Chapter 2

Programming with Lisp

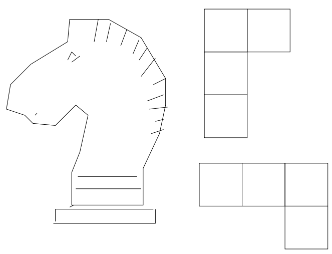
2.1 Symbolic Programming

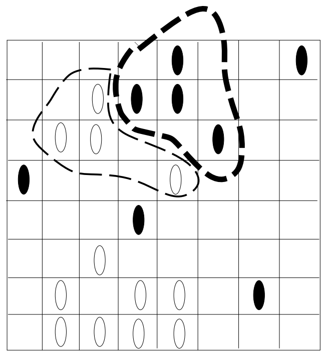
How do you build a house? You have to put up windows and doors, but first the frame of the house must be in place. Before you build the frame of the house, you need a foundation, and before you build a foundation, you must choose a site for the house and buy it. There are a number of steps involved in construction of a building, some of which cannot be done until previous steps are done. We need to plan these steps. In a typical programming language, the variables are snippets of memory like integers or strings. What if the elements in the program were steps in the plan to build a house? We call these *symbols* instead of variables.

A symbol has a name that we associate with a concept. You are already familiar with using the symbols $1, 2, 3$ to represent the mathematical concepts one, two, and three. These symbols have meaning, and we make up rules, like addition and subtraction, which allow us to manipulate the symbols and get other symbols. $2 + 2 = 4$. We use the mathematical symbols and their rules to calculate things about the world.

Symbols can also have complex meanings. Instead of dealing with the complex concepts directly, we manipulate their symbols. For example, a plan to build a house is composed of several smaller steps. We define symbols *window1* and *window2* to stand for two windows of the house, and *frame* to stand for the frame of the house. Just like there are addition rules for mathematical symbols, there are also rules about how windows and frames are placed together. And as with mathematics, calculating with these symbols tells us something about the real world, namely what a practical plan to assemble a house might be.

A game of chess can be analyzed by a symbolic program. Each chess piece is a symbol, under the appropriate rules of piece movement. For example, there are symbols for the four knights: *black_knight1*, *black_knight2*, *white_knight1*, *white_knight2*. The knight symbols can move in any hook pattern, and hop over pieces. For each move in the game, the computer will need to choose the "best move". At a higher level of abstraction, we want to think about moving groups of pieces into certain configurations. So we'd make symbols for standard chess attack and defense positions. Our program could manipulate symbolic expressions entirely at this higher level. For example, the last move in a game might be when one player has the symbol *king_in_check*, and finds a symbolic expression that gives the symbol *checkmate*. In the diagram, white has black in check, where the symbols for the circled groups of pieces are *De_Lomb_attack* for white, and *Monte_Cristo_defense* for black. One of the rules for the De Lomb attack and the Monte Criso defense is that if we have the symbol *king_in_check*, the operation "close the attack" will result in the symbol *checkmate*.





We are interested in *symbolic programming*, where we can encapsulate complex ideas with symbols, and manipulate the symbols directly. These programs deal with symbolic expressions such as $2 + 2 = 4$ or *window1* plus *window2* plus *frame2* is *house10*. An ambitious goal would be to have a program that proposed a scheme to assemble a 4 or a *house10* given available resources and some knowledge of the general symbolic rules. We are ultimately interested in writing programs that actually construct other programs and are even capable of rewriting themselves.

Throughout this text, we use a programming language *Lisp* that was designed for manipulating symbolic expressions. Lisp is the language traditionally used for artificial intelligence programming because many high level concepts can be expressed clearly. The symbols save the programmer from the low level details. Because the syntax of Lisp is simple, it is easy to learn and to write with.

◁ Lisp is a symbolic list processing language

Lisp was invented by John McCarthy at MIT before 1960, making it the second oldest computer language still in current use, after FORTRAN. Lisp was designed to facilitate symbolic programming. However, it was not the first language designed to manipulate lists and symbolic expressions; an earlier list-processing language called IPL was developed by Herbert Simon and Allen Newell for their work in automated problem solving (see Chapter /searchchapter).

