ ⟨logic-expr⟩)
              | (OR ⟨logic-expr⟩ ⟨logic-expr⟩)
              | (NOT ⟨logic-expr⟩)
              | (LET ⟨id⟩ ⟨logic-expr⟩ ⟨logic-expr⟩)
```

   Define in abstract syntax a function that will evaluate such expressions, assuming that it is given as input an association list of all identifiers and their current values. Show that it works for the expression:

   (LET $x$ F (LET $y$ T (AND (NOT $x$) (OR $x$ $y$))))
9. Write an abstract program that represents the curried form of each of the following functions. Then show what is returned when this form is applied to the specified argument. Remember that the curried form of a function f($x$, $y$) is a function f′(z) such that for any A and B, (f′(A))(B)=f(A, B). (*Hint:* Look at what Figure 6-9 would do.)
   a. Ackerman's function, argument=1.
   b. Append, argument=(1).
10. Modify Figure 6-15 to handle an extension to **cond** such that if the last element of **cond**'s list has only one expression in it, and no prior test passes, the value of this expression is returned as the value of the **cond**. For example, (cond ((= 1 2) F) (() 4 5) F) ((+ 1 2))) would return 3.
11. Rewrite Figure 6-9 to employ a special form for the Y combinator to handle recursion. This special form would have syntax (Y ⟨lambda-function⟩).
12. Write a function **evaluate** that calls **eval** of Figure 6-15 as often as required to fully reduce an expression so that there are no embedded closures. Show that this interpreter works on (member 1 (quote (2 1))).
13. Write s-expression forms of functions **read**() and **write**($e$) that read and write arbitrary s-expressions. Assume that the only input/output (I/O) built-ins you have are **read-atom**(), which returns the next atom from the input, and **write-atom**($e$), which writes one to the output. Thus a loop on **read-atom** where the input device has "(cons 1 2)" would return successively "(," cons, "1," "2," and ")." Sending these back to **write-atom** would print the same expression back.
14. Rewrite the applicative order form of Figure 6-3 as a single s-expression in the format of Figure 6-5, including a read-eval-print loop as the expression to be evaluated. Assume that you are provided with functions **read** and **write** to handle complete s-expressions I/O.
15. Write the expanded interpreter of Figure 6-9 and Figure 6-15 in the syntax of Figure 6-5, again with a read-eval-print loop. Is this interpreter capable of interpreting itself? If not, what is missing?

# CHAPTER 7

# THE SECD ABSTRACT MACHINE

So far we have described two simple functional languages based on lambda calculus: abstract programming and a prefix s-expression equivalent. The operation of these languages has been described in terms of interpreters for such languages (written in themselves). This corresponds to the general concept of ***denotational semantics***.

An alternative approach to describing the semantics of such languages is through ***interpretative semantics***, where we define a simple ***abstract machine*** and combine this with a description of how a compiler would take programs written in the language of interest and generate matching code for the abstract machine.

This chapter takes such an approach. We will define the ***SECD Machine*** as an abstract machine with properties that are well suited to functional languages, and will give a simple compiler for the prefix s-expression form discussed in the last chapter. The purpose for doing this is twofold. First, it should reinforce the reader's understanding of how functional languages work. Second, it serves as a very clean departure point for discussing those hardware architectural concepts that show up in real machines for real functional languages.

In terms of organization, the first section discusses briefly a very simple form of the memory model used by the SECD Machine, namely, a list-oriented one. The next chapter will expand on it in great detail, but much of that is not needed to understand the rest of the SECD architecture.





This is followed by descriptions of how the SECD Machine uses such memory to implement several important data structures, namely, stacks, association lists, and closures.

Next is a description of the important registers and basic instructions needed by the SECD Machine. The descriptions of individual instructions are via essentially ***axiomatic semantics***, namely, how the machine state is changed by any of these instructions being executed. Extensions to this ***instruction set architecture (ISA)*** to cover special features are discussed in later chapters.

Following this is development of an abstract program that generates SECD code for simple s-expression programs. Given the equivalence of this s-expression format to abstract programs, the existence of such a compiler means that we could compile into SECD code any of the interpreters discussed earlier, or even the compiler itself.

Finally, the chapter discusses one of the thorniest problems to handle with functional languages, namely, how to handle arguments to functions which are themselves functions, particularly ones that have been only partially evaluated or curried. The is the famous ***funarg problem***, and our discussion will address not only what it is but how it affects the design of real machines and compilers, and why our simple SECD model avoided it.

The SECD Machine itself was invented by Peter Landin (1963) as a sample target for the first function-based language, LISP. As pictured if Figure 7-1, it has been used since as a basis for semantic descriptions, as an intermediate language for compilers and interpreters for LISP and other functional languages, and as the starting point for many of the current generation of LISP and Artificial Intelligence workstations. The ver-

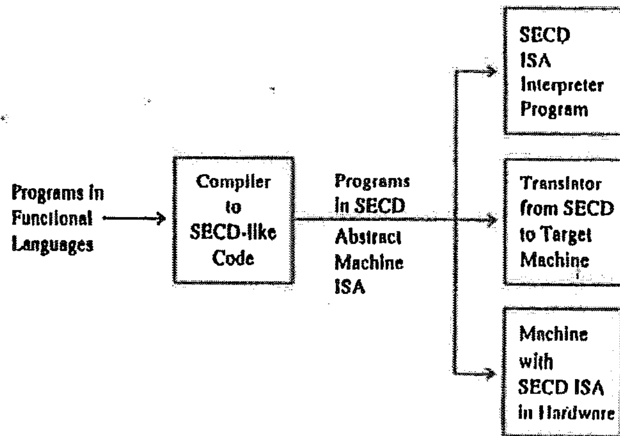



**FIGURE 7-1**
**Uses of the SECD instruction set architecture.**

226

sion of the SECD Machine described here is most closely related to that of Henderson (1981), but it has been modified in several aspects to improve its educational value. Actual Pascal code that describes an interpreter for an SECD Machine can be found in both Henderson (1981) and Henderson et al. (1983). Burge (1975) contains another good description.

## 7.1 LIST MEMORY

As described earlier, functional languages and s-expressions seem to go well together. Further, the implementation of s-expressions from conventional random-access memory seems to be fairly efficient. Consequently, it should come as no surprise if we assume that our abstract SECD Machine incorporates a memory model that supports s-expressions directly. This section gives an overview of a very simple form of such a model; Chapter 8 is devoted to a detailed discussion of more efficient implementation on real memory structures.

The basic SECD memory model assumes a large collection of identically formatted *cells* in a single memory pool (Figure 7-2). Each cell consists of some fixed number of memory bits and has a unique address through which the contents of the cell may be read or modified. Further, the contents of a cell may have several formats, each of which divides the cell's bits into several fields.

A common *tag field* in each cell describes how the rest of the cell

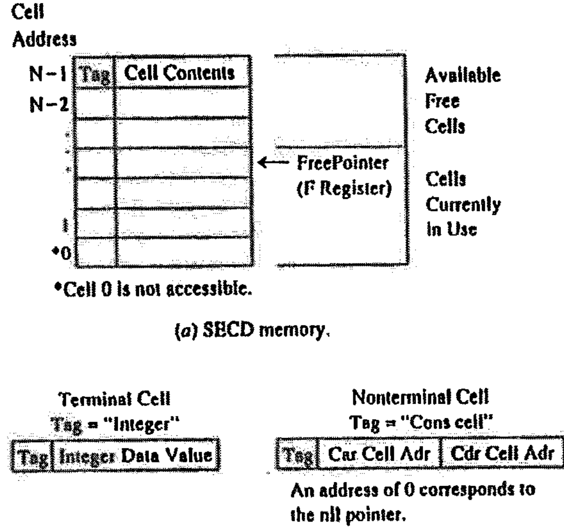



FIGURE 7-2
The SECD memory model.



should be interpreted. For this chapter we assume only two basic types: *integers* and *cons cells*. Later chapters will expand the set of interesting possibilities. The former correspond to *terminal cells*; the latter to *nonterminal cells*.

In integer-type cells the rest of the cell other than the tag field indicates the binary representation of some integer. For cons cells, the rest of the cell is divided into two halves: the *car field* and the *cdr field*. Both of these fields contain pointers (addresses) to other cells in the memory.

Arbitrary s-expressions are constructed as described in Chapter 3. When some kind of list or dotted pair is needed, it is built out of one or more cons cells interconnected by encoding the addresses of other cells in the car or cdr fields.

A pointer value of "0" indicates a *nil pointer*. Thus cell 0 is not usable by the system.

For now, when an SECD instruction wishes to build a new s-expression, it allocates new cells from memory, in a bottom-up fashion. A register in the machine, the *freelist pointer* (or *F register*), points to the current boundary. All cells at it or below it in memory are in use by the program; all cells above it are free to be allocated as needed. Allocating a new cell causes the F register to be incremented. The cell addressed by this incremented F register is then available for use as the new cell. The appropriate car and cdr values can then be written into it.

At program start, F is initialized to 0.

At this point we will ignore what happens when F runs over the top of available memory. This, and related questions on how to "reclaim" cells below F that are no longer in use, is a subject of the next chapter.

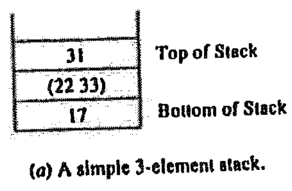
Finally, to simplify drawings, we will not show tag fields of individual cells unless necessary. If a cell is depicted as a single rectangle with a number in it, it has an integer tag. If it is divided into two subrectangles, it is a cons cell. A nil in either box represents a pointer to cell 0, the nil pointer. Also, as described in Chapter 3, we will very often record the value of a terminal cell in the field of the nonterminal that points to it. The meaning of this should be that the actual value is in fact in a separate cell with a pointer to that cell in the nonterminal.

## 7.2 BASIC DATA STRUCTURES

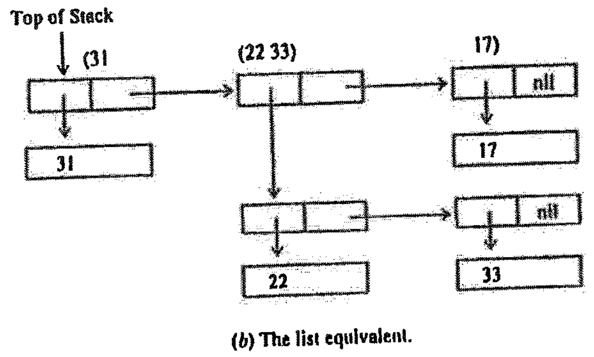
There are five key kinds of data structures that the SECD Machine will build, manipulate, and keep in memory:

- Arbitrary s-expressions for computed data
- Lists representing programs to be executed
- Stacks used by the program's instructions
- *Value lists* containing the arguments for uncompleted function applications
- Closures to represent unprocessed function applications

252




(a) A simple 3-element stack.




(b) The list equivalent.

FIGURE 7-4
Stacks as lists.

For example, if Figure 7-4(*a*) is implemented conventionally in sequential memory locations, popping 31 off and then pushing, say, 101, back on will cause the memory location holding the 31 to now contain 101. The 31 is lost.

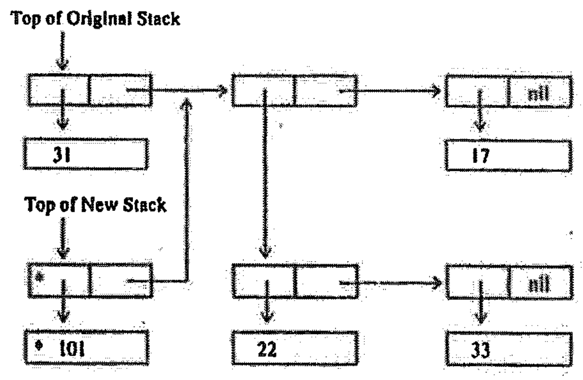
In comparison, doing the same thing to Figure 7-4(*b*) erases neither the cell containing the 31 nor the cell containing the pointer to it. The cdr of that cell still points to the second list cell, as does the cdr of the new list cell whose car points to 101. See Figure 7-5.

There are cases where keeping this old stack available without modification is a valuable feature. There are many other cases, however, where this represents the generation of ***garbage***, that is, memory cells that are no longer used but are not available for reuse. Recovering these old cells if in fact no one else needs them is a function of a ***garbage collection*** system, which the SECD Machine defined to this point does not have, but which most real machines with such memories do have. Again, the next chapter will address such systems.

### 7.2.3 Value Lists

The previous chapter demonstrated the utility of association lists in implementing lambda calculus-based languages. They permit simple combinations of normal- and associative-order evaluations. Equally important for future topics, they provide, through closures, a natural mechanism for deferring an application.






Operation performed: Push 101 onto stack resulting from popping top off of stack (31 (22 33) 17).

* = new cells allocated during operation.

FIGURE 7-5
Popping and pushing a list-managed stack.

Given that the SECD Machine supports arbitrary s-expressions, it naturally supports association lists. In particular, the instruction set of the next section supports directly a form of association list that includes only the value half of each pair. The SECD form is then a list of sublists, where each sublist contains the argument values (and not the names) for a particular function call that has been made but as yet is not complete. Figures 7-6 and 7-7 diagram some examples of this.

A simple compiler can eliminate the need for the identifier name part by building an analog at compile time, measuring exactly where in

let x=1 and y=2 and z=3 in (x+y)+z
Equivalent to: (λxyz|(x+y)+z) 1 2 3

Association list for code = ( ((x.1) (y.2) (z.3)) )
Name list = ((x y z))
Value list = ((1 2 3))

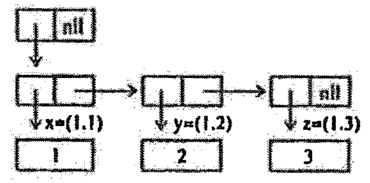



FIGURE 7-6
A simple value list.

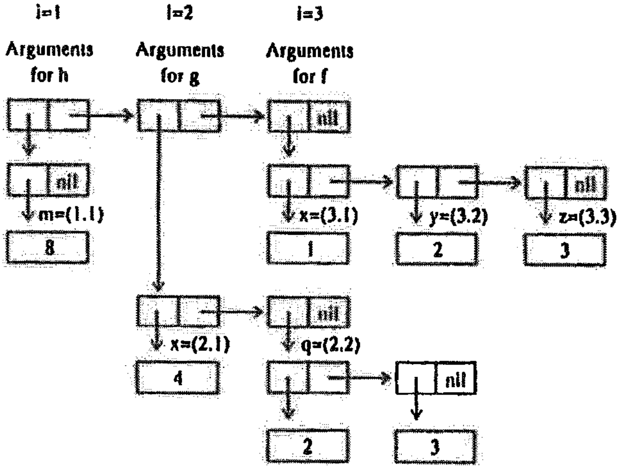
```
let h(m) = (3+m)×9 in
  let g(x,q) = h(car(q)×x) in
    let f(x,y,z) = g(x+3, list(y,z)) in f(1,2,3)
```

Note: f(1,2,3) → g(4, (2 3)) → h(2×4) → (3+8)×9 → 99

(*a*) A sample program.

| When Executing: | Alist: | Value List: |
|---|---|---|
| code for f | ((x.1) (y.2) (z.3))) | ((1 2 3)) |
| code for g | (((x.4) (q.(2 3))) ((x.1) (y.2) (z.3))) | ((4 (2 3)) (1 2 3)) |
| code for h | (((m.8) ((x.4) (q.(2 3))) ((x.1) (y.2) (z.3))) | ((8) (4 (2 3)) (1 2 3)) |

(*b*) Matching association lists.




(*c*) The value list for h.

FIGURE 7-7
More complex value lists.

the association list the value will be at run time, and encoding that index into the appropriate SECD Machine instructions. This index will be of the form (*i*,*j*), where both *i* and *j* are integers. The *i* value determines which sublist of the alist is desired, and then *j* determines which element of that sublist is the actual argument. Thus *i* corresponds to how far back in the stack of pending function calls the identifier is found as an argument, and *j* determines which of that call's arguments is the desired one.

For reference, Figure 7-8 defines a function *locate* that, when given



