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.

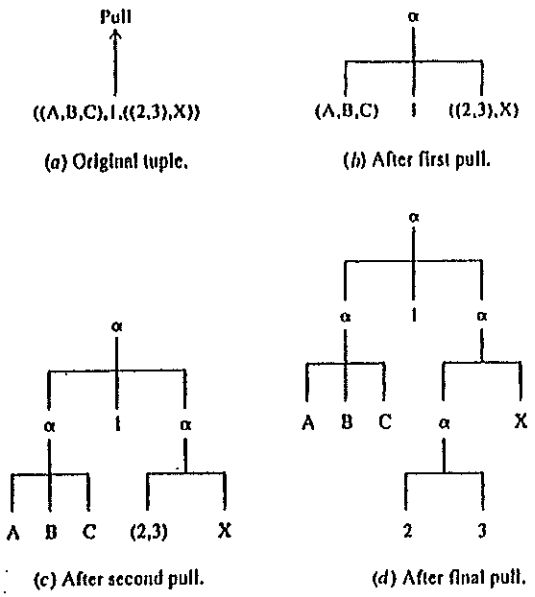
The following subsections briefly describe both the notation, a simple implementation, and the major operators. Because of their historical significance and widespread use in the computer science community, much of the original LISP terms will be used in this text, even though in some cases a more modern 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.



α represents a nonterminal node.

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

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

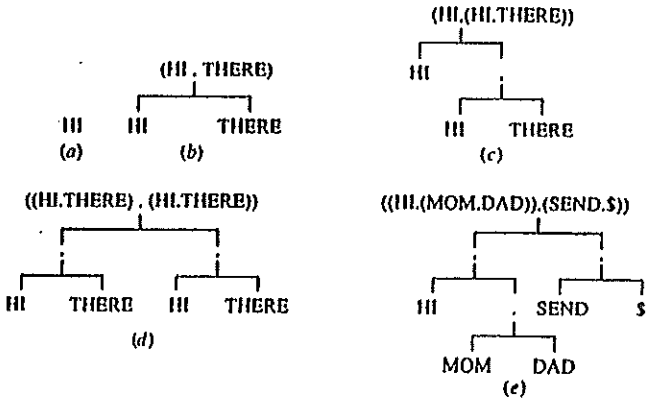


FIGURE 3-3
Sample use of dot notation.

(*a*) tag = "nonterminal." (*b*) tag = "terminal."

FIGURE 3-4
Dot notation and its equivalent.

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

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

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

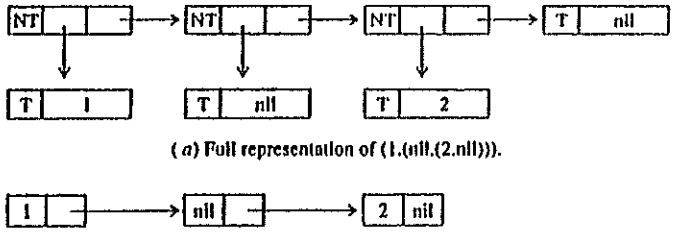
3.1.5 Shorthand Cell Drawing

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

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

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

Finally, very often the car or cdr of a nonterminal cell will point to another cell with a terminal tag. For brevity in notation, we will often draw such diagrams in a fashion similar to that for nil. The nonterminal cell is drawn with the value from the terminal cell in its appropriate field [see Figure 3-5(*b*)]. Even though such a diagram shows only the one



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

(*b*) Condensed drawing.

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

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.

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

$$\text{"}(a_1\ a_2 \ldots a_n)\text{"} \qquad \text{or} \qquad \text{"}(a_1, a_2, \ldots, a_n)\text{"}$$

has an equivalent dot version as

$$\text{"}(a_1 . (a_2 . (\ldots (a_n . \text{nil}) \ldots)\text{"}$$

Note the cases for n=1 and 2:

$$(a_1) = (a_1 . \text{nil})$$

$$(a_1\ a_2) = (a_1, a_2) = (a_1 . (a_2 . \text{nil})) \qquad \text{(THIS IS NOT } (a_1 . a_2))$$

202

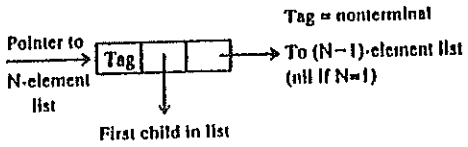
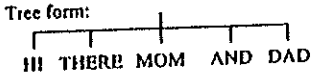


FIGURE 3-6
Linked-list equivalent of dot notation.

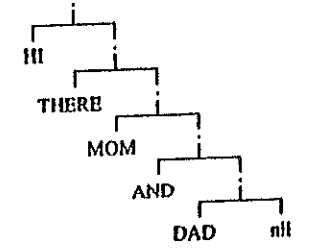
Tuple form: (HI,THERE, MOM,AND,DAD)

List form: (HI THERE MOM AND DAD)

Tree form:



Dot form: (HI.(THERE.(MOM.(AND.(DAD.nil)))))



Equivalent linked list:

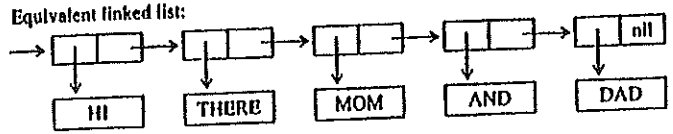


FIGURE 3-7
Multiple children in dot notation.

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).

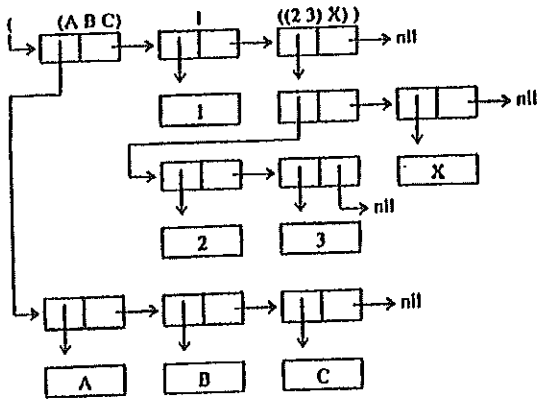
Not only are these common functions, they are often used in long strings of compositions of each other to pick out various elements of objects. For example, car(cdr(cdr(*x*))) picks out the third element of a list *x*; if *x* is the above list ((1 2.3) 3 4), this is 4. Likewise, cdr(car(*y*)) picks out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

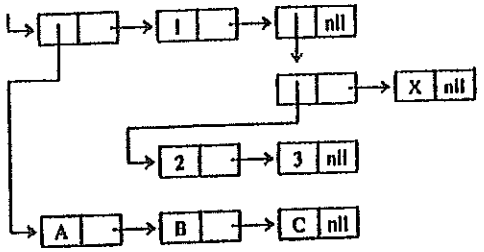
Besides car and cdr, there are several standard functions typically found in systems that support the cell implementation of dot notation (Figure 3-11 gives some examples):

(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))
(*a*) Equivalent representations.	(*b*) Different representations.

FIGURE 3-8
Three-element s-expressions.



(*a*) Explicit cell representation.



(*b*) Shorthand representation.

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

FIGURE 3-9
Pointer structure of ((A B C) I ((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))

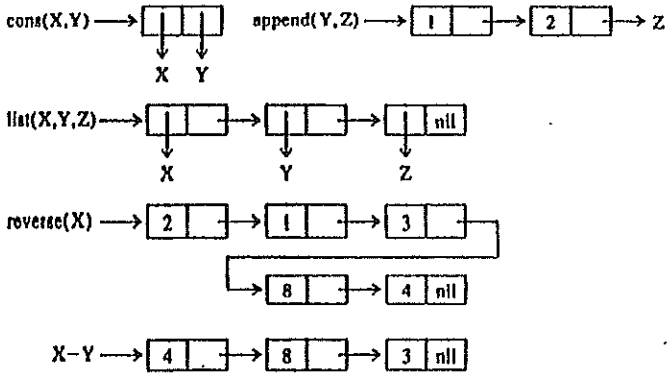
= ((A).(B.C).((D ((E.F).nil)).nil))

= ((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.

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



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

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

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

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

Also *append* $(x-y, y)=x$ and *append* $(x,y)-y=x$.

In addition to these functions there are several common predicates:

atom (x) returns true only if x is a terminal.

null (x) returns true only if x=nil.

eq (x,y) returns:

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

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

3.2 ABSTRACT PROGRAMS

(Henderson, 1980)

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

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

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

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

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
  elself 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 $x=(3\times z+6)$ ln $x^2+3\times x+4$."

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 ln 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 ln 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 ln 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 ln let x=x+1 ln let x=$x\times 2$ ln $7-x$

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 ln 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 ln. 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) = $n \times (n-1) \times (n-2) \ldots \times 1$

Basis test: Does $n = 0$?

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

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

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

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

FIGURE 3-14
Recursive factorial.

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

fact(n) = fact$(n, 1)$
where rec fact(n, z) = if $n = 1$ then z else fact$(n-1, n \times z)$

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

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

Another example of a non-tail-recursive definition is the standard one for a 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-

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 B) C), (A (B C) D)) = (A (D B) C (B C) D)

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

FIGURE 3-15
Some common recursive functions.

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

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

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

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

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

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

→ append( append[ append[ append(nil,D,nil),C,nil] ,B,nil],A,nil)
                                          ↑   ↑     ↑     ↑
                                          1   1     1     1
→ append( append[ append[(D),(C)] ,(B)],(A))
                  ↑
                  1 more

→ append(append[(D C),(B)],(A))
          ↑
          2 more

→ append((D C B),(A)) → (D C B A)
  ↑
  3 more
```

FIGURE 3-16
Counting cons operations in reverse.

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

(*a*) An imperative reverse.

```
reverse(x) = rev(x,nil)
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.

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

3.4 ABSTRACT SYNTAX

(McCarthy et al., 1965)

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

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

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

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

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

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

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

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

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

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

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

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

Predicate functions:

is-a-number(x)—true if x has syntax of a number

is-paren(x)—true if x has outer ( )

is-a-×(x)—true if x is the × operator

is-a-+(x), is-a-/(x),...—similar

Selector functions:

get-value(x)—returns numeric value of x (assuming x is a number)

get-body(x)—returns x without outermost ( )

get-operator(x)—returns outermost ×, /, +, −

get-left(x)—returns left term from outermost function

get-right(x)—returns right term from outermost function

Creator functions:

create-number(x)—converts x into syntatically valid number string

create-infix(left, operator, right)—creates an infix expression

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

FIGURE 3-19
Abstract syntax functions for integer expressions.

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

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


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

FIGURE 3-20
Abstract interpreter for integer expressions.

3.5 PROBLEMS

1. Find equivalent forms for the following s-expressions that minimize the number of dots.
   (((1 . (2 . (3 . nil))) . (4 . (2 . nil))) . nil)
   ((1 . 2) . (2 . 3))
   ((1 2 (3 . nil) . (4 5 6)))
2. Draw a picture of the cells needed to implement the s-expressions of Figure 3-7(*b*), showing all cells.
3. How many memory cells are needed for each of the following? Draw a picture (use condensed form).
   ((1 2)(3 4)(5 6))
   ((1 . 2)(3 . 4)(5 . 6))
   (((1 . 2) . (3 . 4))(5 . 6))
   (((1 . 2) . (3 . 4)) . (5 . 6))
   ((1 . 2) . ((3 . 4) . (5 . 6)))
   ((1 2) . ((3 4) . (5 6)))
   ((( ) ( ) ( ) . (( ) ( )))
4. Find the car, cdr, cadr, cdar, and caddr (if possible) of each of the expressions from the preceding problem.
5. Evaluate Ackerman's function A(2,1) showing all reductions (see Figure 3-13).
6. Write an abstract program that returns the difference of two lists.

7. Rewrite the sumproduct function without using accumulating parameters.
8. Neither union nor intersect as defined in the text guarantee that the elements of their output lists are all unique, that is, there is no duplication of elements. Write some versions that guarantee uniqueness, regardless of whether or not the original input lists had duplicates in them.
9. Write a tail-recursive form of the 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 ($<$, $=$, $>$, $\leq$, $\geq$) between two integer expressions. Add any abstract functions you feel are necessary. Indicate which are recursive.