McCarthy was able to show that a subset of Lisp was able to compute any function that a Turing machine could compute [McCarthy, 1960]. He did so by showing how one could write a *universal function* in Lisp that could interpret any Lisp function; McCarthy's universal function was called `eval`.

Lisp stands for <u>lis</u>t <u>p</u>rocessing language, because symbols are usually kept in ordered sequences formally called *lists*. For example, if building a house involved making the foundation, building a frame, and then installing windows, we might store the plan as a list of symbols: 1. *build_a_foundation*, 2. *build_a_frame*, 3. *install_windows*. Combinations of lists can make complex associations between symbols and groups of symbols. We'll show you how to use symbols in lists in the following sections.

Lisp does its own memory management, freeing you to build up arbitrary list structures and use them temporarily to perform computation. You can rely on Lisp to reclaim the associated storage when you are finished using them. The automatic reclamation of memory by any symbols is called *garbage collection*. Features like garbage collection make Lisp a wonderful language for rapid prototyping and exploratory study.

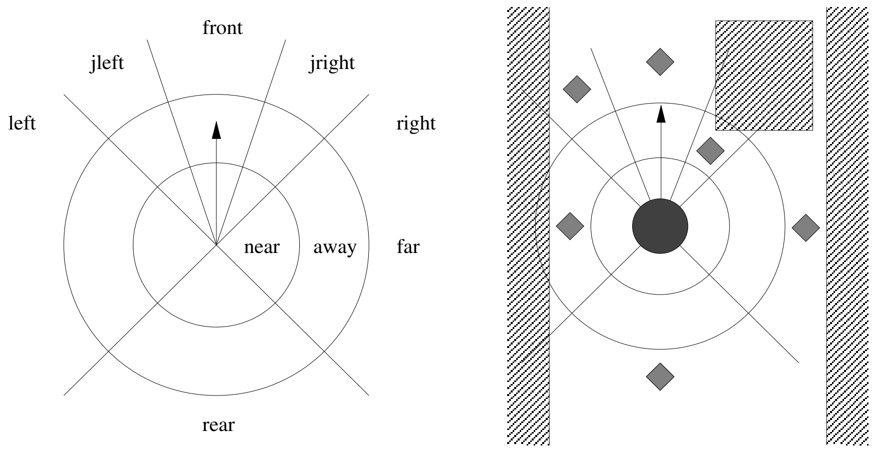
◁ How we describe Lisp

In this chapter, we assume that you already know how to program in at least one programming language. We encourage you to experiment with Lisp and try variations on what you see printed in these pages. Our introduction to Lisp proceeds as follows. We begin with basic syntax, and then talk about symbols. Lisp programs are represented as lists of symbols, so we provide the syntax for lists. We then turn to semantics and describe how the simplest of Lisp programs, corresponding to primitive data types, are interpreted. On the way to writing more interesting programs, we introduce constructs for procedures, functions and subroutine abstraction and flow of control, conditional and iterative. We describe some of the list manipulation techniques available in Lisp; list manipulation is an important part of Lisp in general and of our use of Lisp in particular. We provide the simple tools for data abstraction, to create structured data types. Finally, we describe in detail an example of solving a real world problem using symbols and rules.

2.2 Rule-Based Reactive Systems

A Mobile Robot Example

In this section, we examine a simple rule-based reactive control system for mobile robots. Such a system might be used to control a mobile robot to navigate in the corridors and open spaces of an office building. The system is said to be rule based because it uses a set of rules to determine what to



i. ii.

Figure 2.1: Obstacle detection sensors for a mobile robot

do next. It is said to be reactive because it responds more-or-less directly to changes in the environment indicated by the robot's sensors. We begin by introducing abstractions for sensors and controls.

Environment Abstraction

We assume that there are six sensors that provide information about presence of obstacles in the area around the robot. The sensors are arranged to provide more information about what is in front of the robot to assist in navigation and obstacle avoidance. The symbols `forward`, `justright`, `justleft`, `right`, `left`, and `rear` represent the six sensors. Each sensor is responsible for reporting on the presence of obstacles in a fixed region about the robot, returning one of three values from the set {`near,away,far`}. Figure **??**.i shows the arrangement of the sensors and Figure **??**.ii depicts the robot in a corridor of an office building with a water cooler partially blocking its way.