```
locate(ij, vlist) = loc(cdr(x), loc(car(x), vlist))
  whererec loc(y, z) = if y = 1 then car(z) else loc(y − 1, cdr(z))
= j'th element of i'th sublist of vlist where ij = (i.j)
```

FIGURE 7-8
The locate function.

Closure for h just before application to argument list (8)

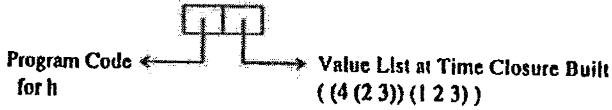



FIGURE 7-9
Closures as lists.

a paired index and a value list, returns the appropriate element. Note the interesting double use of the recursive subfunction loc.

### 7.2.4 Closures

To the SECD Machine a *closure* is the combination of the code for some function and a value list. This combination is such that the actual function can be unpacked and executed at any time after the closure is built, and will return an answer that is absolutely identical to what would have been returned if the function had been executed at the the time the closure was built. As pictured in Figure 7-9, such a closure consists of consing two pointers into a single memory cell—a pointer to the function's code and a pointer to the appropriate value list.

### 7.2.5 Recursive Closures

The last chapter closed with a discussion of association lists that support recursively defined functions. The final solution was to package the expression being evaluated in a closure where the association list included as values separate closures for each of the recursive functions. The association lists for each of these embedded closures was simply a pointer back to the association list for the expression being evaluated. Thus when a function required a call to itself, its association list would lead back to the closure defining it.

Given the SECD memory model, this translates directly into a data structure. Figure 7-10 diagrams such a configuration just before the overall expression is evaluated. The value list for E's closure is a list of cells where the first argument is the list of values for f1 through fn. Each of these values is a closure where the value list is a pointer back to the whole value list for E. The only difference between this and any of the

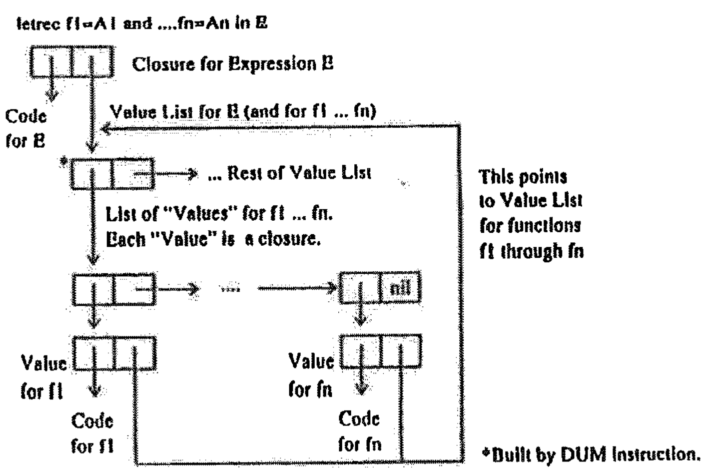



FIGURE 7-10
Value lists and closures for recursive expressions.

s-expressions we have encountered so far is that there is a circular loop in the pointer trail. Construction of such a loop cannot be done with the cons operator; a special SECD instruction to be described later is needed.

## 7.3 THE SECD MACHINE REGISTERS

The basic instruction set architecture of the SECD Machine consists of instructions that manipulate four main data structures found in memory. The structures consist of an evaluation stack (called simply the *stack*) for basic computations, a value list or *environment* for storing argument values, a *control list* for the current program, and a *dump* where copies of the other three structures can be stored when one function application is suspended and another executed. Four machine registers, the S, E, C, and D registers, control each of these structures. As described earlier, a fifth register, the F register, indicates the next available memory cell for any of these structures.

The instruction set is divided into roughly three parts. One set of instructions manages the evaluation of "built-in" functions that the machine is capable of executing directly. Another set of instructions deals with *special forms* such as if-then-else. A final set deals with application of program-defined functions to program-specified expressions for arguments. There is considerable distinction made between nonrecursive and recursive functions.

For the most part, the stack, environment, and dump act like conventional stacks; items are "pushed" on the top and "popped" off in a last-in, first-out fashion. Further, except for the fact that everything is built out of linked cells rather than sequential memory locations, all these



structures, registers, and associated instructions are close analogs to other abstract machine ISAs that support conventional programming languages [such as the p-code architecture for Pascal and the Forth threaded code architecture (Kogge, 1983).]

The following subsections describe these registers in more detail. Later sections describe the instructions and give example programs.

### 7.3.1 The S Register

The *S register* points to a list in memory that is treated as a conventional evaluation stack for *built-in* functions, such as the standard arithmetic (+, −, ×, /) and list operations (car, cdr, cons). Objects to be processed by these functions are "pushed" on by doing a cons of a new cell onto the top of the current stack. The car of this cell points to a copy of the object's value. The S register after such a push points to the new cell. New cells are obtained by incrementing the F register and using the cell location specified by the result.

When an instruction specifying a built-in function application is executed, the appropriate objects for its arguments are obtained from the cars of the cells at the front of the list. The result will be placed in a new cell, and S set to point to yet another new cell whose car points to the new value and whose cdr points to the stack list remaining after the arguments. For example, an add (Figure 7-11) will take the values pointed to by the cars of the top two cells in the S list, add them, and set S to point to a cell whose car contains a pointer to the sum and whose cdr points to the stack after the original top of stack cells.

A key point is that, unlike a conventional stack built out of sequential memory locations, this new result does not overwrite the memory locations containing the original inputs. New cells are allocated both for the value and for the pointer cell in the list. The reason for this is that in various circumstances these original inputs (in fact, the entire stack before the function) are needed at other points in the overall computation. The drawback, of course, is that more storage is taken up, particularly if no one else needs the inputs. As defined so far, this storage for the original inputs is simply lost to future use. The next chapter will describe a common technique used by most real systems to identify when such "garbage" is generated, and to "collect" it for reuse.

### 7.3.2 The E Register

The *environment register* (or *E register*) points to the current value list of function arguments (see Figures 7-6 and 7-7). This list is referenced by the machine when a value for an argument is needed, augmented (via a cons) when a new environment for a function application is constructed, and modified when a previously created closure is unpacked. The pointer from the closure's cdr replaces the contents of the E register.

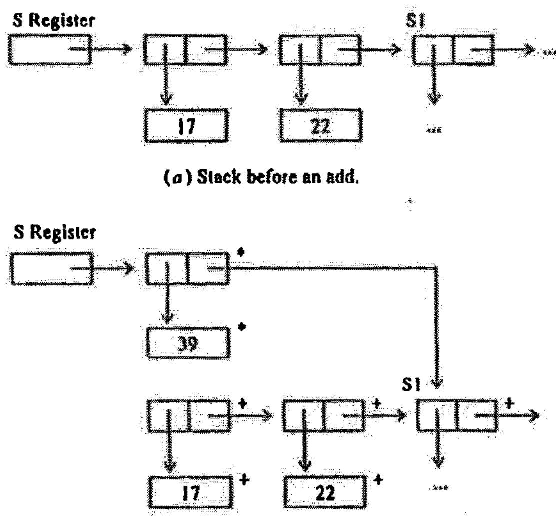



FIGURE 7-11
The S register and the stack.

As with the stack above, the prior value list designated by the E register is not overwritten by any change to E, and is still intact in memory. Other mechanisms, including the dump, often retrieve the old list when the current function completes.

### 7.3.3 The C Register

The ***control pointer*** or ***C register*** functions just like the ***program counter*** or ***instruction counter*** in a conventional computer. It points to the memory cell that designates through its car the current instruction to execute. In the SECD Machine these instructions are simple integers which specify the desired operation. Unlike many conventional computers, there are no specialized subfields for registers, addressing designations, etc. When additional information is needed for an instruction, such as which argument to access from the E register's list, the information comes from the cells chained to the instruction cell's cdr.

