J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

PROGRAM INTERPRETERS

The computer is a powerful tool, capable of doing any calculation that can be specified by an algorithm. However, a computer's *machine language*, consisting of the instructions that can be executed without any processing by other software, contains only relatively primitive operations. A program written in machine language can directly drive the computer's hardware components, because each machine language instruction can be directly represented as a pattern of low and high voltages that, when applied to the computer's central processing unit (*CPU*), causes the specified computation to occur. When a machine language program is represented textually, the pattern of low and high voltages is written as a sequence of 0s and 1s interpreted as a binary number. For human readability, other bases such as octal, decimal, or hexadecimal are also used for writing the numbers corresponding to the pattern of voltages in a machine language instruction.

Development of large programs in machine language is a difficult task, due to the length of the programs and the complexity of the languages' semantics. Ideally, humans desiring to program a computer would like to express their algorithms in a natural language such as English or Chinese. The problems of using natural language to command a computer include the intrinsic ambiguity and complexity of such languages. To overcome the ambiguity and complexity of natural languages while avoiding the difficulties of machine language programming, many high level programming languages have been developed. These include Ada, Basic, C, C++, Fortran, Lisp, Java, and Pascal, as well as many others. However, computer hardware cannot directly accept commands in these forms. There exists a semantic gap between computer and human.

There are two fundamental techniques used to bridge this gap. One technique involves statically translating a program written in a high level language into an executable program in machine language, then running the resulting machine language program. The original program is often called the *source code* and the executable program is called the *machine code*. This technique is known as program compilation. For a given source program, the translation or compilation is done one time only, resulting in a machine language program that can be executed as often as necessary. The second technique, called interpretation, dynamically translates the source program into machine language line-by-line as necessary in order to execute the program. No separate file of machine code is generated. In the past, an interpreter was typically written in the machine or assembly language of the computer on which the source program was to be executed to achieve high performance. However, today, the interpreter may also be written in any programming language for which there is an existing compiler or interpreter. These two techniques are shown in Fig. 1.

Machine Language and Assembly Language

A textual representation of a machine language program, with short alphabetic codes and numerals called *assembly language* replacing the patterns of 0s and 1s, is often used to increase the readability of machine language programs for human programmers. A program called an *assembler* translates a text file of assembly language into the corresponding machine language program. An assembly language can be considered a direct mapping of the machine language into a language that is easier for humans to comprehend. There exists a

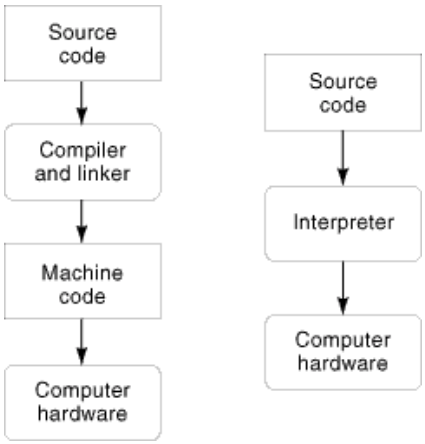

Fig. 1. Compiler and interpreter techniques.

```
Assembly Lang.   Machine   Explanation
Instruction      Code
**************   *******   ***********
mov dl, 54h      B2 54     ;move 54 (hexadecimal) to register dl
mov ah, 40h      B4 40     ;move 40 (hexadecimal) to register ah
```

Fig. 2. Intel 80x86 assembly language and machine code.

one-to-one correspondence between an assembly language instruction and its machine code. Fig. 2 shows a segment of an Intel 80x86 assembly language program and its machine code.

Assembly language is normally the lowest-level programming language that a programmer would use. The translation from an assembly language program to machine code is straightforward. An advantage of the assembly language is its capability of directly utilizing the available resources of the CPU, memory, and I/O systems of the computer in a very efficient manner. However, this specificity also means that the machine language of one type of computer will generally be incompatible with that of another—the sets of instructions recognized by each type of CPU are different. Moreover, writing large programs in assembly language is tedious, time-consuming, costly, and error-prone.

Interpreter

In the simplest form of language interpretation, the source code is executed statement by statement (or appropriate program segment). For each statement in the source program, an appropriate procedure will be called to interpret the meaning of that statement. In this process a typical interpreter goes through the following fetch-evaluate-execution cycle:

(1) Fetch the current statement suitable for execution from the source program
(2) Identify the corresponding procedure to be called for the statement
(3) Evaluate the parameters to be passed to the procedure
(4) Pass execution control to the procedure along with the parameters
(5) Store execution results at appropriate locations for subsequent access
(6) Move to the next statement in the source code and go back to step 1

```
INPUT    A, 002     ;Binary input data from port 002 is stored in A
AND      A, 024     ;Data in A is logically anded with input
                    ;   from port 024, the result is stored back to A
OUTPUT   A, 101     ;Data in A is sent to output port 101
```

Fig. 3. Process control program.

Figure 3 is a segment of a simple sequential process control program. In this example, the interpreter will execute procedures corresponding to INPUT, AND, and OUTPUT statements in sequence. Before the execution is passed to the individual procedures, the associated memory and I/O addresses for A and input/output points must be determined. An interpreter for this purpose can be easily written in any assembly language.

From this example, one can see that there is a clear correspondence between the source code and the actions of the interpreter. This feature facilitates debugging of the source code. For example, a break point can be set at any point in the source program. However, because an interpreter translates only one statement at a time, execution of a given source program is normally much slower than execution of the machine language program that could be produced from the same source code by a compiler.

However, the interactive nature of the development environment that results from this statement-by-statement method of translation makes program development much easier than in a batch-oriented environment in which a program must be edited, compiled, and submitted for execution each time changes are made.

Many conventional programming languages have been implemented by interpreters. The most popular interpreted languages include BASIC, APL, Lisp, and Prolog, as well as many other functional and logic-programming languages. The interactive development environment is a common point between these languages.

Another approach to bridging the semantic gap between computer and human programmer is to combine the compilation and the interpretation techniques. In this approach, the source code is first compiled into an intermediate language that is similar to a machine language, but is independent of any particular computer architecture. The intermediate language program is then executed through an interpreter that translates the instructions of the intermediate language into particular machine language instructions for the processor on which the interpreter is running. Such an interpreter is referred to as a *hybrid* or *bytecode* interpreter.

Interpreters have been used since the 1960s to specify the behavior of languages, even when typical implementations of the language were compiled rather than interpreted.

Finally, interpreters from one machine language to another have been used to allow the running of machine language programs written for one architecture to run on another. Typically, the motivation here is economic: users may be more likely to purchase new, faster computers if they can continue to run the software they have developed on their current computer.

Development and Practice

Pure Interpreters.

The Beginning: LISP. Perhaps the earliest interpreter was the original LISP system developed at MIT in 1956–60 by John McCarthy and others. *LISP* stands for list processor. The first widely distributed version, LISP 1.5 (1), programmed by Stephen Russell and Daniel Edwards, became available in 1962. LISP originally ran on an IBM 704 computer.