Robot Constraints

In addition to the sensors, there are two control parameters, the velocity of the robot and the direction that it is turning, corresponding to the symbols `speed` and `turn`. The `speed` parameter takes on values from {`zero`,`slow`,`fast`}. `Turn` takes on values from {`left`,`straight`,`right`}.

At this point we are prepare to create some data abstraction and work towards a solution utilizing LISP. However, we are not fully versed with LISP, so we will return to this example after you hold a greater knowledge of LISP and symbolic programming.

LISP is a Real Language

We begin our introduction to Lisp by considering what we want and need from a programming language. Certainly we want all of the standard data types (*e.g.,* numbers, strings, and pointers) and operations on such data types (*e.g.,* addition, multiplication, concatenation, and pointer following). In addition, we need flow of control constructs to facilitate sequencing, conditional branching, and procedure implementation. The means of reading and writing from files and standard devices such as terminals will be useful for any real applications. Finally, it will help in writing complicated programs if there are methods for structuring programs (*e.g.,* subroutines and modules) and facilities for creating new control constructs and abstract data types.

There is More Than One LISP

There are many dialects of Lisp and most provide all of the above and more. In the following, we employ a widely used dialect called *Common Lisp.* We choose Common Lisp rather than one of the more elegant dialects of Lisp as Common Lisp has become an accepted standard. We present a subset of Common Lisp and, as a result, much of what we say applies to other dialects as well. The particular subset of Common Lisp we have chosen includes all of the functionality required for the examples and exercises in this text and provides a good basis for learning more about Lisp. To help you use this subset we have included at the end of the book an index of general symbolic programming terms and Lisp notation including function names, special forms and special characters; the general terms are repeated in the general index but the Lisp notation is not.

LISP is Easy

The syntax of Lisp is relatively simple. It is so simple in fact that Lisp can seem structureless to programmers used to languages with greater syntactic restrictions. In Lisp, programs and data are represented as lists, where a list is represented by an open parenthesis followed by zero or more *expressions* and a closing parenthesis.

JON BOX HERE START!

Here are some sample list's that you might see in Lisp. Because of their construction they are all Lisp programs themselves.

`()` ; The empty (or *null*) list. `(1)` ; A single item list. `(1 (2))` ; A list which contains two items, one of which is ; a list also.

Lists provide an abstraction of pointers that simplifies a wide variety of programming tasks. Lisp also has numbers (integer and floating point) and strings (represented in the conventional manner between double quotes, `"string"`) and a variety of other primitive data types, but the ease with which programmers can manipulate lists in Lisp is one of its most attractive features. Lisp provides the programmer with a complete memory management system, thus eliminating the concern of memory allocation and deallocation. This is a clear benefit over other languages where memory management can absorb the entire program.At a later point in the text we will present lists and memory in Lisp in greater detail.

LISP Requires Documentation

Common Lisp provides a number of techniques for documenting code appearing in Lisp programs; we mention only one such technique. Typically, a Lisp program is stored in a file and consists of a sequence of function definitions and assorted other Lisp expressions interspersed with comments that provide documentation. The semicolon (`;`) is the standard comment character. Characters on the same line to the right of a semicolon are ignored, whether they are typed to the interpreter or found in a file. You have already see comments in Lisp, the first simple examples used the semicolon to distinguish between the code and the comments.

```
> ; This comment appears just prior to an expression
  (+ 3 4) ; This comment appears in line.
```

If you have access to the code that was written to accompany this text, then you will have many examples of well documented Lisp code. In this book, however, all documentation appears in the form of text either preceding or following the code it is meant to document.

2.3 Loading and Evaluating Programs

A Lisp program is just a sequence of Lisp expressions. Executing a Lisp program consists of evaluating the expressions in the program in the order that they appear in the sequence. If you write a set of expressions in a file, then you can execute it by *loading* it from the Lisp interpreter. Loading is done by evaluating an expression of the form `(load` *file-specification*`)` where *file-specification* is an expression that evaluates to a string indicating a file.

The exact form of the file specification will depend on your particular implementation of Common Lisp and local operating system. Generally, however, if you start Common Lisp in a directory in which a file 'program.lisp' resides, then evaluating `(load "program.lisp")` will serve to load that file and evaluate the expressions found in it. You can also embed `load` statements in files to establish dependencies between files.