Conventional computers normally increment their program counter after completing most instructions. The analog in the SECD Machine is the replacement of the C register by the contents of the cdr field of the last memory cell used by the current instruction (C←cdr(C)). Again as with conventional machines, there are exceptions to this, particularly for the equivalent of conditional branches, new function calls, and returns from completed application. Here the C register is replaced by a pointer provided by some other part of the machine.



### 7.3.4 The D Register

The ***dump register*** or ***D register*** is the last of the SECD Machine's registers. As with the S register, the D register points to a list in memory, this time called the ***dump***. The purpose of this data structure is to remember the state of a function application when a new application in that function's body is to be started. This is done by appending onto the dump three new cells which record in their cars the values of the S, E, and C registers before the application. When the application completes, popping the top of the dump restores those registers to their original values so that the original application can continue. This is very similar to the ***call-return*** sequence found in conventional machines for procedure or subprogram activation and return.

## 7.4 THE BASIC INSTRUCTION SET

Figure 7-12 diagrams the basic instruction set for the SECD Machine. For each instruction there is a brief description of how the machine's four registers change after its execution. The notation used consists of four s-expressions before and after a "→." The four s-expressions before the "→" represent the assumed lists in memory pointed to by the S, E, C, and D registers just as the machine starts to execute the instruction. The s-expressions after the "→" represent the same four registers after the instruction has been executed.

As described earlier, the notation "(x y.z)" stands for an s-expression whose first two elements are x and y, respectively, with the rest of the expression z.

With this convention, all the original s-expressions for the C register consist of a list with the first element designated by the name of the instruction being executed. In real life each such instruction would be a cell containing a specific integer. Thus any cell containing a "2" in a program list might represent an LDC, while a "15" might represent an ADD. For readability here we will use a mnemonic form.

These instructions break down into six separate groups, namely, those that:

1. Push object values onto the S stack
2. Perform built-in function applications on the S stack and return the results to that stack
3. Handle the if-then-else special form
4. Build, apply, and return from closures representing nonrecursive function applications

202

Several variations of these data structures will be described in later chap-
ters, particularly when we discuss lazy evaluation.
It is clear how the first of the above five categories, namely, s-
expressions, can be built in the memory from the last section. Given that
the SECD Machine contains instructions that perform the equivalent of
cons, building a new s-expression involves allocating a new cell from
memory (bumping the F register by 1), setting its tag to cons, and storing
the appropriate pointers into its car and cdr fields.

The other data structures are addressed individually in the following sections.

### 7.2.1 Programs as Lists

Programs for the SECD Machine look like garden-variety lists. Each el-
ement of the list corresponds to an individual instruction, and execution
proceeds (for the most part) one element at a time from the front of the
list to the rear. A call to a function involves saving where one is in the
current list and starting execution at the beginning of the list associated
with the called function.

While it may seem wasteful to link together strings of often sequen-
tial instructions as a list (rather than as sequential words in an "array"),
there are several significant advantages. First, we do not need to invent a
new data structure just for program storage. Second, and most impor-
tant, programs now look just like data, making it extraordinarily easy to
write program that read, process, or even generate other programs. This
makes writing compilers and interpreters for the SECD Machine in
SECD code a relative snap.

The individual elements of a program list come in two types: simple
integers or arbitrary lists. The former, simple integers, are equivalent to
an *opcode* specifying some basic instruction in the SECD Machine's ar-
chitecture. The latter, embedded sublists, usually represent alternative
branches of program flow that will be decided dynamically when the pro-
gram is run. Instructions which choose between these alternative flows
of control correspond somewhat to the branch and call instructions found
in conventional architectures. For example, to do an if-then-else, one el-
ement of a program's list will be a basic instruction (called out by an in-
teger), which when executed will decide which of the two following ele-
ments of the program list should be executed. These following elements
will be lists themselves. Each represents a then or else snippet of code.
Instructions at the ends of the snippet returns control to the main thread.

Figure 7-3 gives a simple example.

### 7.2.2 Stacks as Lists

The SECD Machine uses several stacks during execution. One serves as
an evaluation stack in a way reminiscent of *reverse Polish* execution.



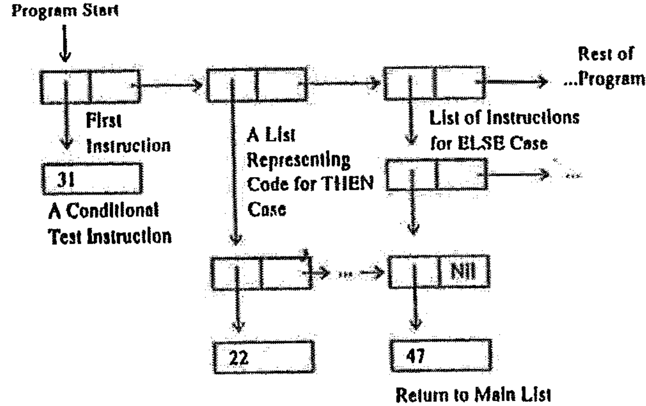



**FIGURE 7-3**
Programs as lists.

Other stacks contain points in the program to pick up from when the current functions are completed.

In all cases the stack operates just as a conventional stack would. We can *push* values onto the stack and *pop* them off again, all in a last-in, first-out manner.

The major difference from a conventional stack implementation is that, like programs, stacks in the SECD Machine are made from lists of memory cells. Again, this may seem wasteful of storage bits, but it does have the advantage of using the machine's natural data structure. Also, it permits us to grow different stacks to arbitrary sizes in arbitrary orders, until memory is exhausted. This is unlike conventional machines, which grow stacks in consecutive locations of memory until some preallocated boundary is reached. In such cases the maximum size of the stack is limited to the amount of storage allocated to it by the system. Overgrowing one stack's area is not permitted, even if storage exists in other stack or data areas.

By analogy, the top of the stack is equivalent to the leftmost element of its list equivalent. Thus, pushing an object to a stack is implemented by consing it onto the matching list. Popping an element requires a car to get the element's value, and a cdr to return a pointer to the rest of the list. An empty stack is the nil list.

Figure 7-4 diagrams a simple example.

A subtle but important difference between these stacks and conventional stacks built out of sequential memory is that with lists a "pop" followed by a "push" does not overwrite the storage allocated to the element "popped" off. The new element is created in a separate memory cell, with the cell's cdr pointing to the cell holding the next element, wherever it is. The cell holding the original value "popped" off still exists, with its cdr pointing to the same next cell.

| Instruction | Operation |
|---|---|
| | —Access Objects and Push Value to Stack— |
| NIL | s e (NIL.c) d → (nil.s) e c d |
| LDC | s e (LDC x.c) d → (x.s) e c d |
| LD | s e (LD (i.j).c) d → (locate((i.j),e).s) e c d |
| | —Support For Builtin Functions— |
| ATOM, CAR, CDR... | (a.s) e (OP.c) d → ((OP a).s) e c d |
| CONS, ADD, SUB,... | (a b.s) e (OP.c) d → ((a OP b).s) e c d |
| | —If-Then-Else Special Form— |
| SEL | (x.s) e (SEL ct cf.c) d → s e c? (c.d) where c? = ct if x≠0(T), and cf if x=0(F) |
| JOIN | s e (JOIN.c) (cr.d) → s e cr d |
| | —Nonrecursive Functions— |
| LDF | s e (LDF f.c) d → ((f.e).s) e c d |
| AP | ((f.e') v.s) e (AP.c) d → NIL (v.e') f (s e c.d) |