CHAPTER 4

LAMBDA CALCULUS

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

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

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

The following sections address the syntax and semantics of lambda calculus, with particular emphasis on the rules for formal parameters and

their replacement under substitution. Also addressed is what is possible when an expression has several different internal function-argument pairs that could be applied at the same step. These discussions will lead in later chapters to opportunities for parallelism that are not found in conventional computing.

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

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

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

4.1 SYNTAX OF LAMBDA CALCULUS

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

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

The alphabet for this language consists of the set of lowercase letters, plus the characters "(," ")," "|," and "λ" (the Greek character *lambda*). The nonterminal "(expression)" describes all valid lambda calculus expressions, and comes in one of three forms: an application, a function, and an identifier. An *application* expression consists of the concatenation of two other expressions, surrounded by a set of parentheses.

```
<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 "((λ*x*|(*yx*))*a*)." The subexpression "(λ*x*|(*yx*))" is an example of a function. "(λ*y*|(λ*x*|(*y*(*yx*))))" is a nested function expression.

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

4.2 GENERAL MODEL OF COMPUTATION

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

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

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

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

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

Given: ((λx|(xi)) ((λz|((λq|q)z))h))

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

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

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

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

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

FIGURE 4-2
Sample computation.

4.3 STANDARD SIMPLIFICATIONS

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

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

$$(\ldots((E_1E_2)E_3)\ldots E_n)$$

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 (λ*x*|λ*y*|λ*z*|M), it is permissible to cascade all the formal parameter identifiers into a single string, as in (λ*xyz*|M). Again, however, the meaning of this is very precise. An expression of the form

$$(\lambda xyz|M)E_1E_2E_3$$

still means that the function is $(\lambda x|\lambda yz|((zy)x))=(\lambda x|(\lambda y|(\lambda z|zyx)))$, and takes only the single argument E_1 for the formal *x*. The result will be a function $(\lambda yz|[E_1/x]M)$, 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-

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 $(\lambda x y | y(yx))$ 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

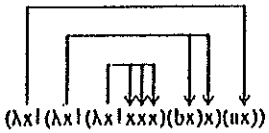
in their list of formal arguments. Each of these function definitions controls a different, nonoverlapping part of the expression. Figure 4-3 gives an example of such an expression and the values taken on by different instances of the identifier x.

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.

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=(λy|A), $y=x$, and there is an instance of x in A.
3. E=(λy|A), $y \neq x$, and x occurs bound in A.

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

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

4.5 SUBSTITUTION RULES

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

The difference is in the first substitution of the second sequence. If we blindly apply the function (λq|(λp|p(pq))(λr|+ p r)) to its argument (+ p 4), we substitute the latter for q inside the function's body. However,

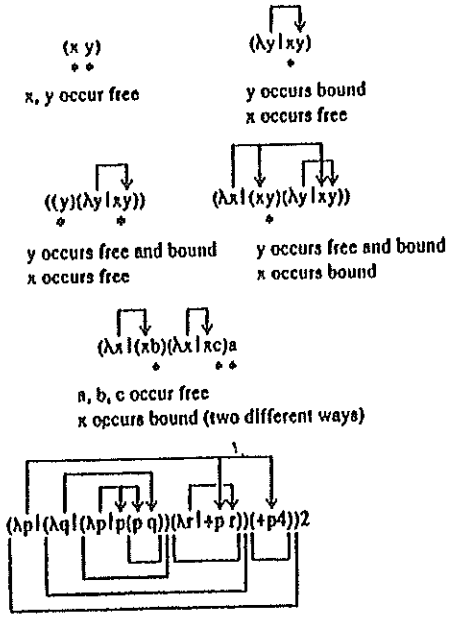


FIGURE 4-4
Sample identifier occurrence patterns.

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 (λp|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 (λx|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 $(\lambda x|E)A \rightarrow [A/x]E \rightarrow E'$

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, $y \neq x$) $E' = E$.
   Examples:
   - $(\lambda x|x)M \rightarrow [M/x]x \rightarrow M$
   - $(\lambda x|y)M \rightarrow [M/x]y \rightarrow y$

2. If $E = (BC)$ for some expressions B and C
   then $E' = (([A/x]B)([A/x]C))$.
   Example: $(\lambda x|xyx)M \rightarrow (([M/x](xy))([M/x]x)) \rightarrow MyM$

3. If $E = (\lambda y|C)$ and C some expression

   a. and $y = x$, then $E' = E$.
      Example: $(\lambda x|(\lambda x|xN))M \rightarrow (\lambda x|xN)$

   b. and $y \neq x$ and y does not occur free in A
      then $E' = (\lambda y|[A/x]C)$
      Example: $(\lambda x|(\lambda y|xyN))M$
      $\rightarrow [M/x](\lambda y|xyN))$
      $\rightarrow (\lambda y|[M/x](xyN)))$
      $\rightarrow (\lambda y MyN))$
      $\rightarrow (\lambda|MyN)$

   c. (RENAMING RULE) otherwise (i.e., y occurs free in A)
      $E' = (\lambda z|[A/x]([z/y]C))$, where z is a new symbol never used before (or at least not free in A)
      Example: $(\lambda x|(\lambda y|xyN))(ay)$
      $\rightarrow [(ay)/x](\lambda y|xyN)$ (note y occurs free in (ay))
      $\rightarrow (\lambda z|[(ay)/x][z/y](xyN))$
      $\rightarrow (\lambda z|[(ay)/x](xzN))$
      $\rightarrow (\lambda z|(ay)zN)$

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 substi-

tution 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.

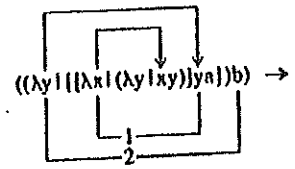
In this context it becomes relevant to discuss under what conditions we know that two separate expressions are in fact the same, and what kinds of standard forms there are that simplify such comparisons.

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: $(\lambda p|(\lambda q|(\lambda p|p(p\ q))(\lambda r|+\ p\ r))(+\ p\ 4))2$

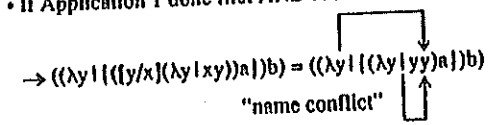
Apply $(\lambda q|(\lambda p|p(p\ q))(\lambda r|+\ p\ r))$ to $(+\ p\ 4)$ first (Properly)!
$\rightarrow (\lambda p|(\lambda z|z(z\ (+\ p\ 4)))(\lambda r|+\ p\ r))2$
$\rightarrow (\lambda z|z(z\ (+\ 2\ 4)))(\lambda r|+\ 2\ r)$
$\rightarrow (\lambda z|z(z\ 6))(\lambda r|+\ 2\ r)$
$\rightarrow (\lambda r|+\ 2\ r)((\lambda r|+\ 2\ r)\ 6))$
$\rightarrow +\ 2\ ((\lambda r|+\ 2\ r)\ 6)$
$\rightarrow +\ 2\ (+\ 2\ 6) \rightarrow 10$

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),



... → 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|xzx) → (λy|yzy)

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

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

FIGURE 4-9
Reduction rules for expression equivalence.

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

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

(λx|+ x y) 3 → (+ 3 y)

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

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

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

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

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

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

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

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

FIGURE 4-10
Sample equivalent expressions.

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

$(\lambda x|Ex)A$ with no free x's in E

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

$[A/x](Ex) \rightarrow EA$, since there are no free x's in E.

Using eta conversion first,

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

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

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

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

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 $(\lambda x_1 \ldots x_n|\ (\ldots(\mathbf{E}x_1)x_2)\ldots x_n)\ x_2)\ldots x_n)$.

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

For any expression A:

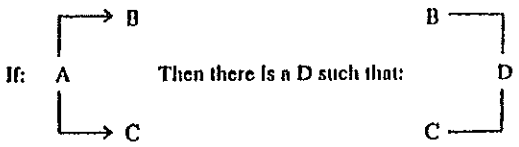


FIGURE 4-11
The Church-Rosser theorem.

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))y)a)} 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.

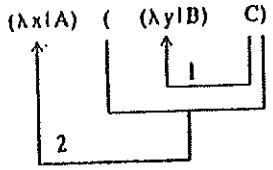
Figure 4-13 gives a generic expression where either applicative- or normal-order evaluation is possible. Normal-order reduces the argument to ((λylB)C) for x into ((λzlA)x). Applicative order reduces the argument to [C/y]B first (and potentially reducing A) before replacing x in A.

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 ((λzlz)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



Normal reduction: Do 2 first, then 1

Applicative reduction: Do 1 first

FIGURE 4-13
Multiple applications.

Expression: (λxl(λwl(λylwyw)b)) ((λxlxxx)(λxlxxx)) ((λzlz)a)
→ ([((λxlxxx)(λxlxxx))/x]([λwl(λylwyw)b])) ((λzlz)a)
→ ([λwl(λylwyw)b]) ((λzlz)a)
→ (λyl((λzlz)a)y((λzlz)a))b
→ ((λzlz)a)b((λzlz)a)) first redundant evaluation
→ (ab((λzlz)a)) second redundant evaluation
→ aba

(*a*) Normal order reduction.

→ (λxl[b/w](wyw)) ([(λxlxxx)/x](xxx)) ((λzlz)a)
→ (λxlyyb) ((λxlxxx)(λxlxxx)(λxlxxx)) ((λzlz)a)
→ (λxlyyb) ([(λxlxxx)/x](xxx)(λxlxxx)) ((λzlz)a)
→ (λxlyyb) ((λxlxxx)(λxlxxx)(λxlxxx)(λxlxxx)) ((λzlz)a)
→ . . . . repeat reductions forever. . . .

(*b*) Applicative-order reduction.

→ ([((λxlxxx)(λxlxxx))/x]([λwl(λylwyw)b])) ((λzlz)a) normal
→ ([λwl(λylwyw)b]) ((λzlz)a)
→ ([λwl(λylwyw)b]) ([a/z]z) applicative
→ ([λwl(λylwyw)b]) a normal
→ (λylaya)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.

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 (λ*xyz*|E). In a sense this notation is "syntatic sugar," since, given multiple real arguments, the formal way to interpret something like (λ*xyz*|E)ABC is one argument at a time, namely:

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

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

Although this handles currying automatically, it is a bit cumbersome for normal usage when it is obvious that an application has all the arguments it needs to satisfy its function object directly. The notational trick we will use to avoid this clumsiness of one argument at a time is a ***multiple simultaneous substitution***, denoted "[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 $n \geq m$, 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.

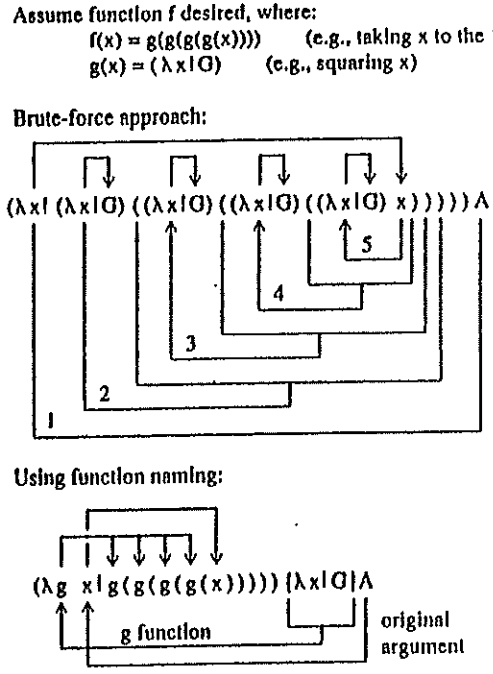


FIGURE 4-15
Function naming example.

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.

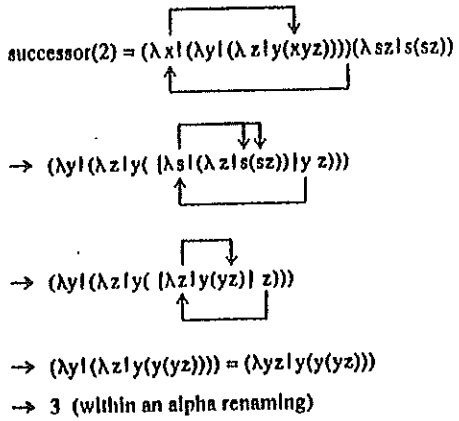


FIGURE 4-16
Sample application of successor function.

522

4.11 BOOLEAN OPERATIONS IN LAMBDA CALCULUS

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

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

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

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

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

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

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

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

not = (λ*w*|*w*FT)
= (λ*w*|*w*(λ*xy*|*y*)(λ*xy*|*x*))
Example: not T → TFT → F

and = (λ*wz*|*w*z**F**)
= (λ*wz*|*w*z(λ*xy*|*y*))
Example: and T F → TFF → F

or = (λ*wz*|*w***T***z*))
= (λ*wz*|*w*(λ*xy*|*x*)*z*)
Example: or F T → F T T → T

FIGURE 4-17
Standard logical connectives.

zero(x) = (λ*x*|*x* F not F)
= (λ*x*|*x* (λ*xy*|*y*) (λ*w*|*w*(λ*xy*|*y*)(λ*xy*|*x*)) (λ*xy*|*y*))

Examples:

zero(0) = (λ*x*|*x* F not F) (λ*nz*|*z*)
→ (λ*nz*|*z*) F not F
→ ([F/*n*,not/*z*]*z*) F
→ not F
→ T

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

zero(Z_k) = (λ*x*|*x* F not F) (λ*nz*|*n*(..(*nz*)..)), k>0
→ (λ*nz*|*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.