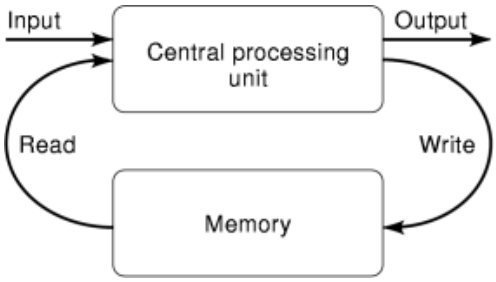

Fig. 4. The Von Neumann computer model.

McCarthy's goal was to develop a language that could be used for symbolic computations (such as nonnumeric tasks—see related article on List processing), a task for which the most popular existing high-level computer language, FORTRAN, was unsuited. The LISP developers were also influenced by an existing symbolic language, Information Processing Language (*IPL*), but considered IPL too close to assembly language to meet their needs. The intention of the LISP developers was to create the language as a tool for Artificial intelligence (see related article) research. Another influence on LISP was the lambda calculus, developed in the 1930s by Alonzo Church (2), with whose work McCarthy was familiar. (See the following section on functional languages for a discussion of lambda calculus). Many different dialects of LISP have been developed subsequently, including CommonLisp, FranzLisp, and the statically scoped dialects T and Scheme.

LISP is an example of a language that possesses some (though arguably not all) characteristics of a functional language. A functional language is perhaps best explained by contrasting it to an imperative language. Imperative languages are those in which a program consists of a collection of directions about how the state of the machine should be modified. Such modifications of a machine's state typically take the form of instructions about how to change the contents of memory. Examples of imperative languages are FORTRAN, C, Pascal, Ada, Modula 3, and ALGOL. The assignment statement in a language such as FORTRAN is an example of an imperative instruction. A statement such as

$$X = Y + Z$$

instructs the machine to load the contents of variable Y from the appropriate location in its memory (the contents of memory define the state of the machine), load the contents of the variable Z, compute the sum of those two values, and store it in the memory location corresponding to variable X. This model of computation is very natural when the underlying computer architecture is that of a Von Neumann machine (see Fig. 4 and Von Neumann computers), a single processor accessing a memory consisting of a one-dimensional array of storage locations, but it might be argued that not all programming languages should be so closely tied to the underlying architecture on which they are implemented. However, virtually all popular single-processor computer systems since the 1940s have been Von Neumann style machines.

Functional languages attempt to abstract away from the notion of a program being a sequence of commands to modify the memory, or state, of a computer. Rather, in a functional language a program is represented as a function, similar to a mathematical function, that receives inputs (its arguments) and produces a result, but does not otherwise affect the state of the machine. Such a function is sometimes a called a *pure* function, and a function that does not meet that definition is called a function with *side effects*. Side effects occur when a running function modifies locations in the computer's memory other than the location where the value to be returned by the function is stored. An example of a side effect would be the modification of a global variable by a procedure.

The higher level of abstraction afforded by functional programming may make reasoning about programs easier, since a programmer only need keep track of the arguments and results of a function in order to understand it, rather than also being required to keep track of the effects on memory of all the function's instructions.

In the subset of LISP consisting of pure functions, there are only two types of data objects: atoms and lists. Furthermore, programs are merely a special type of data object—a program turns out to be a type of list. A list is written by enclosing in parentheses an enumeration of the elements of the list. For example

$$(X \quad Y \quad (A \quad B \quad C))$$

is a list containing three elements, the first two of which are atoms, and the third of which is a list—lists may be arbitrarily nested. A list has two parts: a head (the first element in the list) and a tail (the remainder of the list excluding the first element). LISP contains built-in functions CAR and CDR for extracting the head (CAR) and tail (CDR) of a list, as well as functions for constructing lists from atoms and from other lists. A function can be written as a list with three elements:

$$(LAMBDA(arg_1 \ldots arg_n)\ list)$$

The first element of the list, *LAMBDA*, is a keyword of LISP, and denotes that this list describes a function. The second element is a list of the names of the arguments to the function, and the third element is an arbitrary list that contains the body of the function (the code to be executed when the function is called). For example, the following function returns the second element of its parameter, which is assumed to be a list:

```
(LAMBDA(ARG1) (CAR (CDR ARG1)))
```

A call to this function would look like

```
(LAMBDA(ARG1) (CAR (CDR ARG1))) '(A B C)
```

and would return *B*. The apostrophe marks the final list or atom as data. It is a shorthand notation for application of a special function called QUOTE that inhibits interpretation of its arguments. Functions can be named in LISP, and then invoked by referring to the name. The following LISP code defines a function named SECOND which accomplishes the same function as the one derived above:

```
(DEFUN SECOND (ARG1) (CAR (CDR ARG1)))
```

This function can then be invoked

```
(SECOND '(A B C))
```

returning the same result *B*. Consider a slightly more complex function, this time one that reverses the order of elements in a list. It should be noted that LISP is not a strongly typed language, and thus the same function can be used to reverse the order of the elements of any list, regardless of the types of those elements.

```
(DEFUN REVERSE (L)
  (COND
   ((NULL L) NIL)
   ((NULL (CDR L)) L)
   (T (APPEND (REVERSE (CDR L))
              (CONS (CAR L) NIL)))
  )
)
```

The body of function REVERSE consists of a list whose first element is COND—a LISP keyword that directs the LISP interpreter to evaluate the first element of each subsequent element of the list beginning with cond, until one is found that evaluates to the logical value T (True). When such an expression is found, the second component of that list is evaluated. Thus, in the function above, if the function parameter L (the list to be reversed) is NULL, that is, it has no elements, the empty list NIL should be returned. If the list has exactly one element (its tail is NULL), then the list is its own reverse, so it is returned unchanged. Otherwise, we recursively reverse the tail of the list, and append the reversed tail to the list consisting of just the head of *L*. We will examine code to perform this function in a number of other interpreted languages.

One common criticism of LISP is that, due to its interpreted nature, performance of an algorithm implemented in LISP seems to be inherently worse than what the same algorithm would exhibit if implemented in an imperative language. To counter such criticisms, several computer systems known as LISP machines were developed that were optimized at the hardware level for executing LISP programs. Such machines, typically single-user workstations, began with the development efforts of Richard Greenblatt at MIT in 1974. Xerox Corporation produced several commercial models including the Alto and Dorado workstations. Other manufacturers of LISP-optimized machines included Bolt, Beranek, and Newman (*BBN*), Symbolics, and Apollo. By the late 1980s these machines were no longer economical: Higher performance was being achieved by running LISP interpreters on conventional machine architectures. Increasing performance is one of the most important research questions facing interpreted language developers today.

APL. APL was another early interpreted language, developed at IBM by Kenneth Iverson and Adam Falkoff, and first implemented in 1966 on the IBM 360 (3). The inspiration for APL came from mathematical notation. While developed to be simple and practical, APL developed a reputation for being difficult to use, due to the nonstandard character set it employed (which required a special terminal) and because unless special care was taken, even very simple programs were very difficult to understand. In spite of its mathematical roots, APL, like COBOL, was most widely used for data processing applications.