| RTN | (x.z) e' (RTN.q) (s e c.d) → (x.s) e c d |
| | —Recursive Functions— |
| DUM | s e (DUM.c) d → s (nil.e) c d |
| RAP | ((f.(nil.e)) v.s) (nil.e) (RAP.c) d → nil (rplaca((nil.e),v).e) f (s e c.d) |
| | —Auxiliary Instructions— |
| STOP | s e (STOP.c) d → s e (STOP.c) d -stop the machine |
| READC | s e (READC.c) d → (x.s) e c d where x is character read in from input device |
| WRITEC | (x.s) e (WRITEC.c) d → s e c d where x is printed on the output device |

where rplaca(x,y) "replace car of x by y, and return pointer to x"
and locate(x,e) = loc(cdr(x), loc(car(x),e))
wherereс loc(k,e) = if k=1 then car(e) else loc(k−1, cdr(e))

**FIGURE 7-12**
Basic instruction set for SECD Machine.

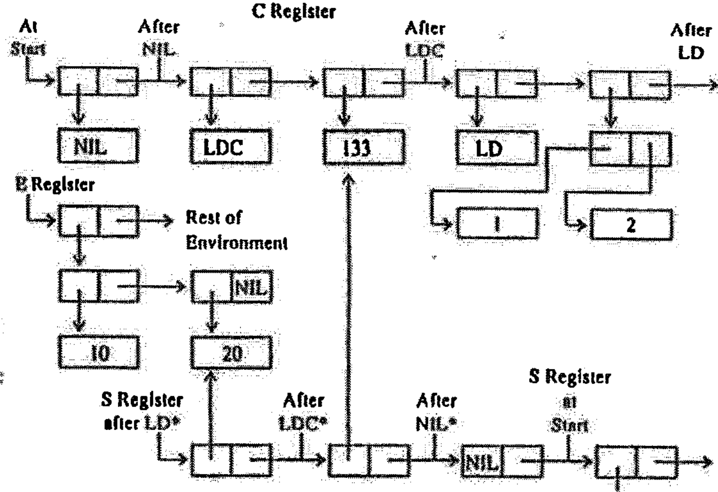
5. Extend the above to handle recursive functions
6. Handle input/output (I/O) and machine control

The following sections describe each of these groups.

### 7.4.1 Accessing Object Values

The first group pushes values of objects onto the S stack (see Figure 7-13 for an example). The first of these, *NIL*, pushes a nil pointer. Again this means that the S register after the instruction has been executed points to a newly allocated cell whose car contains a nil pointer (an address of lo-






*New cells allocated by instructions.

**FIGURE 7-13**
A sequence of data access instructions.

cation 0) and whose cdr points to the list designated by the S register before the instruction was executed.

The second of these, *LDC*, loads a "constant" onto the stack. The constant value to be pushed is found as the next element on the C list after that for the instruction (the cadr of the C list at the time the instruction is started). Again a new cell is allocated, and chained onto the top of the S list. The car of this new cell is loaded with a pointer to this constant. Note that the value is not duplicated—only a pointer to it is. This means that this "constant" could in fact be an arbitrary s-expression, and the effect would be the same. The top of the stack would be that expression.

For this SECD Machine, the representation for the boolean constant F (false) is 0, and T (true) is taken as any integer other than 0, usually 1.

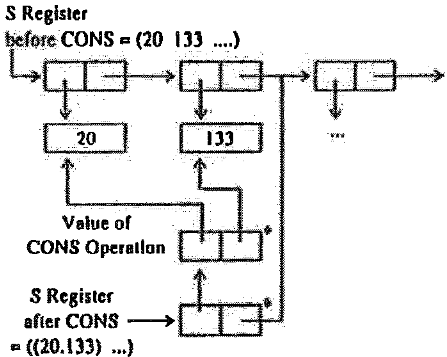
The third instruction of this group, *LD*, "loads" an element of the current environment. The cadr of the C list is a cell of the form (i.j), where the car i is the sublist element of the E list desired, and the cdr j is the element of that sublist. The function locate (see also Figure 7-8) describes how this access works. Again a pointer to that object is stored in the car of a new cell whose cdr points to the old S list.

### 7.4.2 Built-In Function Applications

The next group of instructions handle built-in functions such as CAR, CDR, CONS, ATOM, ADD, SUBtract,... Here the proper number of objects are accessed from the top of the S stack and the result placed in a new cell which is pointed to by the car of a second new cell. The cdr of this latter cell points to the rest of S's original list after the initial arguments. S is reset to point to this latter cell. Figure 7-11 diagrams a typical case for an ADD. A somewhat more complex example is the CONS instruction. This instruction allocates not one but two cons cells, one to hold the cons of the operands, and one to cons this result onto the S list. Figure 7-14 diagrams this, with Figure 7-15 diagramming the same situation in terms of possible memory location values.

### 7.4.3 Instructions for If-then-else

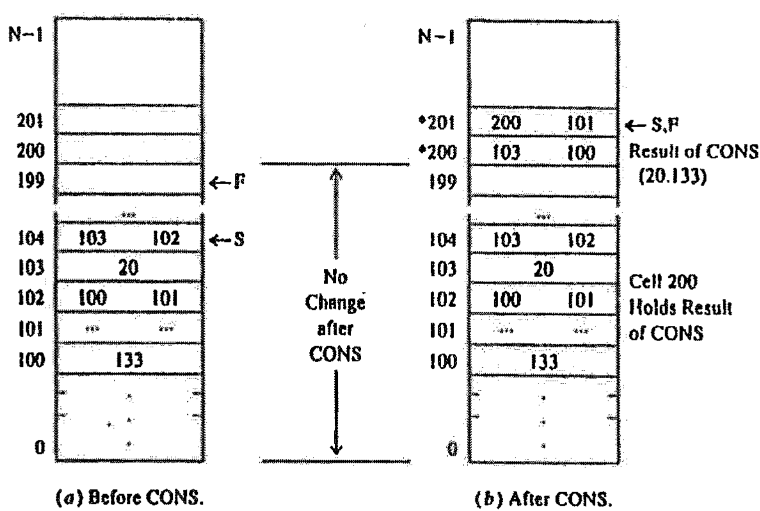
The group of instructions used to implement If-then-else special forms are used in a specific order. The *SEL* instruction ("select") assumes that the top of the S list is an integer either zero or nonzero (encoded as described above to represent a boolean). Following the SEL in the control list are two elements (the cadr and caddr of the C list), both of which are themselves lists of instructions. The last instruction in each list is a *JOIN*. When executed, the SEL will push onto the dump a pointer to the C list just beyond the second sublist (i.e., a pointer to the cdddr of the original C list). The machine will then pop the top element off the S stack, test it, and replace C with its cadr if the value was nonzero, and with its caddr if the value was zero. Thus, these sublists correspond to the code for then and else expressions, respectively. The last instruction of each of these




*New cells allocated during CONS execution.

FIGURE 7-14
Execution of the cons instruction.






*New cells allocated by CONS.

FIGURE 7-15
Memory before and after a cons instruction.

sublists, the JOIN, then resumes the original program by popping the top off the dump and resetting C to point to it. Figure 7-16 diagrams an example.

Although the normal use of a SEL is after some test instruction which leaves a boolean on S, its definition also permits its use as a zero test after any arbitrary instruction sequence, such as a SUB. Thus we do not really need a ZERO, NULL, or even an EQUAL instruction in the SECD ISA.

### 7.4.4 Nonrecursive Program-Defined Functions

The nonrecursive function application instructions also work together in a very specific way. The *LDF* ("load function") instruction is followed in the C list by an element pointing to a sublist containing the code representing some program-defined function. The last instruction in this subprogram list is a *RTN*, which will function similarly to a JOIN.

When LDF is executed, it builds in a new cell a closure consisting of a pointer to the new function's code and a copy of the current E register. The latter represents the value list for all the identifiers (other than the function's immediate arguments) that have been bound to specific values by previous code. The closure is pushed onto the top of the S list.

Note that the function is not executed at the current time, merely packaged in a closure on the stack. At some arbitrarily later point in time, an AP ("apply") instruction will find on the top of the S stack a copy of this closure, and underneath it a list representing the argument

258

Code fragment: if null (x) then 10 else −10

Program list (... NULL SEL (LDC 10 JOIN) (LDC −10 JOIN) ...)

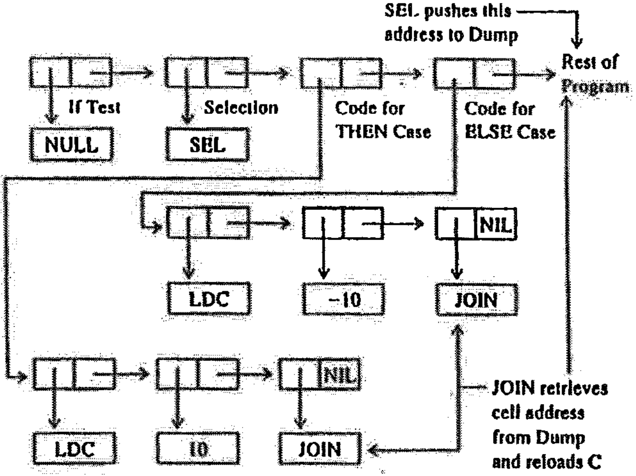



**FIGURE 7-16**
Structure of if-then-else code.

values to be applied to the function. Note that this argument list is a single element on the S list (namely, its cadr), and that the rest of the S list is arbitrary.

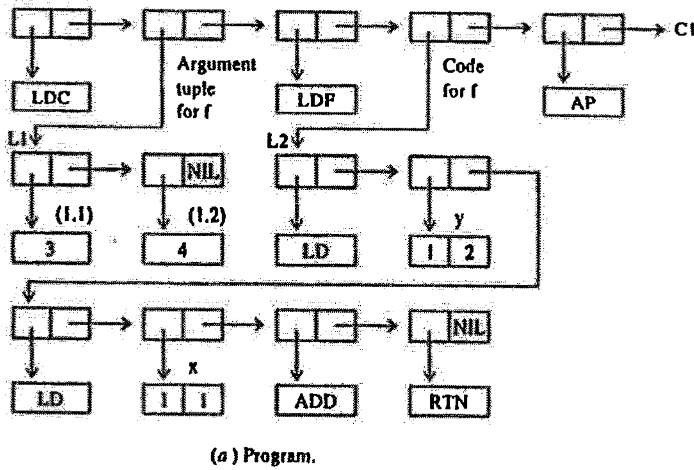
Executing the AP causes the cddr of S, the E, and the cdr of C to be pushed onto the dump. This represents the state the machine is to return to when the function application is complete. After this, the S register is reset to nil, making it an empty stack, the C register is set to the beginning of the code specified by the closure (the car of the closure cell), and the E register is set to the cons of the second element on the original S list and the cdr of the closure cell. This latter operation establishes a value list for the function application which consists of the function's arguments in the first sublist and the environment needed by the function's internal definition as the rest. Appropriate (i,j) indexing constants inside the function's code will give it access to these values.

The last instruction of a function's code list should be a RTN. This instruction takes the top element off the S stack as the value to return from the application. This value is consed to the old S value previously pushed on top of the dump, with S reset to point to this list. The E and C registers are restored directly from the dump, and the calling function is restarted. The only difference from the point at which it called the function is that the top of the S stack now contains the desired result.

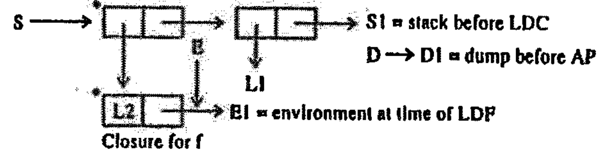