```
f(a) = if zero(a) then 0 else a + 1
= (λalzero a 0 (successor a))
= (λal(λxlx (λnzlz) (λwlw(λxyly)(λxylx)) (λnzlz))
        a
        (λnzlz)
        ((λxyzly(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 $(\mathbf{YR})\mathbf{A} \rightarrow \mathbf{R}(\mathbf{YR})\mathbf{A}$. 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 $(\lambda f x|\mathbf{E})$. 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.

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

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

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

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

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

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

FIGURE 4-20
Reduction of 4

4.13 PROBLEMS

1. Compute all the variations in application sequences possible for Figure 4-2 and show that they yield the same answer.
2. In the following lambda expressions, which identifiers are free, and which are bound, and to which lambda.
   a. ((λy(yxx)y x)
   b. ((λx(λx(λx)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?
   (λmnf(m(nf)) Z_i Z_j
   (λmn(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.

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 n 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 = (λnlif 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 $(\lambda xylx)$ and $(\lambda xyly)$, 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 $(\lambda szls^k z)$ as described earlier. Negative integers have several possible forms, including functions like $(\lambda nln{-}k)$, 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:

$$x+3 \equiv (\lambda wzyxlwy(zyx)) \times (\lambda nzln(n(nz)))$$

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, "$f(A_1,\ldots,A_n)$" is equivalent to "$(F\,B_1\ldots B_n)$," where F is the lambda-expression form of f [i.e., something like $(\lambda x_1\ldots x_n|E)$] 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:

$$(P_1\ E_1\ (P_2\ E_2\ (\ldots(P_n\ E_n\ E_{n+1})\ldots)$$

For simplicity, the keyword elseif will be used for such combinations.

5.5 LET EXPRESSIONS—LOCAL DEFINITIONS

Many conventional programming languages offer a *macroexpansion* capability whereby for a certain region of text in the program (called the *scope*) all occurrences of a certain identifier are to be replaced by some predefined text string. In other languages, where the passing of arguments to a procedure is by *call-by-name* semantics, the occurrence of a formal argument in the body of the procedure actually means replacement of that argument by the result obtained by evaluating the text corresponding to the actual argument.

Both of these mechanisms have two things in common. First, they simplify the writing of expressions by letting some identifier "stand for" some other expression. Second, the use of this identifier is very controlled; throughout the entire scope the identifier's value as an externally defined expression (almost) never changes.

Abstract programming has a notation very akin to a cleaned-up combination of macros and call-by-name. The simplest form of this notation (called a *let expression*) is

let (identifier)=(expression) in (body)

where (body) is itself an arbitrary expression.

This whole expression is equivalent in value to a copy of the body where every free occurrence of the identifier in the definition part is replaced by the expression in the definition. In terms of its lambda equivalent (from which we get its exact meaning), a let expression of the form "let x=A in E" is the same as "$(\lambda x|E)A$," 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

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* = (λ*x*|(*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⟩

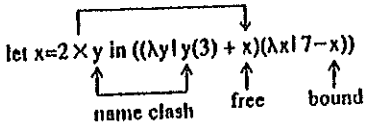
or

(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 (λ*s*|*s*(*s* 2)) (λ*x* | *x*×*x*).



is equivalent to

(λx|(λy|y(3) + x)(λx|7−x))(2×y)
→ ((λz|z(3) + 2×y)(λx|7−x))
→ 4+2×y

FIGURE 5-3
A sample let expression.

Figure 5-4 diagrams another example in detail. In both cases, the new notation is quite easy to read, and agrees closely with notation found in many conventional languages. However, the reader should take care to study these examples closely and verify exactly how each part of the new let form migrates into the multiple functions in the lambda equivalent.

5.5.2 Nested Definitions

Given that a let expression is itself an expression, it is also possible to nest one inside another, particularly inside the other's body. Again the meaning of this is an exact extension of the basic let equivalence. Figure 5-5 diagrams a triply nested let and its lambda equivalent.

As before, it is important to apply the rules for identifying free instances and performing the resulting substitutions properly. The second example in Figure 5-5 diagrams such a case where the middle let has two different x's, the x in the definition expression that is receiving the value 7 from the outermost let, and the x that is equivalent to the value 8 and is being substituted into the third let.

5.5.3 Block Definitions

One important consequence of the nesting rules for let expression is that identifiers that appear free in the expressions for definitions of inner lets are perfectly valid candidates for replacement by definitions in more outer lets. Thus, in "let x=A in let y=...x...in...," the free x in the y definition is replaced by A.

There are many cases, however, where what is desired is a simultaneous replacement of several definitions into a let body, with no cross-substitutions among the definitions. In our abstract programming language the *and* form of definitions permits this simultaneity. An expression of the form

let x=A and y=B and z=C in E

let f = (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:

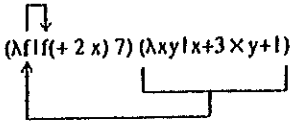
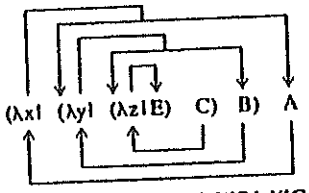


FIGURE 5-4
Ways to define local functions.

```
let x = A in
   let y = B in
      let z = C in E
```



Substitution eqvt: [A/x]([B/y]([C/z]E))

```
Sample: let x=7 in
           let x=x+1 in
              let f(y)=y+2×x in x×f(3)

  → let x=8 in let f(y)=y+2×x in x×f(3)
  → let f(y)=y+16 in 8×f(3)
  → 8×(3+16) → 152

Equivalent: (λx|[λx|[λf|x×f(3)][λy|y+2×x]][x+1])7
  =[7/x]([x+1/x]([(λy|y+2×x)/f](x×f(3))))
  →[7/x]([x+1/x](x×(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:

$$(\lambda xyz|E)\ A\ B\ C$$

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.

Y, defined earlier, at carefully selected spots in the let expression. Unfortunately, the result is not terribly readable.

The alternative used in abstract programming is the letrec expression, a recursive form of the standard let. Here the major difference is that any free occurrence of the definition's identifier in the definition's expression is replaced by the expression itself. Thus, in

letrec $f(n)$=if zero(n) then 1 else $n\times f(n-1)$ ln $f(4)$

the free occurrence f in $f(n-1)$ is replaced (recursively) by the whole expression

if zero(n) then 1 else $n\times f(n-1)$

All uses of f in the letrec's body now are replaced by this whole recursively defined expression.

The conversion from this notation to a pure lambda equivalent is direct. Given "letrec f=A ln 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 (λf|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 ln E

is

$$(\lambda f|E)\ (Y\ (\lambda f|A)) = (\lambda f|E)\ ((\lambda y|(\lambda x|y(xx))(\lambda x|y(xx)))\ (\lambda f|A))$$

A letrec expression is really useful only for defining local functions (try out your patience on "letrec x=x+1 ln x"). Consequently, one would expect the "A" in "letrec f=A ln E" to be a lambda function itself (of the form (λx|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 ln E" involves making a lambda function (λx|B) and creating a nested set of applications from it as defined above. To verify this, consider:

letrec $f(n)$=if zero(n) then 1 else $n\times f(n-1)$ ln $f(4)$

Its pure lambda equivalent is only one reduction away from that detailed at the end of Chapter 4:

$$(\lambda f|f\ 4)\ (Y\ (\lambda f|n|\text{zero}\ n\ 1\ (\times n\ (f\ (-n\ 1)))))$$

The nesting of letrecs is also permissible, and mirrors nested lets directly. Here the definition of any recursive function can reference any recursive function defined in surrounding letrecs, but cannot use definitions in deeper letrecs.

5.6.1 Mutually Recursive Functions

In contrast to let, however, there are some subtle differences between multiple definitions in the *and form* of a letrec versus a let. In the latter there is absolutely no connection between the expressions used in the definitions, even though they may use some of the same identifiers being given definitions. In the letrec form, however, the expression in each anded definition has complete access to every other definition. The result is a set of *mutually recursive functions* that are defined together.

This mutual dependency makes conversion of a multiple-definition letrec somewhat more challenging than anything we have done yet. Consider, for example, the expression

letrec $f(x)$=A and $g(x,y)$=B ln 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 (λfg|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 (λfgx|A) for f and ($\lambda fgxy$|B) for g.

Define combinators:

Y_1 = (λfg|RRS)

Y_2 = (λfg|SRS)

where:

R = (λrsf(rrs)(srs))

S = (λrsg(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.

222

```
letrec f(x) = A and g(x,y) = B in B

is equivalent to

(λfgIB) {Y1(λfgxIA)(λfgxyIB)} {Y2(λfgxIA)(λfgxyIB)}
```

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.

```
map(f,x) = "apply function f to each element of list x"
         = if null(x) then nil
           else cons(f(car(x)), map(f,cdr(x)))
           e.g., map((λz| × 3 z),(3 5 7)) = (9 15 21)

map2(f,x,y) = "f applied to matching elements of lists x and y
            = if null( x) then nil
              else cons( f(car( x),car( y)), map2( f,cdr( x),cdr( y)))
              e.g., map2((λxy| x + y),(1 2 3),(4 5 6)) = (5 7 9)

reduce(f,t,x) = "recursively apply f to each x and prior result"
              = if null(x) then t
                else f(car(x),reduce(f,t,cdr (x)))
                e.g., reduce( + ,0,(1 2 3)) = 1+(2+(3 +0)) = 6

vector(f,x) = "apply list of functions f to x"
            = if null(f) then nil
              else cons((car(f)) x, vector(cdr(f), x))
              e.g., vector(((λx|2 × x) (λz|z−6) (λx|x)),3) = (6 −3 3)

while(p,f,x) = "while loop"
             = if p(x) then while(p,f,f(x)) else x
               e.g., cdr(while({λx|car(x)>0},
                               {λx|cons(car(x)−1, car(x)×cdr(x)},
                               (4.1))) = 4!

compose(f,g) = composition of functions f and g = (λx|f(gx))
  e.g., compose((λx|3×x), (λx|(4+x)) → (λx|3×(4+x))
```

FIGURE 5-10
Some higher-order functions.

An entirely different kind of higher-order function is *while*. This function takes two function arguments and a third object as an accumulating parameter. It operates much like a *while loop* in a conventional programming language. As long as the application of the first function argument (the loop's iteration test) to the accumulating parameter returns T, while recursively calls itself with the accumulating parameter modified by applying it to the second function argument (the loop body). When the result is F, the accumulating parameter is returned as the value. The example in the figure computes the factorial of a number.

The final higher-order function is *compose*. This function takes as its arguments two other functions, and produces as a result a new function which is the composition of the two.

5.9 AN EXAMPLE—SYMBOLIC DIFFERENTIATION

As an example of many of the above capabilities, we consider here the problem of writing a function that can take the symbolic derivative of an

$dx/dx = 1$

$dy/dx = 0\ (y \neq x)$

$d(A+B+C+..)/dx = dA/dx + dB/dx + dC/dx + \ldots$

$d(A-B)/dx = dA/dx - dB/dx$

$d(A\times B)/dx = (dA/dx)\times B + A\times dB/dx$

$d(A/B)/dx = \ldots$

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.