Basic. In the late 1970s, single-user Microcomputers (see related article) began to be widely used. The typical microcomputer of the era consisted of a single-chip CPU with an 8-bit data path, between 1K and 16K of main (DRAM) memory, a keyboard and screen, and a disk drive, paper tape reader, or cassette tape device for storing programs off-line. The system software consisted of a simple disk operating system such as CP/M, or perhaps simply an interpreter (in ROM) for the BASIC programming language. *BASIC* (Beginners All-Purpose Symbolic Instruction Code) was developed approximately a decade earlier (1966–7) by John Kemeny and Thomas Kurtz at Dartmouth for use on large time-sharing computers (4). Their goal was to develop as user-friendly a language as possible, with which a novice programmer could make use of a computer without the frustrations attendant to using FORTRAN. Their overriding design principle, fairly radical for the era,

was that BASIC should be a language in which software could be developed easily and quickly: for the first time, the programmer's time was more important than the computer's time. The interactive, interpreted nature of BASIC contributed greatly to its ease of use. A FORTRAN programmer of the mid-1960s would typically prepare a program using an offline device such as a paper tape writer or punch card printer. Then the user would submit the tape or card deck to an operator, wait for it to be processed, and retrieve output in the form of tapes or printouts from the operator. In contrast, a BASIC programmer could run a program from an interactive terminal as soon as it was written, make modifications from the terminal, rerun the program, and continue development at a much more rapid pace.

BASIC was particularly attractive for microcomputers because of the small memory and performance requirements it needed. Whereas it would be very difficult to implement a FORTRAN or COBOL compiler in the small memories of early microcomputers, BASIC would fit easily. In 1976, Paul Allen and Bill Gates (the founders of Microsoft) wrote a BASIC interpreter in assembly language for the Intel 8080 processor used by the MITS Altair microcomputer. The machine language code for their interpreter would fit in only 4K of RAM, and hence could be used on many 8080-based microcomputers with little modification. Virtually every microcomputer supported a dialect of BASIC, ranging from very small subsets of Kemeny and Kurtz's BASIC to supersets that included operating system functions accessible from the interactive interpreter.

Without a doubt, for quite a while (perhaps lasting even to the present day) BASIC was the most widely used programming language in the microcomputer community, and enjoyed wide usage in science, engineering and business well into the 1980s, due to the ease with which a nonspecialist could become proficient and due to the convenience of program development. However, BASIC was not without its detractors. The very things that made BASIC easily implementable on small systems meant that BASIC did not contain such features as abstract data types and structured flow of control that were widely considered attractive for easing the task of writing large, complex programs.

In most versions of BASIC it would be impossible to code the list-reversing function we described in LISP, because BASIC does not support the rich variety of data structures that LISP does. Typical BASIC implementations may support only integer, floating point, and character scalar variables. The only data structure for which support exists is typically arrays. Some versions of BASIC developed in the late 1980s and early 1990s begin to address these deficiencies of BASIC. An example of such a version is Microsoft's Visual Basic. However, we can write a BASIC program to read a sequence of non-negative integers, store them in an array, reverse the sequence, and then print the contents of the array:

```
5 N = 0
7 REM INPUT THE ARRAY OF INTEGERS
10 N = N + 1
20 INPUT A(N)
30 IF A(N) < 0 THEN GOTO 35 ELSE GOTO 10
35 REM REVERSE THE ARRAY
40 FOR I = 1 TO N-1
45 REM SWAP A(I) AND A(N-I+1)
50 T = A(I)
60 A(I) = A(N-I+1)
70 A(N-I+1) = T
80 NEXT I
90 FOR I = 1 TO N-1
100 PRINT A(I)
110 NEXT I
```

Here, line 5 initializes an index variable *N*; lines 10 to 30 accomplish reading the integers from the keyboard and entering them into the array; lines 33 to 80 reverse the array; and lines 90 to 110 print the reversed array. Note that every line requires a line number, that the extent of the statements to be repeated in the body of a FOR loop is terminated by a NEXT statement, and that lines beginning with REM are treated as comments and ignored by the interpreter.

The imperative nature of BASIC can be seen from this example program. The flow of control is ordered by the line numbers, which order the execution of the statements sequentially, unless the programmer explicitly requests a transfer of control via a GOTO statement, conditional (IF . . .ˆTHEN) statement or FOR loop. The other statements direct the memory of the computer to be modified in some fashion.

A user would enter such a program at the terminal or console of a computer running a BASIC interpreter by typing the lines exactly as written above. If a mistake were made, a line could be corrected by typing a new line beginning with the same number, which would overwrite the erroneous line. A number of commands typically exist for program management:

(1) RUN—execute the program currently in memory
(2) NEW—delete the program currently in memory
(3) LIST—print the lines of the program currently in memory on the screen
(4) SAVE—store the program currently in memory to tape or disk
(5) LOAD—fetch a program stored on tape or disk to memory

Functional Programming. Functional programming languages are another large class of languages typically implemented with interpreters. LISP, discussed above, shares many of the characteristics of functional languages, but was developed long before the class of functional languages was defined, in James Backus's Turing Award lecture (5), *Can Programming Be Liberated from Its Von Neumann Style*, in which he outlined the characteristics of functional languages, described the benefits to be obtained by programming in a functional language, and defined a functional language FP.

The aim of functional languages is to make reasoning about programs and the development of correct programs easier, by eliminating the need for the programmer to think of execution of the program under development as a sequence of modifications to state of the machine. Rather, a functional program resembles a mathematical function: Given certain inputs, the program will produce the specified output without otherwise modifying the state of the computer. These modifications, or side effects, seem particularly difficult to keep track of, and can often be responsible for hard-to-find bugs.

The syntax of functional languages is largely derived from Church's lambda calculus formalism (2). Lambda calculus is a notation for expressing computation as a series of function applications. There are only three syntactic constructs in lambda calculus:

(1) *Identifiers,* or variables, for naming data and functions
(2) *Functions,* consisting of a function symbol, argument, and a lambda calculus expression to be executed when the function is called
(3) *Applications,* of functions to arguments

In particular, there are no assignment statements that explicitly modify the state of the computer. The only variables present are arguments to functions, which are local to the function in which they are defined. Control does not flow from one statement to the next, executing each sequentially, as in the imperative languages. Rather, one function calls another when necessary. Looping can be implemented by recursion: a function may call itself.

Such languages are characterized by features such as

(1) Simple syntax
(2) Rich type systems, including polymorphic typing
(3) Lazy, or on-demand evaluation of functions
(4) Advanced pattern-matching features for use in function definitions

Functional languages have typically been interpreted rather than compiled, probably because LISP has historically been interpreted. Even when compiled, however, execution of functional languages on conventional Von Neumann computers has proven to be slower than execution of imperative programs implementing the same algorithms.

Many functional languages exist, including ML, Haskell, and Miranda. Use of functional languages has mostly centered around academic projects. Functional languages have not achieved wide popularity outside academia.

Logic Programming. Another family of languages typically implemented by interpreters are the logic programming languages. The first such language, Prolog, was developed by Alain Colmerauer and Philippe Roussel to aid in research in the processing of natural languages (6).