Figure 7-17 diagrams a simple sequence of code that uses this set of instructions. Note that the actual program list shown was chosen for its



Assume: let f(x,y) = x + y in f(3,4)
Possible SECD code: (...LDC (3 4) LDF (LD (1.2) LD (1.1) ADD RTN) AP ...)

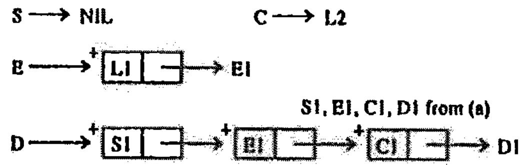



(a) Program.




*New cells allocated by LDF.

(b) After the LDF.




+New Cells Allocated by AP

(c) After the AP.

**FIGURE 7-17**
Basic nonrecursive application instructions.

simplicity in explaining LDF and AP and thus is somewhat different from that which would be produced by the basic compiler described later in this chapter.

In Figure 7-17 the argument list for the function application is a list of constant values. This is not always the case. Normally the arguments for the call must be computed by some set of function applications themselves. Given the way the AP instruction works, though this means that however the arguments are computed, they must be gathered into a list before we can invoke the AP. In the SECD architecture the easiest way to do this is to evaluate the arguments before invoking the function, but cons them together first. This involves using an initial NIL instruction before any argument is evaluated, and then evaluating the arguments one at a time, from right to left, with a CONS instruction after each evaluation sequence to put the argument value onto the growing argument list. The leftmost argument is thus evaluated last and placed on the front of the list. Figure 7-18 diagrams an example of this. The compiler discussed later implements the process of generating the proper SECD code.

### 7.4.5 Recursive Program-Defined Functions

The group of instructions handling recursive function calls are extensions of the previous set, and are perhaps the most difficult of the SECD instructions to understand. The reader should simply try to get the general idea here, and then observe later examples to see how everything fits together.

The *DUM* instruction conses onto the front of the environment list a new cell whose car is nil. This corresponds to a new argument list that is initially empty (a "dummy" list). This list will eventually be a pointer to the self-looping value list as shown in Figure 7-10.

A DUM instruction is used in a program just before the LDF(s) to build the closures for the recursively defined functions. This will make the environments stored in those closures point to a value list where the first element is this current "dummy" sublist.

The *RAP* instruction ("recursive apply") assumes that the top of the S stack looks like that for an AP, namely, a closure representing a function to be executed, and an argument list (i.e., the arguments for the function to be executed, and an argument list (i.e., the arguments for the expression at the end of the letrec...and..in). In this case the function in the closure is the expression that calls the recursively defined functions (see Figure 7-19). The closure's environment is a pointer to the same list indicated by the current E register. This, in turn, should be a list where the first element was that built by the DUM. Furthermore, the list of argument values should be a list of closures, one per recursively defined function, where the environments of these closures are also pointers to the same dummy cell.

Execution of the RAP is identical to the AP except that the car of



Program: let f(x,y)=x+y in f(2×3,6−4)
Code: (NIL LDC 6 LD 4 SUB CONS
LDC 3 LDC 2 MPY CONS
LDF (LD (1.2) LD (1.1) ADD RTN)
AP ...)

S → SI

(*a*) Before code starts.

S → [nil | ] → SI

(*b*) After NIL.

(*c*) After first CONS.

(*d*) After LDC 3 LDC 2 MPY.

(*e*) After final CONS.

**FIGURE 7-18**
Construction of argument lists.

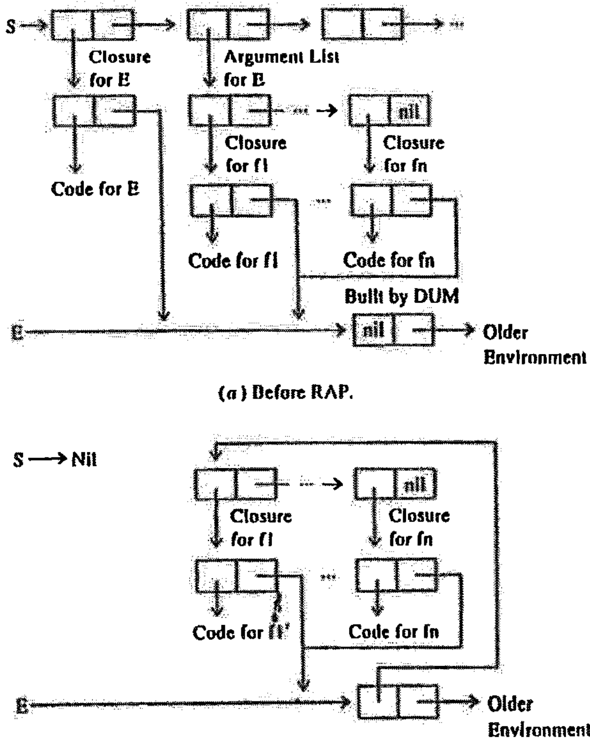
the cell pointed to by E (i.e., the dummy cell) is reset to point to the second argument of the S stack (the list of closures). Given that the closures in that argument also point to the dummy cell, the result is exactly the loop of pointers desired.

Within the code called by the RAP, any required calls to the i-th recursively defined function fi are initiated by a LD(1,i) followed by an AP. The LD fetches the closure for the function from the environment, and AP unpacks it as before. The environment established by the closure is the same environment it came from, with the exception that the AP adds a list to the front representing the arguments to the function. Thus, a LD(2.i) from inside the code for the function fi will retrieve an identical copy of its closure, and another AP will thus start a recursive call to fi properly.

Again, a RTN at the end of the expression unstacks the original S, E, and C values stacked by the RAP function(s).

Assume: letrec f1 = A1 and ... and fn = An in E
= (λ f1 ... fn|E) A1 ... An

Code = (DUM NIL LDF (..code for An... RTN) CONS
LDF (..code for A1..RTN) CONS
LDF (..code for E..RTN) RAP )




(*a*) Before RAP.

(*b*) After RAP.

**FIGURE 7-19**
Executing a RAP instruction.

### 7.4.6 Machine Control Instructions

The final set of SECD instructions control other aspects of a real machine's operation. The first of these, ***STOP***, stops the machine in its tracks. No more instructions are executed, and the registers are left unchanged. This should be the last instruction in any program.

The ***READC*** and ***WRITEC*** instructions perform simple input/output operations. READC takes a character from some input device, constructs the integer equivalent in a new cell, and pushes a pointer to it onto the S stack.



WRITEC takes the integer on the top of the S stack and prints a character equivalent of it out to the standard output device.

These instructions are included here simply for minimal completeness. In real life much more complex I/O would be found, and would be handled much as in conventional machine architectures.

## 7.5 COMPILING SIMPLE S-EXPRESSIONS

Writing an elementary ***compiler*** to generate SECD code from an s-expression program is a relatively straightforward process. We recursively take the s-expression, look at its car element, and generate a standardized set of code based on its value and structure. Subexpressions embedded in the s-expression are converted into SECD code lists and then appended into the overall code. Figure 7-20 lists these SECD code sequences for each major special form.

For simplicity of notation here we assume that *(expr) stands for the SECD code compiled from the s-expression (expr). Also, AB stands for the result of appending list A to list B, as in (1 2)(3 4)=(1 2 3 4).

Figure 7-21 outlines an abstract program which implements these transforms; Figure 7-22 lists some functions used in this program to handle special forms. The main function compile has three arguments, the expression *e* being compiled, a namelist *n*, and an accumulating parameter *c*. The namelist represents the variables that would be available in the environment when the s-expression *e* is executed. The accumulating parameter represents already-generated code to which the new SECD code for *e* should be appended on the front. Thus the initial call to compile an expression *e* would look like

compile(*e*, nil, (STOP))

With the exception of the if-then-else form, most of the s-expression forms are compiled into SECD code which emulates an applicative-order evaluation—the arguments to a function are evaluated first.

There is no error-checking built into this compiler, either syntatically or semantically.

### 7.5.1 Data Accessing Forms

The data accessing forms translate directly into sequences of SECD instructions that push the appropriate value onto the top of the S stack. In the case of an identifier, this involves identifying what entry in the value list will have the appropriate value at run time. The compiler of Figure 7-21 computes this by looking through argument *n* for the first entry that has the same symbolic name and keeping track of where the match occurred (the function index in Figure 7-22).