```
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) $3\times x + x\times y + (4-x)\times z$
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) $(0\times x + 3\times 1) + (1\times y + x\times 0) +$
$((0 - 1)\times z + (4-x)\times 0)$

FIGURE 5-13
Abstract program for symbolic differentiation.

the variable to drive the differentiation. The main function is recursive and has one local function (triple) of its own.

The main body of dif is a set of nested if-then-elses that determine what kind of expression the argument is. As usual, the first leg of the tests is the basis case (is *e* an atom?) whose T result is a nested test of whether or not that atom is the same as *x*. The else leg of this test covers all the recursive cases where *e* is not an atom, i.e., an expression with subexpressions that will have to be differentiated in turn. It starts with a let expression that picks off the first element of *e*, which in prefix notation must be the operator name. A series of tests then cover the different operators individually.

The first of these cases is for "+." Here the rule says to add up the derivatives of all the terms of the original sum. The approach chosen uses the recursive higher-order function **map2s**, which operates like map2 except that the second argument for the nested calls is a *scalar* object that does not change from call to call. The actual arguments given to **map2s** include dif itself, the list of sum terms to be differentiated, and the derivative symbol. The result is a list to which we add a "+" to recreate the whole result.

After "+," the operators involve exactly two operand expressions. A dual let expressions picks these out of *e* before proceeding. Then, in each case, the actual expression for the derivative uses the simple nonrecursive local function triple to create the appropriate results from derivatives of the subexpressions.

5.10 PROBLEMS

1. Write as an abstract program the predicate free(*x*,*e*), which returns true if the identifier *x* occurs free in the lambda expression *e*.
2. Write an abstract definition for a function subs(*y*,*x*,*m*), where *y* and *m* are any valid s-expressions and *x* is an atom representing a variable name. The result of the function should be an s-expression equivalent to $[y/x]m$. Assume that you have available a function newvar(), which at each call returns a new variable name that has not been used before. Your definition should detect and perform renaming properly.
3. Translate any of the abstract programs for reverse into pure lambda notation. Assume only the primitive functions null, cons, car, and cdr.
4. (Project) Write an abstract program that converts an abstract program back to pure lambda calculus. Use abstract syntax functions as required.
5. (Hard Project) Go the other way—from pure lambda calculus to some readable form of abstract programming.
6. What are the values bound to *x*, *y*, and *z* at each level of the following:

```
let x=4 in
  let y=x+2
  and z=x-3 in
    let x=y+x
    and y=z+3
    and z=y+6 in x+y+z
```

7. Show where to put the Y expression if one wanted to define a single recursive function using just the let expression.
8. Show that Figure 5-7 is correct.
9. In analogy to Figure 5-7 and Figure 5-8, show how to translate to pure lambda a letrec expression of the form "letrec f(*x*)=A and g(*x*,*y*)=B and *h*(*x*,*y*,*z*)=C in E." Discuss how this generalizes to an arbitrary number of mutually recursive functions.
10. Convert the abstract program in Figure 5-13 to pure lambda calculus form. (You need not translate car, cdr, cons, atom, =, or character representations for "+," "-," "*,"....)
11. Write an abstract program to optimize arithmetic expressions constructed according to Figure 5-12. Use optimization rules of the form:

"(× 0 A)" is 0
"(× A 1)" is A
"(+ A 0)" is 0

An expression involving only numbers can be reduced to a number.

You may assume that the predicate number(*x*) returns true if *x* is a number and false otherwise.

12. Rewrite reduce to use an accumulating parameter. You may redefine reduce to "parenthesize" the other way if you wish.
13. Define and write in abstract syntax an until function modeled after the while function described in the text.
14. Complete Figure 5-13 to include division and the unary operators "−" and "sqrt."

CHAPTER 6

SELF-INTERPRETATION

Lambda calculus is powerful enough to express any computable function, and is thus as powerful as any other computing language. Given this, it is interesting to ask if the execution model of lambda calculus is itself expressible as a computable function. If it is, then we have the possibility of writing in lambda calculus an interpreter for itself. Such an interpreter would express the semantics of lambda calculus in itself (and thus for any language that lambda calculus supports—such as abstract programming).

As hinted at several times, and perhaps partially demonstrated by the differentiator evaluator of the last chapter, the answer to this is a resounding yes. We can write a lambda calculus function, usually called ***eval***, which when given any arbitrary lambda expression E, reduces it as far as possible, and returns the result. In particular, since eval(E) (before reduction) is itself a valid lambda expression, we are perfectly free to try eval(eval(E)), that is, have eval determine what happens when eval itself is turned loose on an expression. The answer returned from eval(eval(E)) is the same as that from eval(E), and both are equivalent to the reduced form of E. Eval thus forms a completely valid ***interpreter*** for lambda calculus, expressible in lambda calculus.

The first few sections in this chapter describe eval using abstract syntax functions for the syntax parsing of generic lambda expressions. Two versions are given, one which performs normal-order evaluations, and one which performs applicative-order evaluations.

After this we will introduce a particular concrete syntax for lambda calculus using s-expressions and prefix notation similar to that for the in-

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.

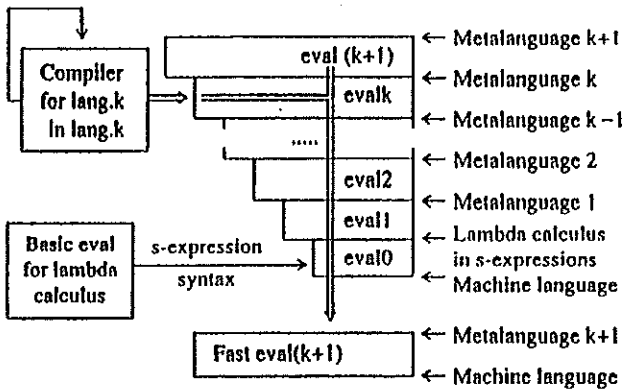
Such languages are often called *metalanguages*, and the evals which support them, *metainterpreters*. When given a program in the highest such language, the operation of such a system corresponds to the lowest-level interpreter simulating the execution of the first metainterpreter as it simulates the 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:

$eval_0(eval_1(eval_2(\ldots eval_n(P_n)..) \rightarrow$ result

Assume $compiler_k$ = compiler program for metalanguage k written in metalanguage k.

$eval_0(eval_1(\ldots eval_k(compiler_k(compiler_k)..) = {}^*compiler_k$

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((((λxl(λylx)) a) b))
→ apply(get-function((((λxl(λylx)) a) b)),
        get-argument((((λxl(λylx)) a) b)))
→ apply(((λxl(λylx)) a), b)
→ apply(eval(((λxl(λylx)) a)), b)
→ apply(apply((λxl(λylx)), a), b)
→ apply(eval(subs(a, get-id((λxl(λylx))), get-body((λxl(λylx))))), b)
→ apply(eval(subs(a, x, (λylx))), b)
→ apply(eval((λzla)), b)
→ apply(create-function(get-id((λzla)), eval(get-body((λzla)))), b)
→ apply(create-function(z, eval(a)), b)
→ apply(create-function(z, a), b)
→ apply((λzla), b)
→ eval(subs(b, get-id((λzla)), get-body((λzla))))
→ 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)

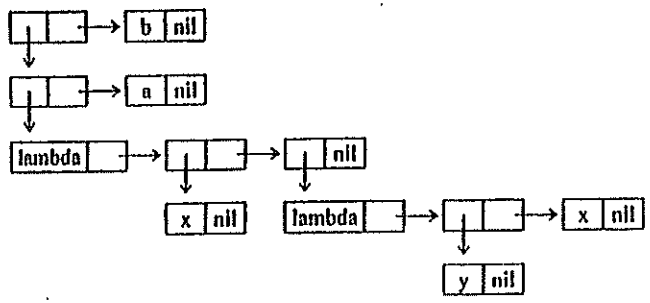


FIGURE 6-5
Lambda expression translation into s-expressions.

252

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 $(\lambda xyly(xx)) \Rightarrow$ (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:

$$(\lambda xyly(xx))\ w\ z \Rightarrow \text{((lambda } (x\ y)\ (y\ (x\ x)))\ w\ z)$$

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.

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.

```
apply-builtin(f,a) = "an applicative-order evaluation"
  let args = map(eval, a) in "args = list of evaluated arguments"
     if f = "car" then caar(args)
     elseif f = "cdr" then cdar(args)
     elseif f = "+" then car(args) + cadr(args)
     elseif ...

eval-cond(a) = "special evaluation for cond"
  if null(a) then nil
  else "iterate thru the list of pairs"
       let z = car(a) in
          if eval(car(z)) = T
          then eval(cadr(z))
          else eval-cond(cdr(a))

subs2(v,n,e) = "substitute v's for n's in e"
  if atom(e) "if e variable, replace"
  then lookup(e,n,v)
  elseif car(e) = lambda
  then "substitute into a lambda expression—rename"
       let old = cadr(e) in "cadr = id list"
          let new = map(new-id,old) in
          list(lambda,new, subs2(v, n, subs2(new,old,caddr(e))))
  else "e is an application—substitute in all parts"
  map((λz|subs2(v,n,z)), e)

lookup(z,n,v) = "find z in n and replace by value in v"
  if null(n) or null(v)
  then z
  elseif car(n) = z then car(v)
  else lookup(z,cdr(n),cdr(v))
```

FIGURE 6-10
Support functions for expanded interpretations.

avoid for now the complications described earlier for mutually recursive functions.

The major differences between Figure 6-9 and Figure 6-3 are in the handling of multiple arguments and in applications involving *special forms* (where "keywords" are prefixes). Syntatically, multiple actual arguments for a standard lambda application are represented as multiple elements in an s-expression list following the prefix element [a function of the form "(lambda (...) (...))"]. They are handled by expanding the subs function into subs2, which "simultaneously" replaces all the occurrences of identifiers from the lambda function by matching value expressions from the application list. Internally, subs2 recursively processes the lambda function's body expression one element at a time, using the function *lookup* each time an element which is a variable is found:

Currying is detected when the number of actual arguments is less that the number of formal identifiers in the lambda expression. In this case, only those identifiers with matching variables are replaced, and the rest are left as identifiers for a new lambda expression.

Special forms are handled by special if-then-elseif tests in eval. "Built-in" functions are handled as described above. All the arguments are evaluated, and then code corresponding to the function is executed.

For the keyword prefix lambda, the evaluation process simply evaluates the body of the function. Note that we are evaluating this body only if a request was made to evaluate the function to begin with. This is in tune with the rules of normal-order reduction.

For if, the evaluation procedure takes the first argument, the test expression, and evaluates it. On the basis of the resulting value, either the second or third argument is evaluated. This is a mix of applicative- and normal-order evaluation.

Evaluation of cond is handled similarly. The arguments in this case are pairs of expressions, the car of which are evaluated sequentially until one is found whose value is true. Then its matching cdr is evaluated. As with built-ins and if, these car tests must be fully evaluatable.

Expressions containing let are translated exactly as the abstract-language version was translated earlier in this chapter. A new function expression is created, with arguments represented by the list in the car of the let expression. This function is passed to apply along with the second element of the argument list, which itself is a list of argument values for the specified identifiers.

Evaluation of letrec is similar but more complex. For simplicity, the version shown in Figure 6-9 handles properly letrecs with at most one identifier given a recursive definition. It also assumes that this definition is itself a lambda expression (as converted into s-expression notation). As with let, the processing generates a new function expression (again in s-expression notation) and passes it to apply. In this case, however, the argument expression for apply is taken apart and put back together with an extra identifier. This is the name for the recursive function. It is then made into an application with a prefixed Y expression. Again, this is exactly as described for abstract programs.

Note that if we had wanted to, we could have added Y as a special form to eval which performed the same operations without the substitutions.

Figure 6-11 gives a partial trace of the evaluation of the factorial function. Most of the major parts of eval and apply are executed here.