2.4 A Simple Procedure

Common Lisp has a lot of built-in functions. You can scan through a reference manual (*e.g.,* [Steele, 1984]) to get some idea of the range of functions, but most if not all of the functions found in other widely used programming languages are also available in Common Lisp in some form or another. For arithmetic, Common Lisp includes functions for addition `+`, subtraction `-`, multiplication `*`, and division `/` involving one or more arguments. These functions can generally be counted on to behave reasonably given any mixture of integer or floating-point arguments.

```
> (+ 3 4 5) ; Simple addition of three numbers
12
> (- 4 2 1) ; Here 2 and 1 are subtracted from 4
1
```

Expression Evaluation

As with any language, understanding the process of expression evaluation is critical when learning Lisp. Fortunately Lisp provides an evaluation scheme that is common to mathematics. General expressions are evaluated left to right, expressions involving nested function calls are evaluated from the innermost to the outermost. For those expressions which have arguments that are themselves expressions, the nesting evaluation is carried out in left to right order. Figure **??** shows the tree of evaluations that results from

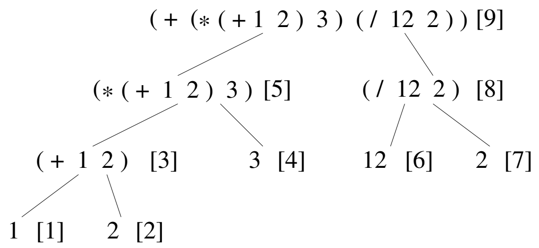


Figure 2.2: Order of evaluation in nested function calls

the following function calls; the order of evaluation for subexpressions is indicated by the integers in square brackets. The code is presented so that it is clear that the addition takes place between what is labeled list one and list two. List one is evaluated independently of list two prior to the overall summation.

```
> (+               ; Outer layer
    (*(+ 1 2) 3)   ; List one.
    (/ 12 2))      ; List two.
15
```

JON, BOX START HERE

```
> (*            ; Last evaluation
    (+ 4 3)     ; First list to be evaluated
    (/ 2 1))    ; Second list to be evaluated
14
```

2.5 Functions in LISP

You will also want to define your own functions. The examples of Lisp that have been presented thus far have included only operators that are defined by Lisp, and integers that are automatically assigned the appropriate value by Lisp. However, to Lisp, the operators and the integers that we have used are all symbols. The user defined functions that you implement use a special operator `defun` that takes a symbol as the name for the function, a list of symbols corresponding to the arguments (or formal parameters) of the function. The main body of the Lisp function follows, which consists of one or more well defined expressions in Lisp syntax.

```
> (defun square (x)  ; title line, and arguement
    (* x x))         ; operation line
SQUARE
> (square 3)         ; call of function
9

> (defun increment (x) ; title line, and arguement
     (+ x 1)           ; operation line
INCREMENT
> (increment 3)        ; call of function
4
```

The IF Construct

Lisp provides the common program control statement `if` found in most complete programming languages. Within Lisp syntax `if` is an operator, whose first arguement is evaluated. When the arguement is true the first list that follows is evaluated, otherwise the second list is evaluated. The example takes an arguement a returns the next odd number among integers.

```
> (defun next-odd-num (x)  ; Returns the next odd number after x.
    (if (= (% x 2) 0)      ; Uses the modulus operator (%) to determine
       (+ x 1)             ; if x is even.  When x is even add one,
       (+ x 2)))           ; otherwise add two.
```

The COND Construct

Lisp also provides a more general conditional construct, in other languages `cond` is analogous to a switch or case statement. When implemented a `cond` statement is considered an operator with multiple two part clauses, `(cond` *clauses*). The first portion of each clause is a test statement to be evaluated, the second portion is a body to be executed when the test statement true, (*test body*). A `cond` statement is evaluated in the same manner as arithmetic expressions, from left to right, evaluating the test of each of its clauses in turn until one test succeeds (*i.e.,* returns something other than `nil`). When a test succeeds the expressions in the body of the same clause are evaluated in sequence and the value of the last expression is returned as the value of the `cond`. If the clause has no expressions in its body, then the value of the test is returned. If no test succeeds, then the `cond` returns `nil`.