```
Syntax: <number>
Code: (LDC <number>)

Syntax: nil
Code: (NIL)

Syntax: <identifier>
Code: (LD (i.j)) where (i.j) is index into E

Syntax: (<builtin> <expr>1 ... <expr>n)
Code: *<expr>n|| ... ||*<expr>1|| (<builtin>)
Example: (MPY (ADD x 1) 256)
        → (LDC 256 LDC 1 LD (1.1) ADD MPY)

Syntax:(IF <expr>1 <expr>2 <expr>3)
Code: *<expr>1 || (SEL) || (*<expr>2 || (JOIN)) || (*<expr>3 || (JOIN))*<expr>2
Example: (IF (null x) 1 (car x))
        → (LD (1.1) NULL SEL (LDC 1 JOIN) (LD (1.1) CAR JOIN))

Syntax: (LAMBDA (<id>1 ... <id>n) <expr>)
Code: (LDF)||(*<expr> || (RTN))
Example: (LAMBDA (x y) (ADD x y))
        → (LDF (LD (1.2) LD (1.1) ADD RTN)

Syntax: (LET (<id>1 ... <id>n) (<expr>1 ... <expr>n) <expr>)
Code: (NIL)|| *<expr>n || (CONS) || ... *<expr>1 ||
        (CONS LDF) || (*<expr> || (RTN)) || (AP)
Example: (LET (x y) (1 2) (+ x y))
        → (NIL LDC 2 CONS LDC 1 CONS
           LDF (LD (1.2) LD (1.1) ADD RTN) AP)

Syntax: (LETREC (<id>1 ... <id>n) (<expr>1 ... <expr>n) <expr>)
Code: (DUM NIL) || *<expr>n || (CONS) || ... *<expr>1 ||
        (CONS LDF) || ( (*<expr> || (RTN)) RAP)
Example: (LETREC (f) ((LAMBDA (x m)
            (IF (null x) m (f (CDR x) (+ m 1)))) (f (1 2 3) 0))
            → (DUM NIL LDF (LD(1.1) NULL SEL
                                (LD (1.2) JOIN)
                                (NIL LDC 1 LD (1.2) ADD CONS
                                 LD (1.1) CDR CONS LD (2.1) AP JOIN)
                                 RTN)
                  CONS
                  LDF (NIL LDL 0 CONS LDC (1 2 3) CONS (1.1) AP RTN)
                  RAP)

Syntax: (<expr> <expr>1 ... <expr>n)
Code: (NIL)|| *<expr>n||(CONS)|| ... || *<expr>1||(CONS)|| *<expr>||(AP)
Example: ((LAMBDA (x y) (MPY x y)) 1 (PLUS 2 3))
            → (NIL LDC 3 LDC 2 ADD CONS LDC 1 CONS
               LDF (LD (1.2) LD (1.1) MPY RTN) AP)


Note: A||B = append(A,B). Thus (NIL)||(CONS) = (NIL CONS).
Also *<expr> is compiled form of <expr>.
```

FIGURE 7-20
Code sequences for s-expressions.



```
compile(e,n,c) = "compiler for expression e"
  if atom(e)
  then "a nil, number, or identifier"
       if null(e)
       then cons(NIL,c)
       else let ij = index(e,n) in
            if null(ij)
            then cons(LDC, cons(e, c))
            else cons(LD, cons(ij, c))
  else let fcn = car(e) and args = cdr(e) in
       if atom(fcn)
       then "a builtin, lambda, or special form"
            if member(fcn, builtins)
            then compile-builtin(args, n, cons(fcn, c))
            elseif fcn = LAMBDA
            then compile-lambda(cadr(args), cons(car(args), n), c)
            elseif fcn = IF
            then compile-if(car(args), cadr(args),
                            caddr(args), n, c)
            elseif fcn = LET or fcn = LETREC
            then let newn = cons(car(args), n)
                 and values = cadr(args)
                 and body = caddr(args) in
                    if fcn = LET
                    then cons(NIL, compile-app(values, n,
                              compile-lambda(body, newn, cons(AP,C))))
                    else " a letrec"
                    append((DUM NIL),
                           compile-app(values, newn,
                           compile-lambda(body, newn, cons(RAP, c))))
            else "fcn must be a variable"
              cons(NIL, compile-app(args, n, cons(LD, cons(index(fcn, n), cons(AP, c)))))
       else "an application with nested function"
       cons(NIL, compile-app(args, n, compile(fcn, n, cons(AP, c)))
```

FIGURE 7-21
A compiler from s-expressions to SECD code.

### 7.5.2 Built-in Function Applications

The code generated for the application of ***built-in functions*** consists of the sequences of SECD code needed for each argument appended to the SECD instruction that performs the function. By convention, the rightmost argument in the s-expression form is computed first, and then execution proceeds from right to left. The net effect of this is that at execution time the topmost value on the stack is the leftmost argument, with later values further down the stack.

### 7.5.3 Conditional Forms

The compilation of a ***conditional form*** is an SECD code list consisting of the sequence of instructions that evaluates the test expression, followed