6.4 ASSOCIATION LISTS

All the lambda interpreters discussed up to now have employed brute-force substitution to handle applications. The argument expression (either evaluated or unevaluated) is substituted in total for every free occurrence of the binding variable in the function's body. This requires

```
eval(((lambda (x y) (+ x (× 4 x))) (+ 3 5) 6))
  → apply((lambda (x y) (+ x (× y x))), ((+ 3 5) 6))
  → eval(subs2(((+ 3 5) 6), (x y), (+ x (× y x))))
  → eval(map((λz)subs2(((+ 3 5) 6), (x y), z), (+ x (× y x))))
  → eval((+ (+ 3 5) (× 6 (+ 3 5))))
  → apply-builtin(+, ((+ 3 5) (× 6 (+ 3 5))))
  → let args = map(eval,((+ 3 5) (× 6 (+ 3 5)))) in ...
  → let args = list(eval((+ 3 5)), eval((× 6 (+ 3 5)))) in ...
  → ...
  → let args = (8 48) in ...
  → car((8 48)) + cadr((8 48))
  → 8 + 48 → 56
```

(*a*) A simple example.

```
eval( (letrec (f) (lambda (f) (lambda (n) (cond (zero n) (T (× n (f (− n 1)))))))) (f 3)))
  → apply( (lambda (f) (f 3)), (Y (lambda (f n) (cond ...))))
  → eval(subs2((f 3), (f), (Y ...)))
  → eval(((Y ...) 3))
  → ...
  → eval(((lambda (f n) (cond ...)) (Y...) 3))
  → apply(eval((lambda (f n) (cond ...))), ((Y...) 3))
  → apply((lambda (f n) (cond ...)), ((Y...) 3))
  → eval(subs2((cond...), (f n), ((Y...) 3)))
  → eval((cond ((zero 3) 1) (T ...)))
  → eval-cond(((zero 3) 1) (T ...)))
  → if apply-builtin(zero 3) then 1 else ...
  → ...
```

(*b*) A version of factorial.

FIGURE 6-11
Sample partial interpretations.

essentially scanning a function's body twice, once to find all the free occurrences and do the substitutions, and once to find the next application to attempt.

While conceptually simple, such an approach suffers from obvious inefficiencies when the function's body contains (as most do) one or more nested if-then-else structures, or when there are multiple arguments. More passes may be needed (up to two per argument), and part, if not most, of the resulting substitutions may be wasted effort (due to then-else cases that are not used).

One alternative to such substitutions is simply to remember at the time of an application which identifiers are to get which values, and then do the substitution on an identifier by identifier basis when the function's body is scanned for the next application. For example, in the application

(λ*wxyz*|E) W X Y Z

we wish to remember the four-way simultaneous substitution [W/*w*,X/*x*,Y/*y*,Z/*z*]. This pairing is often called the ***context***, ***binding***, or ***environment*** under which the expression E is to be evaluated or reduced.

The easiest data structure in which to record such a substitution is an ***association list***, or ***alist***, which is created at the time an application is encountered and which, in some way, pairs up identifier names and the values assigned to them via applications. Such a data structure is passed as required between apply and eval when various parts of the function's body are actually evaluated. Finding an identifier in such an expression should cause eval to look up the identifier in the alist and replace it by the identifier's matching value. Thus both eval and apply must have their definitions extended to include extra arguments that pass these alists.

There are two major forms that an alist for a single application typically takes. First, it can be a single s-expression list whose elements are 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.