However, Prolog and other logic programming languages have become popular for use in many other artificial intelligence and knowledge engineering applications. Logic programming languages are members of the class of declarative programming languages. In contrast to the above-mentioned class of imperative programming languages, which consist of sequences of commands or instructions describing steps the machine should carry out to modify its state, declarative languages allow the programmer to state a problem (hence the name) without specifying a particular algorithm to solve the problem.

Programs in Prolog or other logic programming languages resemble a collection of sentences in a Formal logic (see related article) rather than a collection of functions or procedures. A logic program is executed when the machine is given a logical formula whose truth or falsehood is to be determined, based on the totality of other logical facts and rules that have been entered—these facts and rules constitute the logic program.

It is well-known that any sufficiently powerful mathematical logic is undecidable; that is, no decision procedure exists that can, for every formula in the logic, determine its truth or falsehood. Further, even if one restricts the logical language sufficiently to make all theorems of the language decidable, the computational complexity of the task often remains very high—sometimes hyperexponential. Thus, the logic available to programmers using logic programming languages must be restricted quite severely to achieve reasonable execution.

First let us consider a simple Prolog program to deduce family relationships between a group of individuals. A predicate is a function of several arguments that returns a Boolean value (true or false). We can define a number of relationships between individuals by logical predicates, such as father(X,Y), mother(X,Y), or sibling(X,Y). We will define father(X,Y) to mean that *"X is the father of Y."* Prolog allows us to enter facts about whether this predicate holds for particular individuals as part of our logic program. The following Prolog fragment consists of a sequence of such facts:

```
father(wolfgang,fritz).
father(fritz,leonhard).
mother(anna,leonhard).
sibling(anna,johanna).
sibling(anna,ulrike).
```

After entering this program, the Prolog interpreter could be queried regarding whether certain relationships held. We could ask whether wolfgang was the father of fritz, and the interpreter would reply that this was the case.

```
| ?- father(wolfgang,fritz).

yes

| ?-
```

If we had asked whether a relation held that in fact did not hold, the response would look like

```
| ?- father(wolfgang,karl).

no

| ?-
```

We could replace the name of one of the arguments with a variable (variable names in prolog are denoted by identifiers beginning with a capital letter), and the interpreter would tell us all possible instantiations of that variable that made the predicate true:

```
| ?- sibling(anna,X).
X = johanna
X = ulrike
no
| ?-
```

In addition to defining simple predicates as facts, and querying the database, Prolog allows one to define complex relationships such as grandfather by combining predicates. Logically, an individual X is the grandfather of an individual Y if there is an individual Z such that X is the father of Y, and Y is the father of Z. Prolog allows us to code this sort of a relationship as an inference rule that allows new facts to be deduced from known facts. We can code the grandfather relationship with the following Prolog statement:

```
grandfather(X,Y) :- father(X,Z),father(Z,Y).
```

One reads a Prolog inference rule, or *clause* such as this one in the following manner. The part of the clause to the left of the ':–' symbol must be true whenever all the predicates to the right of the ':–'symbol are true. If you think of the left-hand side as the *goal* of the program, then the predicates on the right hand side are *subgoals*, such that if all the subgoals are satisfied, the goal is guaranteed to be satisfied. This form of reasoning is known as backward chaining. The Prolog interpreter works backward from the goal it wishes to prove by

trying to satisfy the subgoals, each of which may have subgoals of its own, and so on, until finally each subgoal is satisfied by some fact in the Prolog interpreter's database of known facts.

To continue with our relationships example, if a Prolog interpreter is given the facts listed above and also the grandfather inference rule, by backward chaining it can deduce

```
grandfather(wolfgang,leonhard).
```

Prolog restricts the form of inference rules to Horn clauses, which have the logical form

$$(subgoal_1(..) \wedge subgoal_2(..) \wedge \ldots \wedge subgoal_k(..)) \Rightarrow p(arg_1, \ldots arg_n)$$

If each of the predicates $subgoal_i(..)$ is true, then the consequent of the implication, $p(arg_1, \ldots, arg_n)$ must be true also. A Horn clause of that form directly translates to a Prolog inference rule

```
p(arg1,...,argn) :- subgoal1(..),subgoal2(..),
...,subgoalk(..).
```

One of the biggest shortcomings of Prolog as an implementation of mathematical logic is its treatment of negation. The negation of a prolog clause P is found to be true whenever the clause P cannot be proved. This is referred to as "negation as failure." Anything that the Prolog interpreter cannot derive is assumed to be false. The problem arises due to the nature of the Horn clause. While it is logically sound to infer that p must be true if all the $subgoal_i$ are true, it is not legitimate to conclude that p must be false if some $subgoal_i$ cannot be proven.

The list reversal example we examined in LISP, a functional language, and BASIC, an imperative language, appears as follows in Prolog. We define a predicate reverse(X,Y) that we wish to be true when Y is a list that is the reversal of X. We can compute the reversal of a list, say the list [1,2,3,4], by asking the Prolog interpreter to prove the goal

```
reverse([1,2,3,4],X)
```

Prolog will find an instantiation of the variable X that satisfies the goal—it will compute the reverse of the list given as the first argument. The code looks like

```
reverse([], []).
reverse([Singleton],[Singleton]).
reverse([Head|Tail],Reversed) :-
  reverse(Tail,Rtail),
  append(Rtail,[Head],Reversed).

append([],L,L).
append([H|T],L,[H|T1]) :- append(T,L,T1).
```

We define first two facts: The reverse of the empty list is the empty list, and the reversal of a list with one element is the same list. The third line defines the reverse of a list with head and a non-empty tail as the reverse of the tail, appended to the head. Since Prolog does not have a function to append one list to another, we define a predicate append(X,Y,Z), which is true when Z is the list obtained by appending X to Y. Append is defined with two rules: When you append the empty list to a list L you get the list L, and when you append a list with a head H and tail T to a list L, you get the list you get when you append H to the list you get when you append T to L.

Writing a logic program is a very different activity than writing an imperative program. Rather than explicitly specifying function calls and flow of control, one uses the backward chaining inference process to accomplish computation. This necessitates a different way of thinking about problem solving and programming.

Abstract Machines: Interpreters as Specification Tools. Interpreters have long been used as tools for formally defining programming languages (even for languages which are typically implemented using compilers rather than interpreters) because of the clarity with which the interpreter can be written and understood. This task of defining a programming language by giving an algorithm for constructing the meaning of programs in the language is known as providing a *semantics* for the programming language. Much as philosophers might discuss the semantics of a construct in a natural, human language, one can discuss the semantics of programming language constructs as well.

The simplest form of semantics one might give for a computer language L would be something like, "The meaning of a program P in language L is what happens when program P is compiled with compiler C and run on machine M." The problem with this approach is that the meaning of the language is now tied to the particular implementation C of the compiler, and to the particular features of machine M. Suppose that bugs are discovered in the compiler, or even in the machine itself. The language definition should not incorporate those bugs. Thus, we want to define the meaning of programs in a language in a way that does not depend on any particular implementation of that language.