The following function illustrates the use of `cond`. `Let` is an operator that assigns values to symbols, in this case `response` will be assigned the value of `read` and number will be assigned the results of ycffrandom. `Read` is a function of no arguments that reads an expression from the standard input (usually a terminal), and `random` is a function that takes a single argument corresponding to a positive number n and returns a number of the same type — in this case an integer — between zero (inclusive) and n (exclusive).

In the case of n being an integer, the possible results appear with the (approximate) frequency $1/n$.

```
> (defun guess ()                                           ; Title line
    (princ "Guess an integer from 0 to 9: ")                ; Terminal display
    (let
      ((response (read)) (number (random 10)))              ; Variable assignment
      (cond
((> response number) (princ "Too high!"))
         ((< response number) (princ "Too low!"))
         (t (princ "Lucky guess!")))))                      ; Default is t
GUESS
> (guess)
Guess an integer from 0 to 9: 3
Too low!
```

The `3` was typed by the user in response to the prompt. In the function `guess`, `t` (a symbol that evaluates to itself) plays the role of a test that always succeeds. It is considered by some bad form to write a `cond` statement for which no test succeeds, and, hence, you will often see `cond` statements whose last clause is of the form `(t` *body*`)`.

2.6 Recursion and Iteration

A *recursive* function is one in which the function is called recursively in the body of its definition. In the recursive call, the function is generally applied to some reduction of the original arguments. In addition, there is some criterion that determines when further recursive calls are no longer required.

Simple Recursion

The function `raise` takes two arguments, a number of any type and a non-negative integer n, and returns the number raised to the nth power. A recursive definition is provided below.

```
> (defun raise (x n)                ; x raised to the power n
    (if (= n 0)                     ; check for end of recursion
      1                             ; at base return a 1
      (* x (raise x (- n 1)))))     ; otherwise x * x raised to the
                                    ; power n-1
RAISE
> (raise 3 3)
27
```

The base-case criterion (terminating condition) for the recursion is the case of $n = 0$; any number raised to the 0th power is 1. If $n \neq 0$, then we multiply `x` times the result of calling `raise` with arguments `x` and `(- n 1)`. Figure **??** depicts the environments as they are created throughout the execution of `(raise 3 2)`.

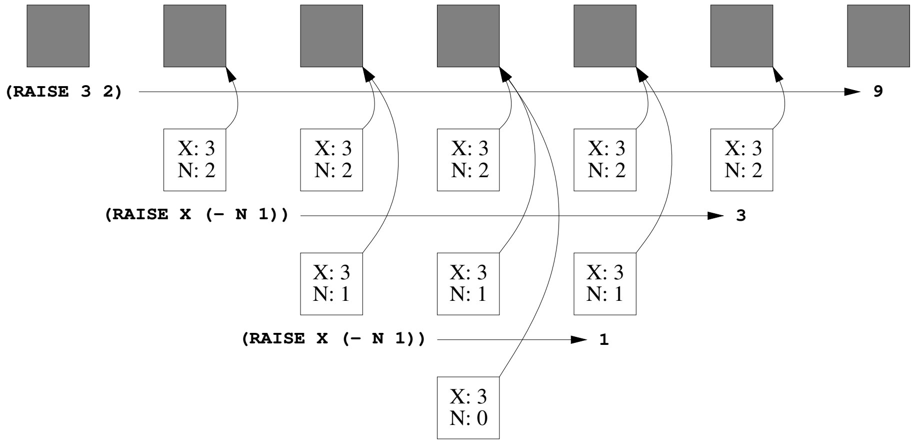


Figure 2.3: Environments created during recursive function invocation

2.7 Iterative Constructs

The `do` construct is one of the most widely used iterative constructs in Common Lisp. The general form is `(do` *index-variable-specifications* `(`*end-test result*`)` *body*`)`, where *index-variable-specifications* is a list of items of the form `(`*step-variable initial-value step-value*`)`, *end-test* is any expression, and *result* and *body* consist of one or more expressions. The step value or the step value and the initial value can be left out of an index-variable specification; in the latter case, you need not enclose the step variable in parentheses. Upon entering the `do` each step variable gets its initial value or `nil` if no initial value is provided. On each subsequent loop through the `do` each step variable gets its step value. The step variable assignments are carried out in parallel.[1] On each loop, after the step variables are assigned, the end test is evaluated. If the end test returns non-`nil`, the result expressions are evaluated in order returning the value of the last one, otherwise the expressions