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,allst) = "evaluate e with context allst"
  if atom(e)
  then let z = assoc(e,allst) 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,allst) 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,allst) = "find (a,"value") in alist"
  if null(alist) then nil
  elseif caar(alist) = a then car(alist)
  else assoc(a,cdr(alist))
```

FIGURE 6-13
A simple interpreter using association lists.

The result should be (lambda (*x*) (+ *x* 4)). Eval, however, will pass control to apply, which in turn will call

eval((lambda (*x*) (+ *x y*)), ((*y*.4)))

This look good, but eval, as stated above, will return as its result its first argument unmodified (the argument is a lambda expression). The context ((*y*.4)) is lost.

One solution would be to replace the elseif line of eval by

elseif car(*e*)=lambda then list(lambda, cadr(*e*), (eval(caddr(*e*),*alist*))

This evaluates the body of the function, and would replace the *y* above by the appropriate 4, but would have its own problems. The expression eval(((lambda (*y*) (lambda (*y*) (*y y*))) 4), nil) would result in (lambda (*y*) (4 4)). This time we should have avoided the substitution because of the name clash.

Again there is a direct solution to this, namely, rename the inner function's binding variable. This renaming can be done by augmenting the alist used to evaluate the body of a lambda by a new pair that substitutes a unique identifier for the binding variable. Figure 6-14 diagrams such an approach, again for a case where the renaming is always done.

The problem with this approach is that we have essentially done much of the work we were trying to avoid by creating an association list; we end up going through an expression and doing brute-force substitution. A later section will introduce a solution to both this problem and several others. The reader should, however, remember the source of the problem, because it will show up in somewhat different circumstances later as the *funarg problem*.

6.4.2 Multiple Arguments

Multiple arguments are a feature that we definitely want to handle in real systems. A solution mentioned above involves making an association list into a list of lists, where each list element is actually the substitution list for each function application. Thus it is the list of binding variables and their matching actual arguments. The changes to make this happen in the above interpreter are small. First, the last line of eval becomes:

else apply(car(*e*), map((λ*x*leval(*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),((caadr(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) ⇒ R(R...(YR)) ⇒ ....

Matters get even worse when a set of mutually recursive functions are desired, as in

letrec *f*=F and *g*=G and *h*=H in E

Here the context (association list) for E is of the form

(((*f*.F1) (*g*.G1) (*h*.H1))...)

However, F1 (the value for *f*) (and likewise for G1 and H1) requires knowing the values for *f*, *g*, and *h* before they are computed. But this is the context for E. We need to know the association list for E before we can compute the association list for E.

The solution for this problem involves new mechanisms to be discussed in the next section.

6.5 CLOSURES

Most of the problems of the last section come from trying to improve the "efficiency" of a basic interpreter through a combination of applicative-order reduction and the use of simple association lists in place of immediate substitutions. These proved not to provide the full efficiency gain hoped for (examples where name clashes might occur), and were relatively inadequate for general recursion.

A better solution to both problems involves "packaging" an expression with its environment into a single unit which can be passed around at will, but still be unpackaged and evaluated when needed. Such a package is called a *closure* and is something that not only makes it possible to consider using alists in interpreter descriptions, but also introduces some very novel ideas that permit opportunities for parallelism and for the easy expression of essentially infinite objects. Later sections 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

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, 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:

$$\text{letrec } f_1=A_1 \text{ and} \ldots \text{and } f_n=A_n \text{ in } E$$

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:

$$[E, ((f_1.\text{"closure for } f_1\text{"})\ldots(f_n.\text{"closure for } f_n\text{"}))]$$

where the "closure for f_j" is a closure,

$$[A_j, ((f_1.\text{"closure for } f_1\text{"})\ldots(f_n.\text{"closure for } f_n\text{"}))]$$

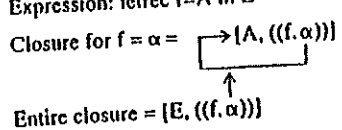


FIGURE 6-16
Value in a recursive closure.

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

$$[E,\beta] \text{ where } \beta=(\ldots(f_j.\text{"context for } f_j\text{"})\ldots)=(\ldots(f_j.\beta)\ldots)$$

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

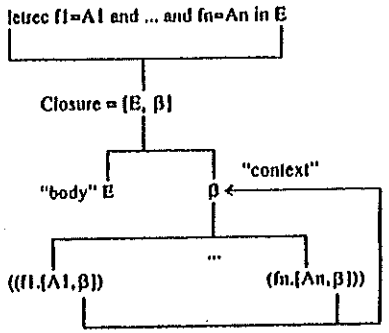


FIGURE 6-17
A fully recursive context.

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 llstener(e) whererec llstener(e)=llstener(wrlte(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 llstener 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 llstener never completes—as soon as write(eval(read())) completes, llstener starts up another go-around. The argument for llstener 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. $(((\lambda x\vert(\lambda y\vert(yx)))a)b)$
   b. $(\lambda x\vert(\lambda y\vert(\lambda z\vert(y((xy)z)))))(\lambda n z\vert z)$
   c. $((\lambda x\vert(\lambda y\vert y)))\ ((\lambda x\vert(xx))(\lambda x\vert(xx))))\ w$
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 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-

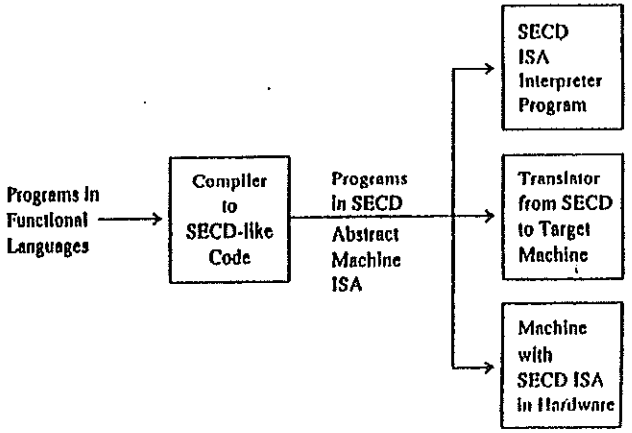


FIGURE 7-1
Uses of the SECD instruction set architecture.

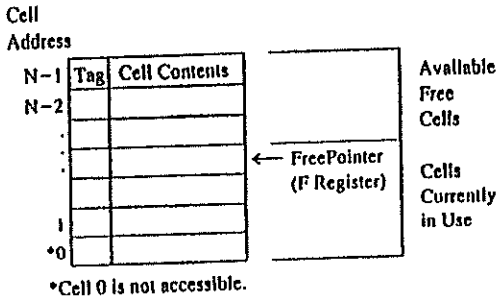
sion of the SECD Machine described here is most closely related to that of Henderson (1981), but it has been modified in several aspects to improve its educational value. Actual Pascal code that describes an interpreter for an SECD Machine can be found in both Henderson (1981) and Henderson et al. (1983). Burge (1975) contains another good description.

7.1 LIST MEMORY

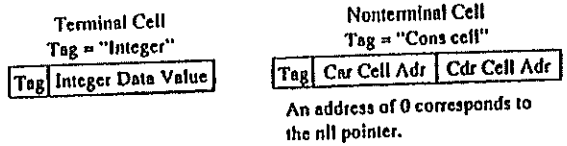
As described earlier, functional languages and s-expressions seem to go well together. Further, the implementation of s-expressions from conventional random-access memory seems to be fairly efficient. Consequently, it should come as no surprise if we assume that our abstract SECD Machine incorporates 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



(*a*) SECD memory.



(*b*) Cell formats.

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.

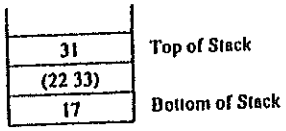
At this point we will ignore what happens when F runs over the top of available memory. This, and related questions on how to "reclaim" cells below F that are no longer in use, is a subject of the next chapter.

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.

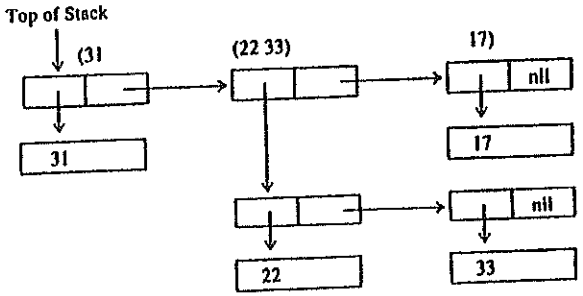
7.2 BASIC DATA STRUCTURES

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



(*a*) A simple 3-element stack.



(*b*) The list equivalent.

FIGURE 7-4
Stacks as lists.

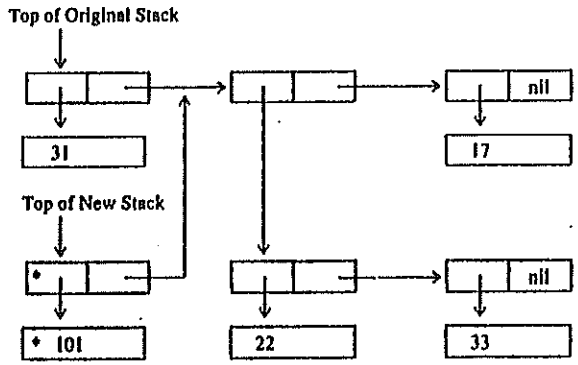
For example, if Figure 7-4(*a*) is implemented conventionally in sequential memory locations, popping 31 off and then pushing, say, 101, back on will cause the memory location holding the 31 to now contain 101. The 31 is lost.

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))

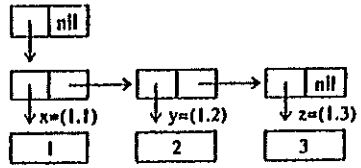


FIGURE 7-6
A simple value list.

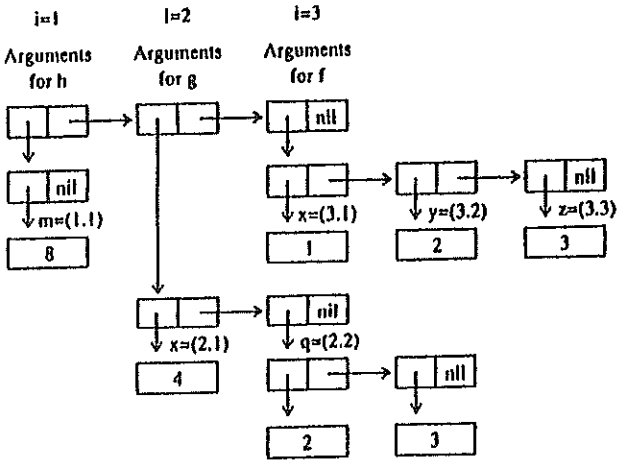
```
let h(m) = (3+m) ×9 in
  let g(x,q) = h(car(q) × x) in
    let f(x,y,z) = g(x+3, list(y,z)) in f(1,2,3)
```

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)))	((4 (2 3))
	((x.1) (y.2) (z.3)))	(1 2 3))
code for h	(((m.8)	((8)
	((x.4) (q.(2 3)))	(4 (2 3))
	((x.1) (y.2) (z.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)

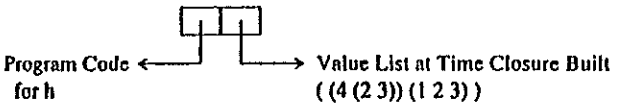


FIGURE 7-9
Closures as lists.

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

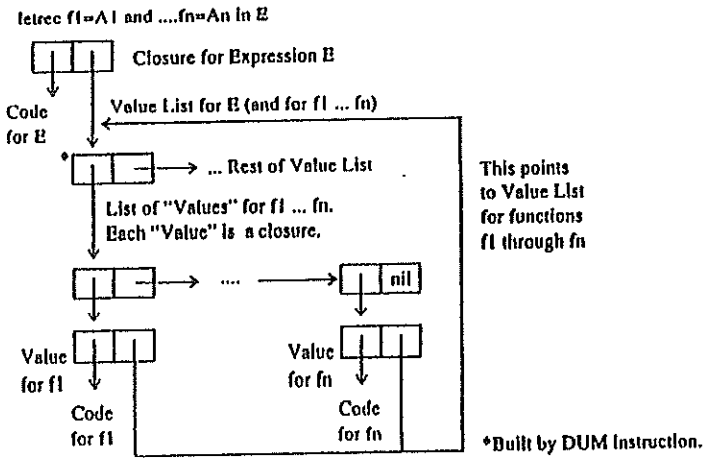


FIGURE 7-10
Value lists and closures for recursive expressions.

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

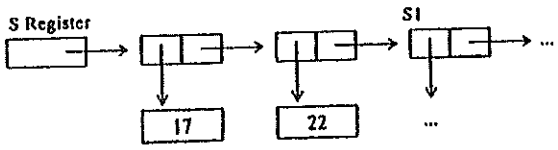
The *S register* points to a list in memory that is treated as a conventional evaluation stack for *built-in* functions, such as the standard arithmetic (+, −, ×, /) and list operations (car, cdr, cons). Objects to be processed by these functions are "pushed" on by doing a cons of a new cell onto the 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.

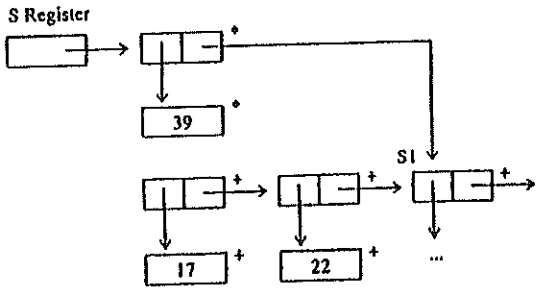
A key point is that, unlike a conventional stack built out of sequential memory locations, this new result does not overwrite the memory locations containing the original inputs. New cells are allocated both for the value and for the pointer cell in the list. The reason for this is that in various circumstances 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.



(*a*) Stack before an add.



*New cells allocated by add instruction.
+Not changed but no longer referenced after add complete.

(*b*) Stack after an add.

FIGURE 7-11
The S register and the stack.

As with the stack above, the prior value list designated by the E register is not overwritten by any change to E, and is still intact in memory. Other mechanisms, including the dump, often retrieve the old list when the current function completes.

7.3.3 The C Register

The *control pointer* or *C register* functions just like the *program counter* or *instruction counter* in a conventional computer. It points to the memory cell that designates through its car the current instruction to execute. In the SECD Machine these instructions are simple integers which specify the desired operation. Unlike many conventional computers, there are no specialized subfields for registers, addressing designations, etc. When additional information is needed for an instruction, such as which argument to access from the E register's list, the information comes from the cells chained to the instruction cell's cdr.

Conventional computers normally increment their program counter after completing most instructions. The analog in the SECD Machine is the replacement of the C register by the contents of the cdr field of the last memory cell used by the current instruction (C←cdr(C)). Again as with conventional machines, there are exceptions to this, particularly for

the equivalent of conditional branches, new function calls, and returns from completed application. Here the C register is replaced by a pointer provided by some other part of the machine.

7.3.4 The D Register

The *dump register* or *D register* is the last of the SECD Machine's registers. As with the S register, the D register points to a list in memory, this time called the *dump*. The purpose of this data structure is to remember the state of a function application when a new application in that function's body is to be started. This is done by appending onto the dump three new cells which record in their cars the values of the S, E, and C registers before the application. When the application completes, popping the top of the dump restores those registers to their original values so that the original application can continue. This is very similar to the *call-return* sequence found in conventional machines for procedure or subprogram activation and return.

7.4 THE BASIC INSTRUCTION SET