One early attempt to do this for a language similar to the lambda calculus was Landin's SECD machine (7). Landin created a programming language called *applicative expressions* which is similar to lambda calculus. The language was defined by describing an abstract machine and giving rules for evaluating applicative expressions using the machine as an interpreter. This sort of abstract machine has been a favored computer science formalism since the 1930s: Landin's inspiration for the SECD machine certainly included the Turing Machine, an abstract machine designed to explore the limits of what could be computed mechanically (that is, by an algorithmic process). Due to the simplicity of the machine, and the power of the language it can process, it is instructive to examine its abstract interpreter in detail, since nearly every interpreter follows its structure to some extent. A detailed examination is provided in the last section of this article.

Bytecode (Hybrid) Interpreters. As mentioned in the introduction to this article, one of the biggest drawbacks to using pure interpreters is the slow execution speed of interpreted programs. This is because the interpreter must expend quite a lot of effort translating the source code to executable machine instructions at run-time. The situation becomes worse for iterative code such as loops or recursive programs, because the interpreter will waste time retranslating the same source code multiple times. The same computation, coded as a program for a pure interpreter, may run perhaps 200 times more slowly than that computation would if a compiler were used. Hence, a compromise that is sometimes employed is the use of a hybrid interpreter. A hybrid interpreter first compiles the source code to an intermediate representation, sometimes called a bytecode, which can be rapidly translated to many different computers' machine codes. The bytecode is then translated to machine instructions by an interpreter-like program. The two approaches are compared in Fig. 5.

Run-time interpretation of a program represented as bytecode may result in only a slowdown factor of two or three, versus perhaps the factor of several hundred that a pure interpreter might incur. This approach gives a performance increase because most of the work of translating the source code to a low-level machine-code-like language is accomplished by the first (compiling) step, leaving very little work for the run-time interpreter.

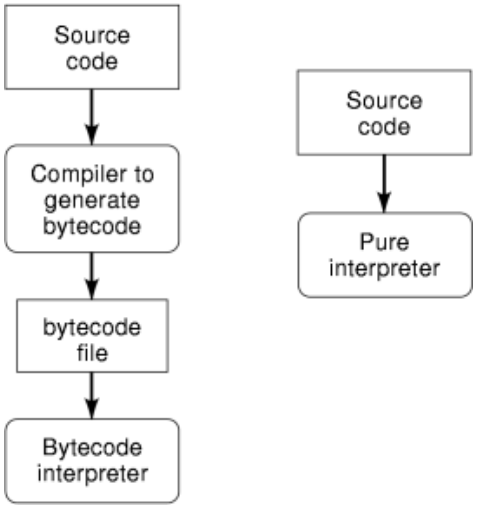

Fig. 5. Hybrid versus pure interpreters.

In addition to the performance increase, the use of a hybrid interpreter leads to greater portability of programs that can be run using the hybrid interpreter. The bytecode compiler will perform the same translation regardless of the underlying architecture on which the program will be actually run. Thus, to port the language to a new computer architecture, we need only implement a bytecode interpreter for the new architecture—we can reuse the program that compiles the source code to bytecodes.

Such an approach is not new. Compiler writers have searched for an intermediate bytecode format that would be appropriate for many languages and many computer architectures for many years, beginning with *UNCOL*, or UNiversal COmpiler Language. Such research has met with mixed success. The differences between architectures, and between different source languages, have proven great enough to prevent the effective definition of a single intermediate form.

One intermediate form that was widely used for a number of years in the 1970s was the UCSD P-system (8), a bytecode format that was used to port the Pascal programming language to many early microcomputers. Pascal programs were compiled to a format called P-code, which was then interpreted at run-time.

Many bytecode interpreters exist because of the needs of users to run programs written for other architectures. A member of this class of bytecode interpreters is also called an *emulator* or *object-code translator* (*OCT*). This type of bytecode interpreter takes as its input a machine code file of instructions for one processor and operating system, and dynamically translates it to run on another processor and operating system. The main motivation for this sort of bytecode interpreter is economic. Users can continue to use the code they have. Their reasons for doing this may be that

(1) The new computer offers performance increase, even considering the overhead of running the bytecode interpreter.
(2) The old computer is obsolete, or broken, or no longer serviceable.
(3) The cost of porting the application to run in native mode on the new computer is excessively high, or not possible.

Current commercial examples of bytecode interpreters that translate one machine language to another include

(1) Sun Microsystem's WABI runs Intel 80x86 applications on Sun SPARC-based workstations running the Solaris operating system.
(2) Digital Equipment Corporation's FX!32 provides the capability to execute Intel 80x86 applications written for Microsoft's Windows (Win32) operating system on Digital's Alpha microprocessor. FX!32 consists of a runtime environment and a binary translator.
(3) Apple Computer's Mac Application Environment allows software developed for its Macintosh computers, which use the Motorola/IBM PowerPC processor, to run on Sun and Hewlett-Packard Unix workstations, which use Sparc or PA-Risc processors (respectively).

Java. One bytecode-interpreted language whose popularity is rapidly increasing is Java (see Java, Javascript, Hot Java), developed by Sun Microsystems (9,10). While Java was developed as a language for developing portable, real-time, embedded applications such as those found in consumer electronics, Java has achieved popularity as a language for Network computing (see related article). Java programs are compiled to a bytecode format that is the machine language of an abstract computer architecture called the *Java virtual machine*. At the time of Java's release, no machine existed that could run Java virtual machine bytecode as its machine language—a bytecode interpreter was required for each computer architecture on which Java bytecodes were to be run. Thus, the same Java bytecode file could be executed on several different computer architectures. In mid 1997, several corporate development efforts to build microprocessors that run Java virtual machine bytecodes are underway.

Developers who program an application in Java are free from many of the worries related to developing an application to run on a number of different hardware and operating system platforms. In theory, any Java bytecode program would run on any machine's bytecode interpreter in the same way. One early application area for Java has been use in programs that are embedded in World Wide Web (see Internet Technology) pages to provide animation, graphics, or computational services across the Internet These programs are called *applets*. A client machine could connect to a World Wide Web server on which the applet resided, download the bytecode, and run it on the client machine without worrying about compatibility problems between the architecture of the server and that of the client. However, as with any bytecode interpreted language, there is a performance penalty associated with running a Java program versus running a compiled program in an imperative language implementing the same algorithm.

Java is an object-oriented language similar in syntax to C++, a popular compiled object-oriented language. The Java virtual machine provides "hardware" support for such features as dynamic method resolution. Additionally, the bytecode interpreter implements security features intended to prevent an applet from gaining undesired access to the computers which download and run it. The Java interpreter is small—about 40K for Sun Sparc computers, and even when basic standard libraries and thread support are included, the size of the run-time system only goes to 175K (9).

Other Interpreter-based Tools. Many other languages and tools have been implemented using interpretation for language translation. In general, when use of an interactive development environment to speed program development has been deemed more important than performance of the program to be developed, interpreters have been employed. Other categories of tools for which interpreters are often used include: shell programming languages, spreadsheets, graphical user interface builders, and symbolic math packages.