[1]A variant construct `do*` is identical to `do` except that variable assignments are carried out sequentially.

in the body are evaluated in order. Here is a simple example of a `do` loop computing 9! and printing out the numbers 1 through 9 as a side effect.

```
> (do
    ((i 1              ; i is an index variable, initialized to 1
       (+ i 1))        ; i increments by 1
     (j 1              ; j is an index variable, initialized to 1
       (* j i)))       ; j increments by j * i
((= i 10) j)  ; end when i = 10, j = any value
(princ i))    ; print i's while looping.
123456789
362880
```

There are other iterative constructs corresponding to special cases of `do` that represent common patterns of use and are often convenient. `Dolist` is generally called using the form `(dolist (`*var expr result*`)` *body*`)`, where *var* is a symbol repeatedly bound to the elements of the list that results from evaluating *expr*, *body* is evaluated once for each element of the list, and *result* is an optional form that is evaluated and returned as the value of the `dolist` after the last time the *body* is evaluated. If no *result* is provided then `dolist` returns `nil`. `Dotimes` is generally invoked using the form `(dotimes (`*var expr result*`)` *body*`)`, where in this case *var* is bound to the integers zero up to (but not including) the integer resulting from evaluating *expr*. Here are some examples illustrating these iterative constructs.

```
> (dolist
    (x '(a b c)) ; Here x will equal a,b, and then c
    (princ x))   ; During the process the value of x will be
                 ; displayed.
ABC
NIL
> (dotimes
    (i 10 i)     ; Here i will equal 0 through 10
    (princ i))   ; During the process the value of i will be
                 ; displayed.
0123456789
10
```

In the following chapters, we make use of mapping functions, a Lisp specific operation, recursion, and the above iterative constructs to illustrate different styles of programming. There is no one way to write the programs listed in this text and you should experiment to find a coding style that you feel comfortable with.

2.8 Symbols, Scope, Environment

Symbols

Another important concept in Lisp is that of the *symbol*. Symbols have *values* that can be changed and depend on the context in which they appear. In the previous section the user defined function guess used two symbols, number and response. Lisp utilizes symbols to represent values, or even functions. In the case of user defined functions the name of the function is actually a symbol which is considered by Lisp as the function.

Here are several non-funciton symbols. `sym` ; basic symbol `sym14` ; alphanumeric mix ok `sym-one` ; hyphen `sym_two` ; underscore

In lisp the case of the symbol is ignored, the following three symbols are all considered the same symbol by Lisp.

`foo` ; lower case `FOO` ; upper case `Foo` ; mixed

Expressions

An expression, then, is a number, string, symbol, or (inductively) a list composed of zero or more expressions. To make any further progress, you will have to learn more about the semantics of Lisp. As we learn more about the meaning of Lisp programs, we will introduce more of the syntax. In the following, we describe a program called `eval` that serves to interpret or evaluate Lisp programs.

Conveniently, most implementations of Lisp provide an interactive program that allows the user to type Lisp expressions to a terminal interface to be evaluated. We make use of this interactive program (called the *interpreter*) in the following presentation and suggest that, if possible, you follow along at a terminal performing experiments of your own. Consult a local Lisp hacker to find out how to invoke the interpreter and how to recover from the inevitable errors that will occur. Don't be afraid to experiment; the interpreter is designed to facilitate exploration.

When a Lisp interpreter starts up, it generally prints out a message indicating the version number, restrictions on copying, and a variety of other information that you can safely ignore at this stage. When it is finished with its greeting, it displays a prompt indicating that you can begin typing. The interpreter is expecting a complete expression. If you type such an expression followed by a carriage return, the interpreter reads it, evaluates it, and prints out the result. Nested lists are often the most difficult expressions to manage as correctly typing such expressions requires that the parentheses balance; a feat that many fledgling Lisp hackers find difficult to accomplish. We will not persist in giving you advice about how to interact with your interpreter, except to suggest once again that you consult a manual or local expert regarding your implementation of Lisp. You may find that your local Lisp system includes utilities for balancing parentheses, controlling indentation, and a variety of other aids that will make your experience more productive and less frustrating. In the following, we elided the many errors and false starts that occurred as we generated the interactions recorded here.