Figure 7-12 diagrams the basic instruction set for the SECD Machine. For each instruction there is a brief description of how the machine's four registers change after its execution. The notation used consists of four s-expressions before and after a "→." The four s-expressions before the "→" represent the assumed lists in memory pointed to by the S, E, C, and D registers just as the machine starts to execute the instruction. The s-expressions after the "→" represent the same four registers after the instruction has been executed.

As described earlier, the notation "(x y.z)" stands for an s-expression whose first two elements are x and y, respectively, with the rest of the expression z.

With this convention, all the original s-expressions for the C register consist of a list with the first element designated by the name of the instruction being executed. In real life each such instruction would be a cell containing a specific integer. Thus any cell containing a "2" in a program list might represent an LDC, while a "15" might represent an ADD. For readability here we will use a mnemonic form.

These instructions break down into six separate groups, namely, those that:

1. Push object values onto the S stack
2. Perform built-in function applications on the S stack and return the results to that stack
3. Handle the if-then-else special form
4. Build, apply, and return from closures representing nonrecursive function applications

Several variations of these data structures will be described in later chapters, particularly when we discuss lazy evaluation.

It is clear how the first of the above five categories, namely, s-expressions, can be built in the memory from the last section. Given that the SECD Machine contains instructions that perform the equivalent of cons, building a new s-expression involves allocating a new cell from memory (bumping the F register by 1), setting its tag to cons, and storing the appropriate pointers into its car and cdr fields.

The other data structures are addressed individually in the following sections.

7.2.1 Programs as Lists

Programs for the SECD Machine look like garden-variety lists. Each element of the list corresponds to an individual instruction, and execution proceeds (for the most part) one element at a time from the front of the list to the rear. A call to a function involves saving where one is in the current list and starting execution at the beginning of the list associated with the called function.

While it may seem wasteful to link together strings of often sequential instructions as a list (rather than as sequential words in an "array"), there are several significant advantages. First, we do not need to invent a new data structure just for program storage. Second, and most important, programs now look just like data, making it extraordinarily easy to write program that read, process, or even generate other programs. This makes writing compilers and interpreters for the SECD Machine in SECD code a relative snap.

The individual elements of a program list come in two types: simple integers or arbitrary lists. The former, simple integers, are equivalent to an *opcode* specifying some basic instruction in the SECD Machine's architecture. The latter, embedded sublists, usually represent alternative branches of program flow that will be decided dynamically when the program is run. Instructions which choose between these alternative flows of control correspond somewhat to the branch and call instructions found in conventional architectures. For example, to do an if-then-else, one element of a program's list will be a basic instruction (called out by an integer), which when executed will decide which of the two following elements of the program list should be executed. These following elements will be lists themselves. Each represents a then or else snippet of code. Instructions at the ends of the snippet returns control to the main thread.

Figure 7-3 gives a simple example.

7.2.2 Stacks as Lists

The SECD Machine uses several stacks during execution. One serves as an evaluation stack in a way reminiscent of *reverse Polish* execution.

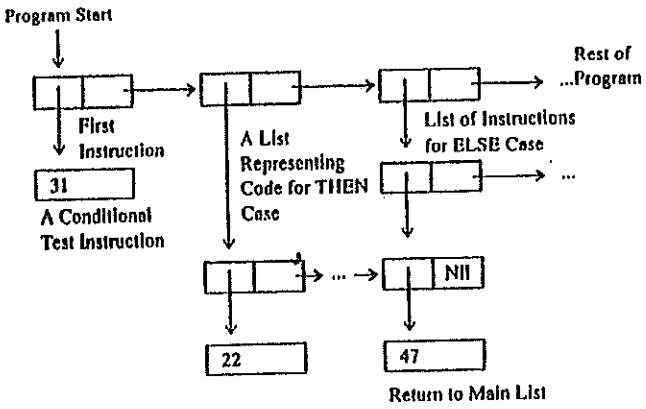


FIGURE 7-3
Programs as lists.

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 caar to get the element's value, and a cdr to return a pointer to the rest of the list. An empty stack is the nil list.

Figure 7-4 diagrams a simple example.

A subtle but important difference between these stacks and conventional stacks built out of sequential memory is that with lists a "pop" followed by a "push" does not overwrite the storage allocated to the element "popped" off. The new element is created in a separate memory cell, with the cell's cdr pointing to the cell holding the next element, wherever it is. The cell holding the original value "popped" off still exists, with its cdr pointing to the same next cell.

Instruction	Operation
	—Access Objects and Push Value to Stack—
NIL	s e (NIL.c) d → (nil.s) e c d
LDC	s e (LDC x.c) d → (x.s) e c d
LD	s e (LD (i.j).c) d → (locate((i.j),e).s) e c d
	—Support For Builtin Functions—
ATOM, CAR, CDR...	(a.s) e (OP.c) d → ((OP a).s) e c d
CONS, ADD, SUB,...	(a b.s) e (OP.c) d → ((a OP b).s) e c d
	—If-Then-Else Special Form—
SEL	(x.s) e (SEL ct cf.c) d → s e c? (c.d) where c? = ct if x≠0(T), and cf if x=0(F)
JOIN	s e (JOIN.c) (cr.d) → s e cr d
	—Nonrecursive Functions—
LDF	s e (LDF f.c) d → ((f.e).s) e c d
AP	((f.e') v.s) e (AP.c) d → NIL (v.e') f (s e c.d)
RTN	(x.z) e' (RTN.q) (s e c.d) → (x.s) e c d
	—Recursive Functions—
DUM	s e (DUM.c) d → s (nil.e) c d
RAP	((f.(nil.e)) v.s) (nil.e) (RAP.c) d → nil (rplaca((nil.e),v).e) f (s e c.d)
	—Auxiliary Instructions—
STOP	s e (STOP.c) d → s e (STOP.c) d -stop the machine
READC	s e (READC.c) d → (x.s) e c d where x is character read in from input device
WRITEC	(x.s) e (WRITEC.c) d → s e c d where x is printed on the output device

where rplaca(x,y) "replace car of x by y, and return pointer to x"
and locate(x,e) = loc(cdr(x), loc(car(x),e))
whererec loc(k,e) = if k = 1 then car(e) else loc(k − 1, cdr(e))

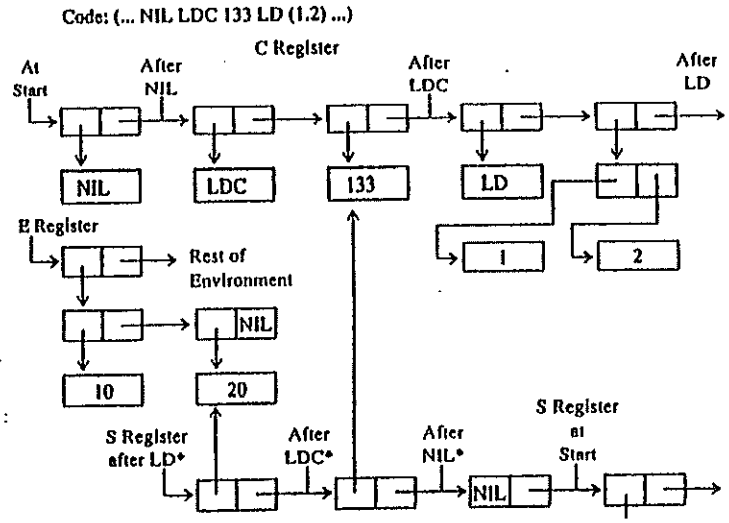
FIGURE 7-12
Basic instruction set for SECD Machine.

5. Extend the above to handle recursive functions
6. Handle input/output (I/O) and machine control

The following sections describe each of these groups.

7.4.1 Accessing Object Values

The first group pushes values of objects onto the S stack (see Figure 7-13 for an example). The first of these, *NIL*, pushes a nil pointer. Again this means that the S register after the instruction has been executed points to a newly allocated cell whose car contains a nil pointer (an address of location 0) and whose cdr points to the list designated by the S register before the instruction was executed.



*New cells allocated by instructions.

FIGURE 7-13
A sequence of data access instructions.

The second of these, *LDC*, loads a "constant" onto the stack. The constant value to be pushed is found as the next element on the C list after that for the instruction (the cadr of the C list at the time the instruction is started). Again a new cell is allocated, and chained onto the top of the S list. The car of this new cell is loaded with a pointer to this constant. Note that the value is not duplicated—only a pointer to it is. This means that this "constant" could in fact be an arbitrary s-expression, and the effect would be the same. The top of the stack would be that expression.

For this SECD Machine, the representation for the boolean constant F (false) is 0, and T (true) is taken as any integer other than 0, usually 1.

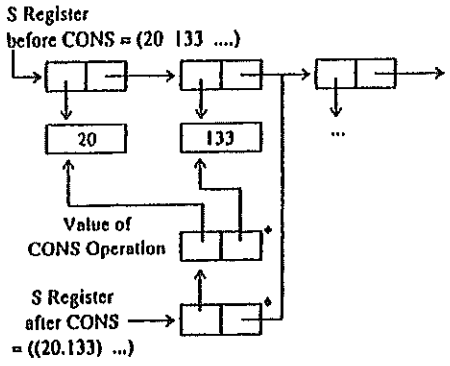
The third instruction of this group, *LD*, "loads" an element of the current environment. The cadr of the C list is a cell of the form (i.j), where the car i is the sublist element of the E list desired, and the cdr j is the element of that sublist. The function locate (see also Figure 7-8) describes how this access works. Again a pointer to that object is stored in the car of a new cell whose cdr points to the old S list.

7.4.2 Built-in Function Applications

The next group of instructions handle built-in functions such as CAR, CDR, CONS, ATOM, ADD, SUBtract,... Here the proper number of objects are accessed from the top of the S stack and the result placed in a new cell which is pointed to by the car of a second new cell. The cdr of this latter cell points to the rest of S's original list after the initial arguments. S is reset to point to this latter cell. Figure 7-11 diagrams a typical case for an ADD. A somewhat more complex example is the CONS instruction. This instruction allocates not one but two cons cells, one to hold the cons of the operands, and one to cons this result onto the S list. Figure 7-14 diagrams this, with Figure 7-15 diagramming the same situation in terms of possible memory location values.

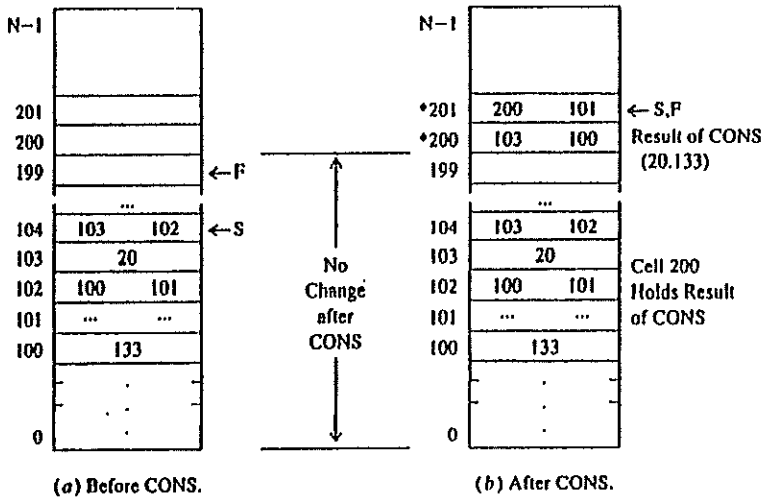
7.4.3 Instructions for If-then-else

The group of instructions used to implement if-then-else special forms are used in a specific order. The *SEL* instruction ("select") assumes that the top of the S list is an integer either zero or nonzero (encoded as described above to represent a boolean). Following the SEL in the control list are two elements (the cadr and caddr of the C list), both of which are themselves lists of instructions. The last instruction in each list is a *JOIN*. When executed, the SEL will push onto the dump a pointer to the C list just beyond the second sublist (i.e., a pointer to the cdddr of the original C list). The machine will then pop the top element off the S stack, test it, and replace C with its cadr if the value was nonzero, and with its caddr if the value was zero. Thus, these sublists correspond to the code for then and else expressions, respectively. The last instruction of each of these



*New cells allocated during CONS execution.

FIGURE 7-14
Execution of the cons instruction.



(*a*) Before CONS. (*b*) After CONS.

*New cells allocated by CONS.

FIGURE 7-15
Memory before and after a cons instruction.

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

Code fragment: if null (x) then 10 else −10

Program list (... NULL SEL (LDC 10 JOIN) (LDC −10 JOIN) ...)

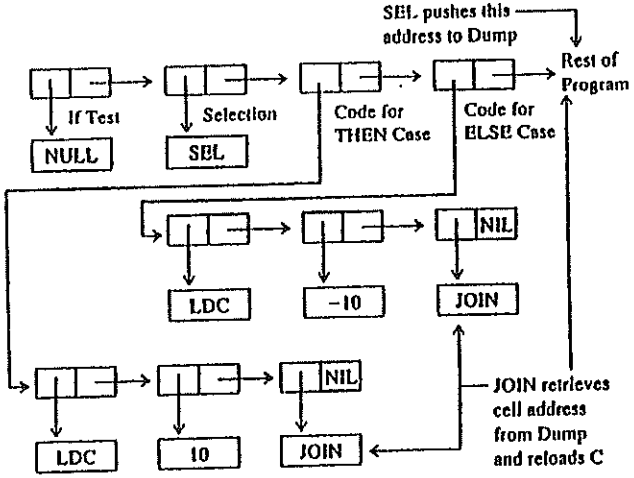


FIGURE 7-16
Structure of if-then-else code.

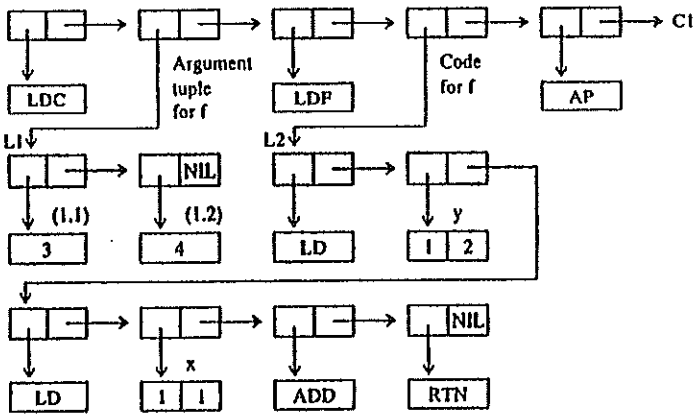
values to be applied to the function. Note that this argument list is a single element on the S list (namely, its cadr), and that the rest of the S list is arbitrary.

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.

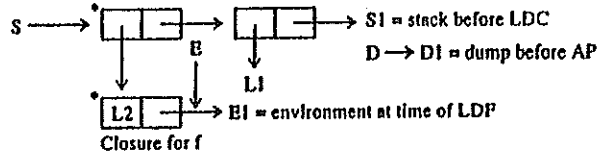
The last instruction of a function's code list should be a RTN. This instruction takes the top element off the S stack as the value to return from the application. This value is consed to the old S value previously pushed on top of the dump, with S reset to point to this list. The E and C registers are 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 ...)

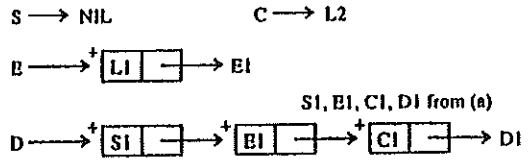


(*a*) Program.



*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 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

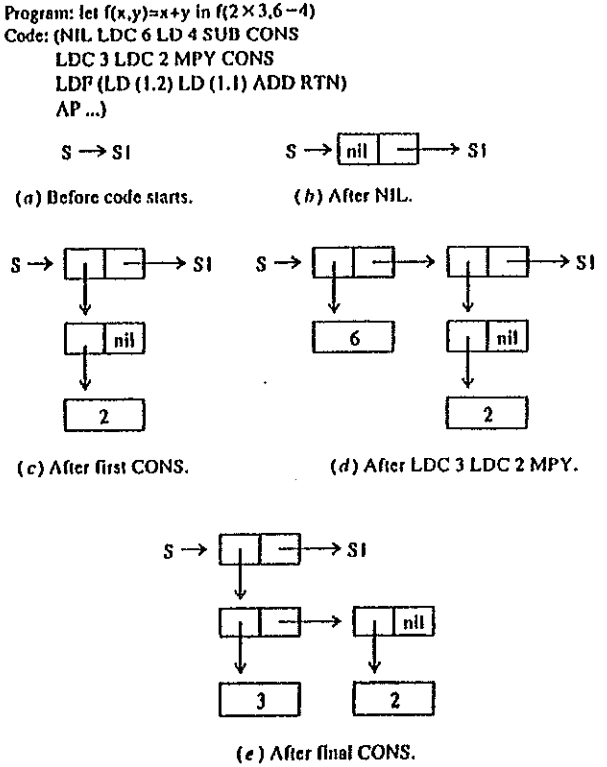


FIGURE 7-18
Construction of argument lists.

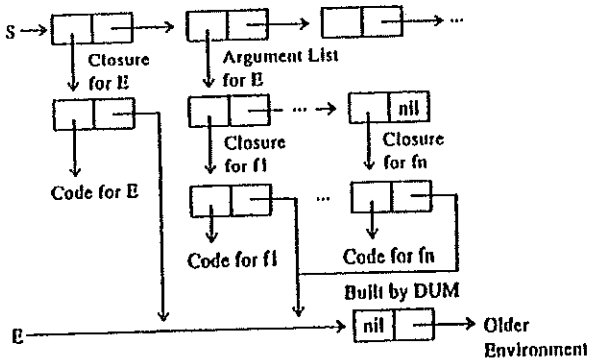
the cell pointed to by E (i.e., the dummy cell) is reset to point to the second argument of the S stack (the list of closures). Given that the closures in that argument also point to the dummy cell, the result is exactly the loop of pointers desired.

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.

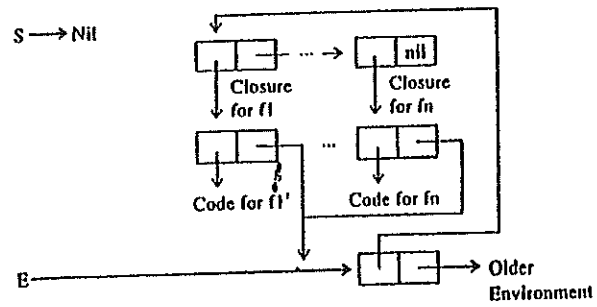
Again, a RTN at the end of the expression unstacks the original S, E, and C values stacked by the RAP function(s).

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>1
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>1 ||(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 varinble"
              cons(NIL, compile-app(args, n, cons(LD, cons(index(fcn, n), cons(AP, c)))))
       else "an application with nested function"
       cons(NIL, compile-app(args, n, compile(fcn, n, cons(AP, c)))
```