Specification and Implementation

Here we discuss the specification and implementation of an interpreter for functional languages based on the lambda calculus. While such languages may support a rich syntax, ultimately any purely functional program

can be reduced to a term of the lambda calculus. Examination of a language with a very simple syntax illustrates the important issues.

We can use a context-free grammar to define the syntax of the lambda calculus as follows. The terminal symbols in the language are the Greek letter λ, the period or dot ., and identifiers consisting of finite strings of symbols from some finite alphabet Σ.

$$\begin{aligned} Term &\rightarrow Identifier | Function | Application | ParenTerm \\ Identifier &\rightarrow \sigma \in \Sigma^* \\ Function &\rightarrow \lambda Identifier.Term \\ Application &\rightarrow Term\ Term \\ ParenTerm &\rightarrow (Term) \end{aligned}$$

Each production corresponds to one of these syntactic constructs.

The SECD Machine. The SECD machine is an abstract interpreter for the above language. Here we examine it in detail. The name "*SECD* machine" is derived from the four components that make up the state of the machine:

(1) S (stack)—A list of intermediate values used in computation. These values could either be simple values, such as integers or floating-point numbers, or they could be the values of functions, called *closures*. The closure of a function is a 3-tuple consisting of the name of the function's argument, the applicative expression for the body of the function, and an environment in which references to identifiers found in the course of evaluating the body are resolved.

(2) E (environment)—An updateable function mapping identifiers to their current values. Values can be of primitive types, but identifiers can denote functions as well. The value of a function is a closure as described above. In the language of applicative expressions functions are *first-class objects;* that is, they can be manipulated in the same ways as any other data structures. This component of the state of the SECD machine could be thought of as its "data memory," or as a function from identifiers in a program to the values stored in them. For implementation purposes, this data structure can be thought of as list of (*name, value*) pairs.

(3) C (control)—A list of applicative expressions to be evaluated. This is the "instruction memory" which holds the currently running program.

(4) D (dump)—A 4-tuple consisting of a stack, environment, control, and dump. The dump is used when a function is called to store the calling routine's values of S, E, C, and D for use upon return from the function call.

To evaluate an applicative expression A, the SECD machine is configured initially as follows:

(1) The stack S is empty.

(2) The environment E contains only the keywords of the languages and the constants.

(3) The control C contains only the single applicative expression A whose value is desired.

(4) The dump D is empty.

Execution proceeds by transforming the current state to a new state according to a set of rules presented below. Execution continues until control is empty and the dump is empty, at which time the value of the original applicative expression can be found on top of the stack. A set of rules exist governing the behavior of the SECD machine, which tell how to compute the next state of the machine from its current state. These rules take the

form, "If condition is true, then the new value of S will be ... and the new value of E will be ..., etc." The rules are as follows:

(1) If the first item on C is an identifier, pop that item from C, look up the value of the identifier in the environment E, and place the value on top of the stack S.

(2) If the first item on C is a function with argument x and body B, pop the function from C, form its closure, (x, B, E), where E is the current value of the environment, and push that closure on to the stack S.

(3) If the first item on C is the application of a function f to an argument a, pop that item from C, push a special token ap which is distinct from all legal applicative expressions onto C, push f onto C, and push a onto C. This will cause the argument a to then be evaluated normally, resulting in the value of a being pushed onto S. Then the function f will be evaluated (its closure will be formed) and pushed onto S. Then, when the special token ap reappears on top of C, we can evaluate the function by substituting the value of the argument a, now in second position on S, into the environment in which the closure of f will be evaluated (see next rule).

(4) If the first item on C is an ap token, then we can expect a function closure (v, B, E') on top of S and an argument value $v1$ in the second position of S. The state of the machine should change according to the rule:

a. **Current State**
b. $Stack = [(v, B, E') \| v1 \| tail(tail(S))]$
c. $Environment = E$
d. $Control = [ap \| tail(C)]$
e. $Dump = D$

a. **Next State**
b. $Stack = [\,]$
c. $Environment = (v, v1) \cup E'$
d. $Control = [B]$
e. $Dump = (tail(tail(S)), E, tail(C), D)$

What happens in this case is that the previous configuration—the old values of S (minus closure and argument value), E, C (minus ap), and D—is saved in D. The body B of the function whose closure is on top of the stack is placed in C to be executed. The new environment is set to the environment E' from the closure, and amended by mapping the function's argument name v to the actual value $v1$ that is found in the second position of the stack S. The new stack is empty.

(5) If C is empty, but D is not empty, then execution of the current function has ended, and we should return from the call to the state stored when the ap token was processed.

a. **Current State**
b. $Stack = [v1 \| tail(S)]$
c. $Environment = E$
d. $Control = [\,]$
e. $Dump = (S', E', C, D')$

a. **Next State**
b. $Stack = [v1 \| S']$
c. $Environment = E'$

Step 1	Step 2	Step 3	Step 4
$S = []$	$S = []$	$S = [2]$	$S = [(x, x, []), 2]$
$E = []$	$E = []$	$E = []$	$E = []$
$C = [(\lambda x.x)2]$	$C = [2, (\lambda x.x), ap]$	$C = [(\lambda x.x), ap]$	$C = [ap]$
$D = ()$	$D = ()$	$D = ()$	$D = ()$
Step 5	**Step 6**	**Step 7**	**Step 8**
$S = []$	$S = [2]$	$S = [2]$	
$E = [(x, 2)]$	$E = []$	$E = []$	*return* (2)
$C = [x]$	$C = []$	$C = []$	*and halt*
$D = ([], [], [], ())$	$D = ([], [], [], ())$	$D = ()$	

Fig. 6. The SECD machine executing $(\lambda x.x)2$.

d. $Control = C'$
e. $Dump = D'$

In Fig. 6 the steps the SECD machine will take when executing the applicative expression $(\lambda\ x.x)2$, or the identity function applied to the constant two are shown.

Internal Representation of a Lambda Calculus Term. The interpreter's first task, *parsing* the text of the program to be interpreted, does not differ significantly from that of the parsing performed by program compilers. The text of the program is processed to build a data structure, the *abstract syntax* tree, that represents the program's derivation according to the context-free grammar by which the language's syntax is specified. Each node in the abstract syntax tree represents a non-terminal or terminal symbol used in the derivation of the concrete syntax from the start symbol of the grammar, and each edge represents the expansion of a right-hand symbol of the production by which the node at the tail of the edge was expanded. Fig. 7 depicts the abstract syntax tree for the lambda calculus term

$$((\lambda x.xx)(\lambda y.yy))(\lambda z.z)$$

In a typical implementation, the nodes of the abstract syntax tree would be implemented as records (struct's in *C*) and the edges as pointers to records.

Implementation Issues. Execution of a functional program (also called *evaluation*, since it computes the value of the program) consists mainly of applications of functions to their arguments (the arguments may themselves require evaluation before they can be passed to the function being applied to them—this is discussed in more detail below). Roughly, a function is applied to an argument by

(1) Mapping the arguments (sometimes called the actual parameters) to the names used for them in the function body (those names are called formal parameters).
(2) Evaluating the body of the function in the resulting environment.