A Lisp program is just an ordered sequence of expressions. Running a Lisp program consists of evaluating the sequence in order. The Lisp interpreter invokes `eval` which behaves much as a compiler does in other programming languages; the fact that `eval` can incrementally compile Lisp programs considerably simplifies writing Lisp code. (Common Lisp supports other forms of compilation besides those implemented by `eval` but we do not consider them in this text.) Strings and numbers evaluate to themselves.

```
> "string"
"string"
> 3.14
3.14
```

Symbols and the Environment

Symbols are a different matter. If you just type a random symbol to the interpreter, you will probably get an error. Note that the particular interpreter used here displays all symbols using upper case letters only; this is the default mode in many Lisp implementations.

```
> sym
Error: The symbol SYM has no global value
```

Global Symbols

Some symbols, however, do have global values. In particular, the symbols `t` and `nil`, which are the default Lisp boolean values, evaluate to themselves. `Nil` and the empty list `()` are equivalent in Common Lisp in the sense that they both point to the same location in memory.

Changing Values of Symbols

We can set or change the value of other symbols using `setq` (for set equal).

```
> (setq sym 2)
2
> sym
2
> (setq sym 3)
3
> sym
3
```

`Setq` takes a minimum of two arguments; the first must be a symbol and is not evaluated and the second can be any expression and is evaluated.

```
> (setq new sym)
3
> new
3
```

`Setq` can be used to set the value of several symbols at once as in the expression `(setq new 1 old 0)`.

Symbols and the Environment Table

The value of a symbol depends on the context in which it is evaluated. When `eval` encounters a symbol, it looks up the symbol's value in a structure that behaves like a table; if there is no entry in the table, then it reports an error. The structure is called an *environment* and `eval` has to figure out the appropriate environment to look in. In the examples above, `eval` looks up symbol values in the *global* environment. Later on, we consider how additional environments are created and referenced by `eval`.

2.9 Functions and Procedural Abstraction

If the expression typed to the interpreter is not string, number, symbol, or one of Lisp's special forms, then it had better be a list. `Eval` assumes that a list — one not specifying a special form — indicates the application of a function[2] (the first element of the list) to some number of arguments (the remaining elements of the list).

[2]We use the terms 'function' and 'procedure' interchangeably in this text.

Functions are Symbols

Functions are notated as symbols. In Common Lisp, a symbol can have a value that is used when the symbol appears as an argument of a function and a definition that is used when the symbol appears as the first element of a list interpreted as a function call.[3] For our purposes, we assume that function definitions are stored in a single global table. This need not be the case in Common Lisp but making this assumption avoids a number of what we consider peripheral issues.

Functions, Arguements and the Environment Table

The argument symbols are handled in a special way during function application. Each time a function is applied, a new environment is created and the function definition, consisting of one or more expressions, is evaluated in that environment. Figure **??** describes the operation of `apply`, the procedure responsible for handling function application for `eval`.

All of the Environments Work Together

An environment allocates storage for symbol values. You can think of the global environment as just a large table. Whenever you `setq` a new symbol, memory is set aside in the global environment to point to the value of that symbol. The environments created during function application and in evaluating certain special forms are somewhat more complicated. In general, an environment is a sequence of tables. A new environment is created from an existing environment by creating a new table and having it point to the

[3] Not all modern dialects of Lisp make this distinction between functions and arguments. For example, there is a dialect of Lisp called Scheme [Abelson & Sussman, 1985] in which each symbol has a single value resulting in a very clean and conceptually simple implementation.

existing (*parent*) environment. The new environment points to this newly created table which allocates storage for specific symbols (*e.g.,* the formal parameters in the case of a function definition). To determine the value of a symbol in an environment, `eval` first looks in the table pointed to by the environment. If there is no entry for the symbol in that table, then `eval` looks in the parent environment, and so on until it reaches the global environment. You can change the (local) value of a symbol in an environment without changing its value in the global environment.

```
> (setq x 7)           ; Create x globally
7
> (defun local (x)     ; New environment x created here.
    (setq x (+ x 1))   ; Variable x changed for current environment.
    (* x x))           ; Operation on local variable x.
LOCAL
> (local 2)            ; Call function with arguement.
9
> x                    ; Value for global x remains the same.
7
```