```
compile-builtin(args, n, c) =
  if null(args)
  then c
  else compile-builtin(cdr(args), n, compile(car(args), n, c))

compile-if(test, then, else, n, c) =
  compile(test, n,
          cons(SEL, cons(compile(then, n, cons(JOIN,nil)),
          cons(compile(else, n, cons(JOIN,nil)), c))))

compile-lambda(body, n, c) =
  cons(LDF, cons(compile(body, n, cons(RTN, nil)), c))

compile-app(args, n, c) =
  if null(args)
  then c
  else compile-app(cdr(args), n,
                   compile(car(args), n, cons(CONS, c)))

index(e,n) = indx(e, n, 1)

indx(e, n, i) =
  if null(n) then nil
  else letrec indx2(e, n, j) =
       if null(n) then nil
       elseif car(n)=e then j
       else indx2(e, cdr(n), j+1) in
  let j = indx2(e, car(n), 1) in
    if null(j).
    then index(e, cdr(n), i+1)
    else cons(i, j)
```

**FIGURE 7-22**
Auxiliary functions to compile special forms.

by a SEL instruction, followed by two lists which correspond to the then and else expressions. respectively. Each of these sublists is computed by recursively calling **compile** with the accumulating parameter initialized to (JOIN). This places the JOIN at the end of each sublist. The auxiliary function **compile-if** in Figure 7-22 performs this process.

### 7.5.4 Lambda Function Definitions

*Programmed functions* are any functions that are defined either explicitly or implicitly by a lambda expression. This includes *lambda expressions, let expressions,* and *letrec expressions.* The former simply defines a function; the last two include both function definition and their application.

The code compiled for a lambda expression consists of a two-element list, the first of which is a LDF instruction and the second of which is a list consisting of the compiled form of the lambda expression's body terminated by a RTN. As with the conditional forms, this RTN is



inserted by recursively calling compile with its third argument set to (RTN) (see **compile-lambda** in Figure 7-22).

Executing such a code sequence will push onto the S stack a closure whose code pointer is pointing to the sublist following the LDF and whose embedded environment is the cell pointed to by E when the LDF is executed.

When compile is called recursively for the lambda's body, the namelist argument has the list of argument identifiers for that lambda appended to the namelist passed into compile when the lambda was discovered. This reflects the fact that once inside the body of a lambda, the top of the value list on E must correspond to a list of the lambda's arguments, with the rest of the environment list consisting of the environment that was present before the function was applied. Further, the order of the identifier names in this sublist should be the same as the order in which the values will be when the code is executed. This permits a search of the namelist to return the proper $(i,j)$ value to encode into LD code.

### 7.5.5 Combined Function Definitions and Applications

Let and letrec expressions correspond to evaluating a series of expressions, associating them with identifiers, and then evaluating a new expression that references these identifiers. The code compiled for these forms must compute each of these expressions and then *cons* the results together into a list that can be passed as an argument to the lambda function implied by the in expression.

**Compile** does this by pushing an initial nil onto the stack, and then generating code for the rightmost let expression that will leave the result of that expression above the nil. A CONS instruction combines these two into a single element list.

The process of compiling code for each of the remaining let expressions is similar. The process goes from right to left, compiling in a sequence of code to evaluate each subexpression, and following it by a CONS to append it to the previous expression list.

When such code is executed, it leaves on the top of the stack a single element which is itself a list. The order of the elements on this list corresponds directly to the order of let expressions, from left to right.

After this, the compiled code sequence consists of a LDF followed by a code list representing the in expression and terminated by a RTN. When executed, this pushes a closure representing the in expression on top of the value list.

For a let expression the final instruction compiled into the sequence is an AP. When executed, this instruction unpacks the closure for the in expression, and starts the body, with the computed values established on the top of E.

The only difference for a letrec expression is that a RAP is used in place of an AP, and there is an extra instruction to begin the sequence,

283

namely, a DUM. As discussed earlier, this helps build a dummy environment that RAP modifies to form the environment loop.

### 7.5.6 Nested Function Expressions

The final special form handled by the compiler is when the expression in the function position of an s-expression is something other than a keyword. The compiler function compile-app handles this case. As with let, the arguments are evaluated one at a time from right to left and consed into a list so that the leftmost one is first. Then the code for the function subexpression is compiled and appended to the right of the argument evaluation code, again just as for let. This subexpression could be anything, as long as when the code is executed it places a closure on the top of the stack. In most cases this subexpression will be a (lambda...) expression or an identifier which has been bound earlier to a (lambda...). Again there is no error checking to see if this is in fact the case.

The final instruction compiled for this type of an expression is an AP. When executed, it will take the list of argument values and the closure and evaluate the combination.

## 7.6 SAMPLE COMPILATION

This section diagrams the code that would be compiled for yet another variation of our familiar factorial function. This definition includes an extra let at the beginning with an assignment of "1" to the identifier *one* just to show the differences between that and letrec. Not all calls to compiler functions are shown; the calls chosen are those with significance, such as major changes in the namelists used to determine where variables are in the environment.

The original abstract program to compile is:

```
let x=3 and one=1 in
    letrec fact(n, m)=
        if (eq n 0) then one
        else fact(n− one, n× m)
    in fact(x,one)
```

The s-expression equivalent of this is:

```
(let (x one) (3 1)
    (letrec (fact)
        ((lambda (n,m) (if (= n 0) one (fact (− n one) (× n m)))))
        (fact x one)))
```



The compilation proceeds as follows:

```
compile((let...),nil,(STOP))
⇒ (NIL.compile-app((3 1), nil,
    compile-lambda((letrec...), ((x one)), (AP STOP))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF.(compile((letrec...), ((x one)), (RTN))(AP STOP)))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app(((lambda...)), ((fact)(x one)),
        compile-lambda((fact x one), ((fact)(x one)),
        (RAP RTN))(AP STOP))))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app(((lambda...)), ((fact)(x one)),
        (LDF
            (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
            RAP RTN)
     AP STOP)))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app((), ((fact)(x one)),
        compile((lambda...), ((fact)(x one)),
            (CONS LDF
              (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
              RAP RTN)
              AP STOP))))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app((), ((fact)(x one)),
        compile-lambda((if...), ((n m) (fact) (x one)),
            (CONS LDF
              (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
              RAP RTN)
              AP STOP))))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app((), ((fact)(x one)),
        (LDF.(compile((if...), ((n m) (fact) (x one)), (RTN))
        (CONS LDF
              (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
              RAP RTN)
              AP STOP)))))))
```

265