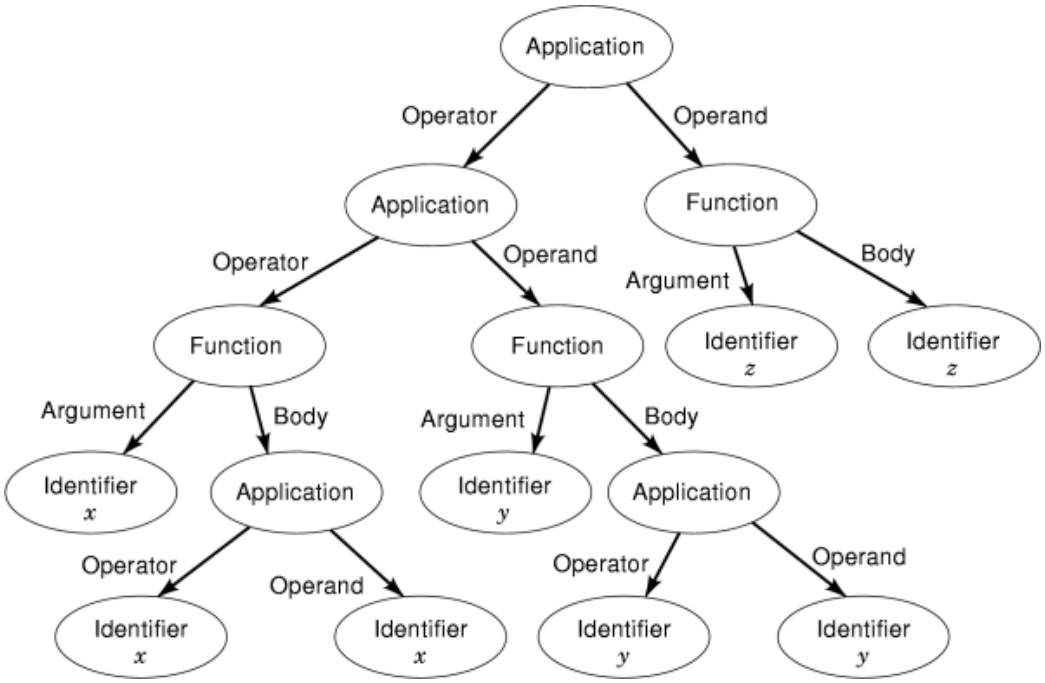

Fig. 7. Abstract syntax tree for $((\lambda x.xx)(\lambda y.yy))(\lambda z.z)$.

Note that the evaluation process is recursive: Application of a function to its argument may result in other applications of functions to arguments that can also be evaluated. When a term of lambda calculus cannot be evaluated further, it is said to be in *normal form*.

For example, consider the reduction of the lambda calculus term below. Each reduction step is indicated by the $\rightarrow$ symbol separating the term before reduction and the term derived after reduction. (See Figure 8.)

$$\begin{aligned}((\lambda x.x)(\lambda y.yy))((\lambda w.ww)v) &\rightarrow (\lambda y.yy)((\lambda w.ww)v)\\ &\rightarrow ((\lambda w.ww)v)((\lambda w.ww)v)\\ &\rightarrow (vv)(\lambda w.ww)v)\\ &\rightarrow (vv)(vv)\end{aligned}$$

In practice a number of additional concerns beyond the scope of this article complicate reduction of lambda calculus terms, including accidental capture, scope, and others. For a full treatment of the lambda calculus, consult Barendregt (11).

Evaluation Order. If an interpreter evaluates the arguments of a function before they are passed to the function, it is said to perform *eager* evaluation. Almost all widely used programming languages support eager evaluation, because it is efficient: Arguments may be used many times in the code for the function, but they are only evaluated once. The SECD interpreter presented previously in this article also does eager evaluation.

In contrast, some programming languages specify *lazy evaluation* for arguments passed to a function. Lazy evaluation schemes delay the evaluation of the arguments until the argument is actually used by the function. The example of reduction in the previous section is an example of lazy evaluation. To see an example

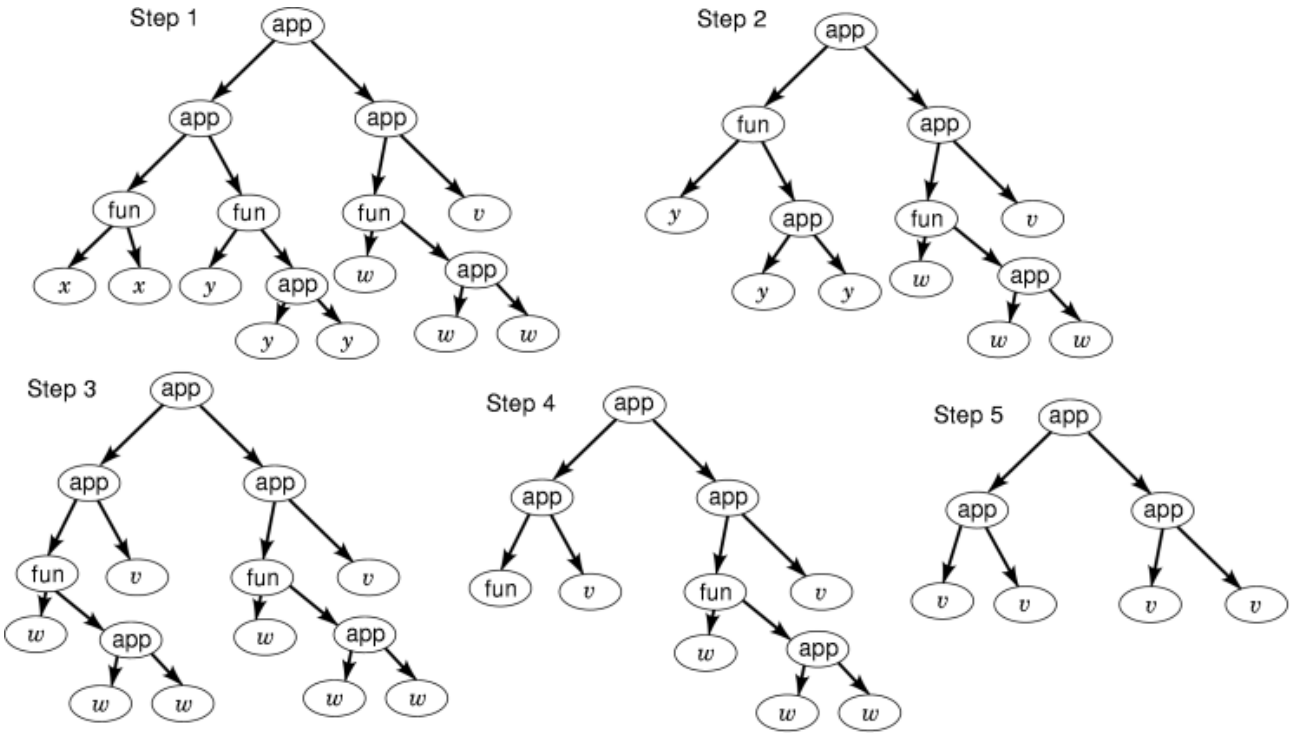

Fig. 8. Tree transformations in reduction of $((\lambda x.x)(\lambda y.yy))((\lambda w.ww)v)$.