FIGURE 7-21
A compiler from s-expressions to SECD code.

7.5.2 Built-in Function Applications

The code generated for the application of *built-in functions* consists of the sequences of SECD code needed for each argument appended to the SECD instruction that performs the function. By convention, the rightmost argument in the s-expression form is computed first, and then execution proceeds from right to left. The net effect of this is that at execution time the topmost value on the stack is the leftmost argument, with later values further down the stack.

7.5.3 Conditional Forms

The compilation of a *conditional form* is an SECD code list consisting of the sequence of instructions that evaluates the test expression, followed

```
compile-builtin(args, n, c) =
  if null(args)
  then c
  else compile-builtin(cdr(args), n, compile(car(args), n, c))

compile-if(test, then, else, n, c) =
  compile(test, n,
          cons(SEL, cons(compile(then, n, cons(JOIN,nil)),
          cons(compile(else, n, cons(JOIN,nil)), c))))

compile-lambda(body, n, c) =
  cons(LDF, cons(compile(body, n, cons(RTN, nil)), c))

compile-app(args, n, c) =
  if null(args)
  then c
  else compile-app(cdr(args), n,
                   compile(car(args), n, cons(CONS, c)))

index(e,n) = indx(e, n, 1)

indx(e, n, i) =
  if null(n) then nil
  else letrec indx2(e, n, j) =
       if null(n) then nil
       elseif car(n) = e then j
       else indx2(e, cdr(n), j + 1) in
  let j = indx2(e, car(n), 1) in
     if null(j).
     then index(e, cdr(n), i + 1)
     else cons(i, j)
```

FIGURE 7-22
Auxiliary functions to compile special forms.

by a SEL instruction, followed by two lists which correspond to the then and else expressions, respectively. Each of these sublists is computed by recursively calling compile with the accumulating parameter initialized to (JOIN). This places the JOIN at the end of each sublist. The auxiliary function compile-if in Figure 7-22 performs this process.

7.5.4 Lambda Function Definitions

Programmed functions are any functions that are defined either explicitly or implicitly by a lambda expression. This includes ***lambda expressions, let expressions,*** and ***letrec expressions.*** The former simply defines a function; the last two include both function definition and their application.

The code compiled for a lambda expression consists of a two-element list, the first of which is a LDF instruction and the second of which is a list consisting of the compiled form of the lambda expression's body terminated by a RTN. As with the conditional forms, this RTN is

inserted by recursively calling compile with its third argument set to (RTN) (see compile-lambda in Figure 7-22).

Executing such a code sequence will push onto the S stack a closure whose code pointer is pointing to the sublist following the LDF and whose embedded environment is the cell pointed to by E when the LDF is executed.

When compile is called recursively for the lambda's body, the namelist argument has the list of argument identifiers for that lambda appended to the namelist passed into compile when the lambda was discovered. This reflects the fact that once inside the body of a lambda, the top of the value list on E must correspond to a list of the lambda's arguments, with the rest of the environment list consisting of the environment that was present before the function was applied. Further, the order of the identifier names in this sublist should be the same as the order in which the values will be when the code is executed. This permits a search of the namelist to return the proper (*i,j*) value to encode into LD code.

7.5.5 Combined Function Definitions and Applications

Let and letrec expressions correspond to evaluating a series of expressions, associating them with identifiers, and then evaluating a new expression that references these identifiers. The code compiled for these forms must compute each of these expressions and then cons the results together into a list that can be passed as an argument to the lambda function implied by the in expression.

Compile does this by pushing an initial nil onto the stack, and then generating code for the rightmost let expression that will leave the result of that expression above the nil. A CONS instruction combines these two into a single element list.

The process of compiling code for each of the remaining let expressions is similar. The process goes from right to left, compiling in a sequence of code to evaluate each subexpression, and following it by a CONS to append it to the previous expression list.

When such code is executed, it leaves on the top of the stack a single element which is itself a list. The order of the elements on this list corresponds directly to the order of let expressions, from left to right.

After this, the compiled code sequence consists of a LDF followed by a code list representing the in expression and terminated by a RTN. When executed, this pushes a closure representing the in expression on top of the value list.

For a let expression the final instruction compiled into the sequence is an AP. When executed, this instruction unpacks the closure for the in expression, and starts the body, with the computed values established on the top of E.

The only difference for a letrec expression is that a RAP is used in place of an AP, and there is an extra instruction to begin the sequence,

namely, a DUM. As discussed earlier, this helps build a dummy environment that RAP modifies to form the environment loop.

7.5.6 Nested Function Expressions

The final special form handled by the compiler is when the expression in the function position of an s-expression is something other than a keyword. The compiler function compile-app handles this case. As with let, the arguments are evaluated one at a time from right to left and 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)))))))
```

```
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL.compile-app((), ((fact)(x one)),
      (LDF
        (LDC 0 LD (1.1) EQ SEL
          (LDC 1 JOIN)
          (NIL LD (1.2) LD (1.1) MPY CONS
            LD (3.2) LD (1.1) SUB CONS
            LD (2.1) AP JOIN)
          RTN)
        CONS LDF
          (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
          RAP RTN)
      AP STOP))))))
⇒ (NIL.compile-app((3 1), nil,
    (LDF (DUM NIL LDF
      (LDC 0 LD (1.1) EQ SEL
        (LDC 1 JOIN)
        (NIL LD (1.2) LD (1.1) MPY CONS
          LD (3.2) LD (1.1) SUB CONS
          LD (2.1) AP JOIN)
        RTN)
      CONS LDF
        (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
        RAP RTN)
      AP STOP)))))
⇒ (NIL LDC 1 CONS LDC 3 CONS LDF
    (DUM NIL LDF
      (LDC 0 LD (1.1) EQ SEL
        (LDC 1 JOIN)
        (NIL LD (1.2) LD (1.1) MPY CONS
          LD (3.2) LD (1.1) SUB CONS LD (2.1) AP JOIN)
        RTN)
      CONS LDF
        (NIL LD (2.2) CONS LD (2.1) CONS LD (1.1) AP RTN)
        RAP RTN)
    AP STOP)
```

This final code is shown as a linked list of cells in Figure 7-23. For compactness, this figure takes any car field that contains a pointer to a terminal cell (tag=integer) and writes the value of that terminal where the pointer would go. In reality, the other terminal cell is still there.

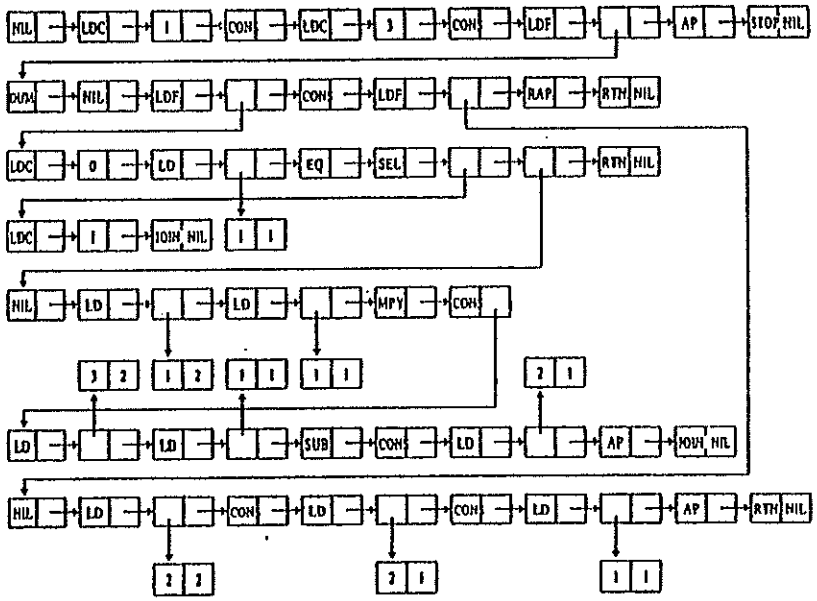


FIGURE 7-23
Shorthand cell equivalent of sample code.

7.7 THE FUNARG PROBLEM

(Bobrow and Wegbreit, 1973; Georgeff, 1982)

The SECD Machine uses stacks for almost everything. This leads to easy-to-understand compilers and relatively simple hardware implementations. It is also very similar to the abstract machines used for many conventional languages such as Pascal.

For someone schooled in conventional architectures, however, a direct implementation of the SECD Machine seems to have some significant inefficiencies. Each time an object is pushed onto the S stack, for example, the F register is incremented and two writes to the cell are performed (one for the data in the car and one for the cdr pointer). This work is doubled if the object being pushed is a new one, for example, the result of an ADD instruction.

In contrast, a conventional implementation of a stack involves simply incrementing a stack pointer and writing the data to memory. This is far cheaper in terms of machine cycles than the above procedure.

Even worse is the list of lists found on the environment. Accessing an argument via a LD instruction requires a doubly nested count down through these lists. A more normal implementation, such as for Pascal, would use a stack of contiguous locations for arguments and often a *display* of pointers into this stack to locate the beginning of an argument list

for each function call (see Figure 7-24). With such an implementation, accessing an argument takes only two indexed memory accesses (even fewer if the display is implemented as an array of registers inside the CPU).

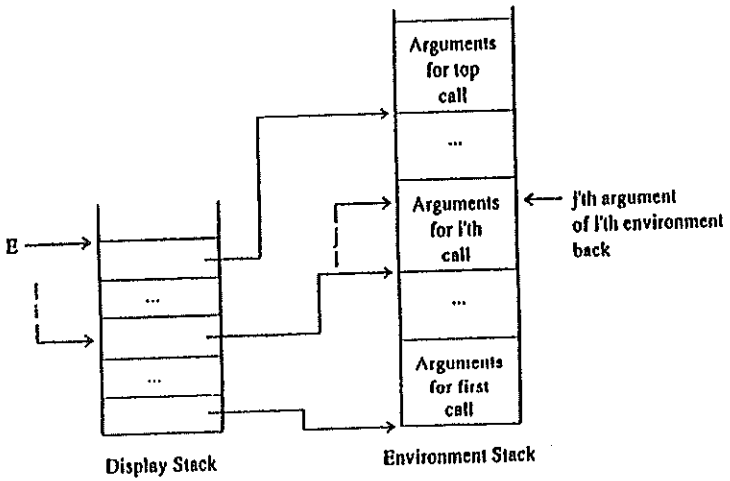
Given this, why would anyone use the SECD method of building stacks from linked lists? A good part of the answer stems from consideration of what happens when closures are generated by one function and passed as arguments to others, and particularly when identifiers within the closure function code have been bound values by functions which called the original closure-generating function. The handling of the stack and the environment becomes crucial, and if care is not taken, all kinds of strange results can occur. This has been studied extensively, and goes by the name of the *funarg problem*.

Consider, for example, the abstract program:

let f=(let y=4 in ($\lambda z \mid y \times z$)) in f(3)

The innermost let builds a valuelist ((4)). The result it passes to the outer let is a closure with a function code equivalent to ($\lambda z \mid y \times z$) and an environment (($y$.4)).

Consider what would happen if we built stacks, particularly the environment stack, as is done in conventional machines. Pushing an item simply causes an increment of the stack pointer and a store into the designated memory cell. Whatever is there from before is overwritten. Popping an object simply decrements the pointer, permitting the next push to overwrite the prior value. In such an implementation, at the exit from the



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.

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-