```
⇒ (NIL.compile-app((3 1), nil,
     (LDF (DUM NIL.compile-app((), ((fact)(x one)),
       (LDF
          (LDC 0 LD (1.1) EQ SEL
             (LDC 1 JOIN)
             (NIL LD (1.2) LD (1.1) MPY CONS
                 LD (3.2) LD (1.1) SUB CONS
                 LD (2.1) AP JOIN)
             RTN)
        CONS LDF
          (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
          RAP RTN)
     AP STOP))))))
⇒ (NIL.compile-app((3 1), nil,
     (LDF (DUM NIL LDF
        (LDC 0 LD (1.1) EQ SEL
            (LDC 1 JOIN)
            (NIL LD (1.2) LD (1.1) MPY CONS
                LD (3.2) LD (1.1) SUB CONS
                LD (2.1) AP JOIN)
            RTN)
     CONS LDF
        (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
        RAP RTN)
     AP STOP)))))
⇒ (NIL LDC 1 CONS LDC 3 CONS LDF
     (DUM NIL LDF
        (LDC 0 LD (1.1) EQ SEL
            (LDC 1 JOIN)
            (NIL LD (1.2) LD (1.1) MPY CONS
                LD (3.2) LD (1.1) SUB CONS LD (2.1) AP JOIN)
            RTN)
     CONS LDF
        (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
        RAP RTN)
   AP STOP)
```

This final code is shown as a linked list of cells in Figure 7-23. For compactness, this figure takes any car field that contains a pointer to a terminal cell (tag=integer) and writes the value of that terminal where the pointer would go. In reality, the other terminal cell is still there.



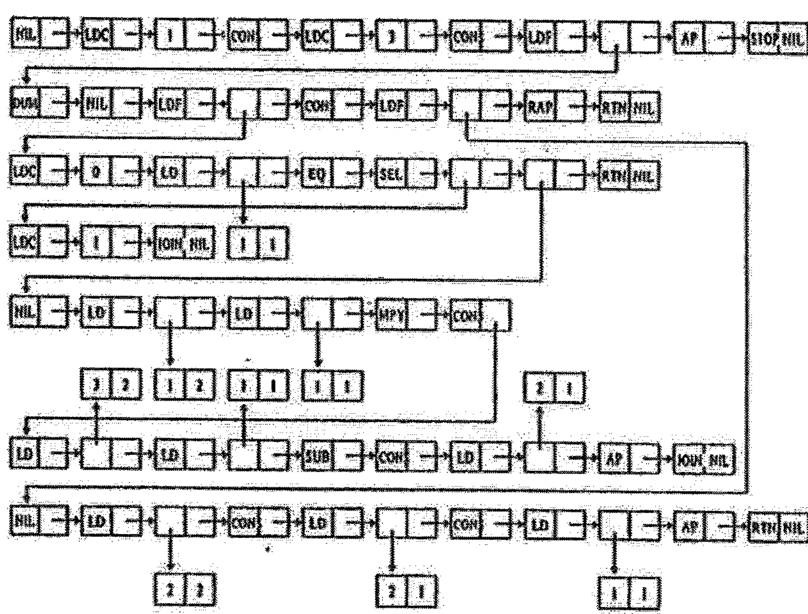


**FIGURE 7-23**
Shorthand cell equivalent of sample code.

## 7.7 THE FUNARG PROBLEM

(Bobrow and Wegbreit, 1973; Georgeff, 1982)

The SECD Machine uses stacks for almost everything. This leads to easy-to-understand compilers and relatively simple hardware implementations. It is also very similar to the abstract machines used for many conventional languages such as Pascal.

For someone schooled in conventional architectures, however, a direct implementation of the SECD Machine seems to have some significant inefficiencies. Each time an object is pushed onto the S stack, for example, the F register is incremented and two writes to the cell are performed (one for the data in the car and one for the cdr pointer). This work is doubled if the object being pushed is a new one, for example, the result of an ADD instruction.

In contrast, a conventional implementation of a stack involves simply incrementing a stack pointer and writing the data to memory. This is far cheaper in terms of machine cycles than the above procedure.

Even worse is the list of lists found on the environment. Accessing an argument via a LD instruction requires a doubly nested count down through these lists. A more normal implementation, such as for Pascal, would use a stack of contiguous locations for arguments and often a *display* of pointers into this stack to locate the beginning of an argument list

267

for each function call (see Figure 7-24). With such an implementation, accessing an argument takes only two indexed memory accesses (even fewer if the display is implemented as an array of registers inside the CPU).

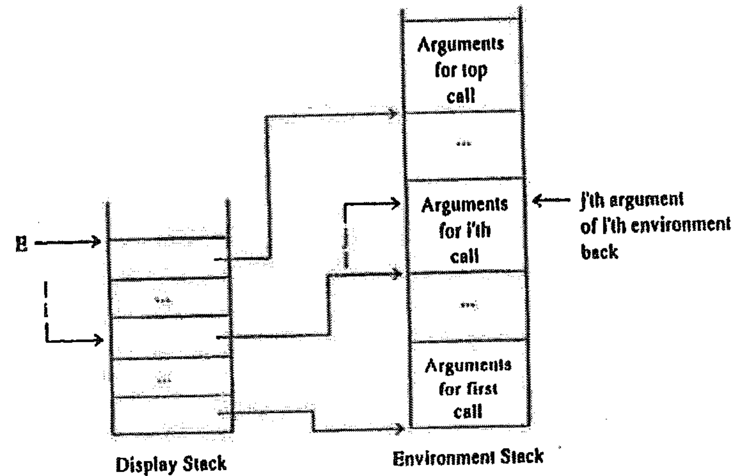
Given this, why would anyone use the SECD method of building stacks from linked lists? A good part of the answer stems from consideration of what happens when closures are generated by one function and passed as arguments to others, and particularly when identifiers within the closure function code have been bound values by functions which called the original closure-generating function. The handling of the stack and the environment becomes crucial, and if care is not taken, all kinds of strange results can occur. This has been studied extensively, and goes by the name of the *funarg problem.*

Consider, for example, the abstract program:

let f=(let $y$=4 in (λz|y×z)) in f(3)

The innermost let builds a valuelist ((4)). The result it passes to the outer let is a closure with a function code equivalent to (λz|y×z) and an environment ((y.4)).

Consider what would happen if we built stacks, particularly the environment stack, as is done in conventional machines. Pushing an item simply causes an increment of the stack pointer and a store into the designated memory cell. Whatever is there from before is overwritten. Popping an object simply decrements the pointer, permitting the next push to overwrite the prior value. In such an implementation, at the exit from the




Note: Both stacks implemented in sequential memory cells.

**FIGURE 7-24**
An alternative to a list environment.



code for the innermost let, the equivalent of the E register would be decremented. Now the code for the outermost let builds a new environment (a closure for *f*) *in exactly the same* memory cells as ((4)) took. The value "4" is lost, and the references to *y* inside the code for the inner let will pick up the closure and not the proper value for *y*. The value 4 is "lost."

This is the funarg problem, and it can occur whenever a function which produces a function result of some sort contains ***free variables*** which are given values by calling functions.

Note what happens in this example with our linked-list implementation for stacks. The valuelist ((4)) is not overwritten by the outer code. Instead the machine allocates new storage for the new valuelist for *f*. All pointers in this closure to the old environment thus remain valid, meaning that executing the code in the closure will retrieve the proper value of 4 for the variable *y*.

Keeping linked lists that are not overwritten is not the only solution to the funarg problem. Other solutions include separate heaps for storing environments when potential problems might occur, implementing duplicate environment stacks (see Harrison, 1982), ***spaghetti stacks*** or ***cactus stacks*** instead of conventional stacks (see Bobrow and Wegbreit, 1973), or even doing explicit code copying and substituting argument values as was done in our very early lambda calculus interpreters. A later chapter on real machines for functional languages will amplify on some of these.

## 7.8 OPTIMIZATIONS

There are quite a few optimizations that can be made to this basic SECD architecture and compiler that would permit faster execution of programs on it. Some of these, such as recovering memory cells that have been allocated but are no longer needed, will be discussed in the next chapter. Others, however, deserve at least a brief mention here. The interested reader is encouraged to consider how these optimizations might be reflected in either the SECD architecture, the compiler, or both.

### 7.8.1 Improved Constant Handling

As currently defined, lists that are made up of constants are painstakingly built up one at a time through a series of LDC and CONSs every time they are needed in the program. This could be considerably simplified by having the compiler check each s-expression before it compiles code for it. If the s-expression is made up of nothing but constants, the compiler could generate a single LDC followed by the unevaluated list (with numbers converted into internal representation, of course). Executing this would result in a pointer to the appropriate list being pushed to the stack without excessive make-work.

992

This would also be the technique used to implement the *quote* function found in many functional languages.

### 7.8.2 Simplifying Conditionals

When compiled by the previous compiler, most functions (particularly recursive ones) have an outermost structure of the form:

((basis test) SEL ((basis case) JOIN) ((recursion) JOIN) RTN)

The SEL pushes the cell address of the RTN onto the dump. Whichever case is selected will then execute, with a final JOIN to pop the appropriate return address from the stack. In the case of code sequences like the one above, this return takes the program to a RTN which then pops the dump again.

In many circumstances this double pop can be avoided by replacing the JOINs by RTNs, and replacing SEL by an instruction which does a similar selection function but does not push anything to the dump. Such an instruction might be called a *TEST* and operate something like:

TEST: (x.s) e (TEST ct.c) d→s e c? d where c?=ct if x ≠ 0 and c?=c if x=0

Note that the false code need not be a separate sublist, but simply the continuation of the code after the TEST.

With this instruction the above typical code structure can be expressed as:

((basis test) TEST ((basis case) RTN) (recursion) RTN)

### 7.8.3 Simplifying Apply Sequences

***Tail recursion*** occurs when the last thing a function does is call some other function with a new set of arguments. A very common piece of code generated by the above compiler for such circumstances looks like (...AP RTN). When executed, the AP pushes values for S, E, and C onto the dump. The return from the function called by the closure invoked by AP will pop these values off the dump and reestablish them in the proper registers, only to have the same values overwritten by the RTN following the AP. Three extra pushes and pops to the dump have been performed, with no useful effect.

An interesting extension to the SECD architecture that avoids this double dump pop might take the form of a *DAP* (for *Direct APply*) instruction. This instruction would have a register transition that performs only the environment modifications from the AP and leaves the dump alone:

DAP:((f.e') v.s) e (DAP c) d→NIL (v.e') f d



Thus the code sequence (...AP RTN) would be replaced by the much more efficient (...DAP). No extra items are pushed to the dump, and we rely on the RTN instruction at the end of the function called by DAP to return control to the function which called the code containing the DAP, with no intermediate stops. In a sense we have short-circuited the intermediate call/return.

### 7.8.4 Avoiding Extra Closure Construction

While DAP avoids the double dump push/pop, that is not the end of the line for optimizations. Consider the very common sequence:

(...LDF (...function code...RTN) AP RTN)

DAP optimizes this to

(...LDF (...function code...RTN) DAP)

The LDF builds a closure that is immediately taken apart by DAP. The environment modified by the DAP is the same one that is already in effect.

Consider instead what would happen if we invent a new instruction *AA* (*Add Arguments*) with the following register transitions:

AA: (v.s) e (AA.c) d→s (v.e) c d

Now the above code sequence could be replaced by:

(...AA...function code...RTN)

We have avoided the extraneous closure-building of the prior sequence. Further, on machines where branches are expensive (as is the case for most high-performance machines), we have also eliminated the change in the C register still present in the DAP form.

### 7.8.5 Full Tail-Recursion Optimization

An astute reader might detect that there is one further optimization that could be performed for tail recursion. In all the above cases we are still consing a new argument list onto the current environment, the top of which is a sublist of arguments for the current function call. In most cases these latter arguments will never be referenced again, so the storage associated with them becomes wasted space. If we could avoid leaving these unused arguments on the environment, we could open up the possibility of recovering the storage.

An expansion to AP (or AA) that avoids this growth would "re-

E97