of a function for which the results of evaluation may differ if lazy evaluation is performed, consider the term:

$$((\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz)))(\lambda w.w)$$

The first part $(\lambda x.\lambda y.y)$ of this term is a function that throws away its argument and returns the identity function. The argument to which this function is applied, $((\lambda z.zz)(\lambda z.zz))$ is one without a normal form: applying $(\lambda z.zz)$ to $(\lambda z.zz)$ reduces to $(\lambda z.zz)$ $(\lambda z.zz)$, which is the original function and argument, and which can be reduced in the same manner again, *ad infinitum*. Thus, if the above term is evaluated by an interpreter that evaluates arguments before passing them to the function that is applied to them, evaluation will never terminate. However, since the argument is never used by the function, an interpreter using lazy evaluation would return the normal form $\lambda w.w$. In fact, according to the Church–Rosser theorem, if a term of the lambda calculus can be reduced to a normal form, an interpreter using lazy evaluation is guaranteed to find that normal form.

Graph Reduction. The process of reduction may be implemented as a sequence of transformations to the abstract syntax tree. Figure 8 depicts the transformations of the abstract syntax tree during the sequence of reductions in the example above. Notice that during the reduction process that some subtrees are replicated. If, rather than replicating portions of the tree, we allow several edges to point to the same node, we can realize savings in space and reduction time—we do not need to copy replicated portions, and we do not need to perform the same reduction multiple times. If we allow this kind of sharing, the data structure representing the term being reduced is no longer a tree, but a general graph. The optimization achieved by carrying out reduction in this manner is *graph reduction*. The graph reduction of the example is shown in Fig. 9. Note the decreased

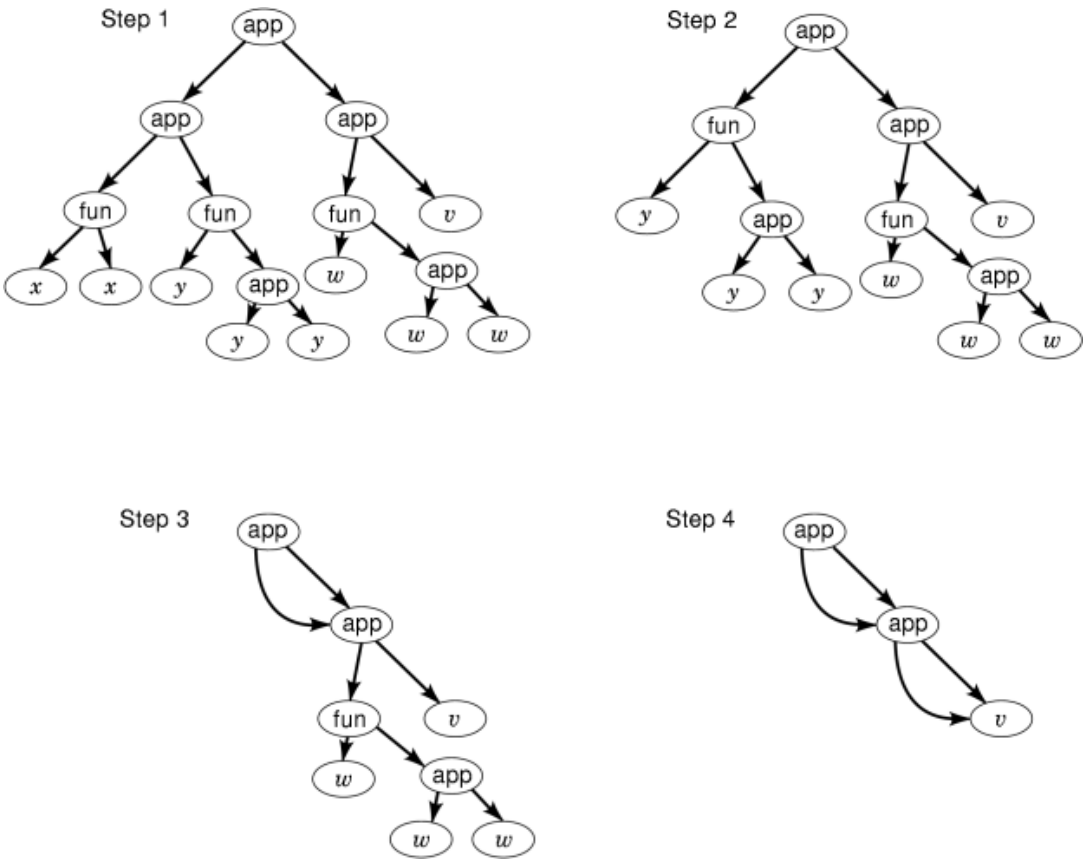

Fig. 9. Graph transformations in reduction of $((\lambda x.x)(\lambda y.yy))((\lambda w.ww)v)$.

storage requirements, and the shortened reduction sequence resulting from doing graph reduction rather than maintaining the tree. For a full treatment of optimizing techniques for interpreting functional languages using graph reduction, consult Peyton-Jones (12) or Plasmeijer and van Eekelen (13).

BIBLIOGRAPHY

1. J. McCarthy *et al.*, *LISP 1.5 Programmer's Manual*, Cambridge, MA: MIT Press, 1962.
2. A. Church, The calculi of lambda conversion, *Annals of Mathematical Studies.* **6**, 1951.
3. A. Falkoff, K. Iverson, The evolution of APL, in R. Wexelblat, ed., *History of Programming Languages*, New York: Academic Press, 1981.
4. J. Kemeny, T. Kurtz, *BASIC Programming*, New York: Wiley, 1967.
5. J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *Communications of the ACM*, **21**(8): 613–641, August 1978.
6. A. Colmerauer, P. Roussel, The birth of prolog, in *Proceedings of the Second Conference on History of Programming Languages*, New York: ACM Press, 1993.
7. P. J. Landin, The mechanical evaluation of expressions, *Computer Journal*, **6**(4): 308–320, 1964.
8. K. Bowles, *Problem Solving Using Pascal*, New York: Springer-Verlag, 1977.

9. J. Gosling, H. McGilton, The java language environment: A white paper, Sun Microsystems, 1995.
10. Sun Microsystems, The java virtual machine specification, Release 1.0 BETA, 1995.
11. H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, Amsterdam: North Holland, 1984.
12. S. Peyton Jones, *The Implementation of Functional Programming Languages*, Englewood Cliffs, NJ: Prentice-Hall, 1987.
13. R. Plasmeijer, M. van Eekelen, *Functional Programming and Parallel Graph Rewritting*, Reading, MA: Addison-Wesley, 1993.

RICHARD O. CHAPMAN
Auburn University
KAI H. CHANG
Auburn University