Don't Try This in Your Environment

Generally speaking, setting the value of symbols in the global environment is frowned upon. One of the main reasons for this attitude is that global variables are often difficult to track down in large pieces of code; widely scattered global variables can make large programs difficult to understand.

```
> (defun global (x)      ; Create x locally.
    (setq sym (+ x 1))   ; Set global sym to value x.
    (* sym sym))         ; Perform operation here.
GLOBAL
> (global 1)
4
> sym
2
```

Let, a Variable and Environment Modifier

You do not have to introduce a symbol as a formal parameter in order to use it as a local variable. The special form `let` can also be used to produce a new environment. Using `let` you can introduce a local variable and later `setq` it to a specific value or you can set it at the same time you introduce it. The general form is `(let` *variable-specifications body*`)`, where *variable-specifications* is a list of zero or more expressions each corresponding to a variable or an expression of the form `(`*variable initial-value-expression*`)` and *body* consists of one or more expressions. In evaluating such a form, Lisp assigns each variable the value of the corresponding initial value expression if such an expression exists and `nil` otherwise. These assignments are carried out in parallel so that you cannot refer to one variable in the initial value expression of another. There is a variant of `let` called `let*` that assigns local variables sequentially so that you can refer to one variable in the initial value expression of another variable appearing later in the list of variable specifications.

Interpolate, the Lisp Way

In the following, we illustrate both styles of setting local variables in the definition of a simple linear interpolation function that takes two points on a line in terms of their `x` and `y` coordinates and returns the `y` coordinate of a third point given its `x` coordinate.[4]

[4] In many cases (present example included), local variables are not strictly necessary. However, if skillfully employed, local variables help to enhance readability.

```
> (defun interpolate (x1 y1  ; point one
                      x2 y2  ; point two
                         x3) ; x coordinate of new point
    (let (m (b y1)           ; b,m assigned value of y1
         (x (- x3 x1)))      ; x assigned value of x3 - x1
      (setq m                ; m assigned value of / of the
            (/ (- y2 y1)     ; y2 - y1, and
               (- x2 x1)))   ; x2 - x1, to determine slope.
      (+ (* m x) b)))        ; Finally calculate y coordinate.
INTERPOLATE
> (interpolate 1 1 5 5 4)
4
```

Multiple Environments With Let

We can nest `let` statements thereby producing structured environments. The *scope* of a variable introduced through an environment is determined by the balanced parentheses enclosing the corresponding `let` or `defun`. For this reason, such variables are often referred to as *lexical* (or *static*) variables, and the rule for determining their scope as *lexical* scoping. Many dialects of Lisp also support *dynamic* variables whose scope is determined at evaluation time. Lexical scoping has the advantage that code relying entirely on lexically scoped variables tends to be easier for programmers to understand and compilers to produce efficient code for. In this text, a variable is either global or local, and if it is local, then it has lexical scoping.

Let, Scope, and Environment

In the following, we use a simple form of Lisp print function to illustrate variable scoping. The function `princ` evaluates its single argument, prints the resulting value, and then returns that value.

```
> (let ((x 1))          ; x = 1
    (let ((x 2))        ; x = 2
      (let ((x 3))      ; x = 3
        (princ x))      ;
      (princ x))        ; x = 2
    (princ x))          ; x = 1
321
1
```

The first line following the `let` statement is the result of the three `princ` calls; the second line is the value returned by the `let` statement which is the value of the last `princ` statement.

The definition for the interpolation function that we introduced earlier has a bug in it. If the two points have the same `x` coordinates then the the previous definition will result in an attempt to divide by zero which will cause an error. We can correct this by adding a conditional statement. In Lisp, any expression can serve as a test; if the expression returns `nil` (or equivalently `()`), then the test fails, otherwise it succeeds. The Lisp `if` statement consists of a test, an expression evaluated only if the test succeeds, and an expression evaluated only if the test fails. The predicates `=`, `>`, `<`, `>=`, and `<=` enables us to compare numbers.

```
> (defun interpolate (x1 y1    ; First point
                      x2 y2    ; Second point
                         x3)   ; Goal X Coordinate
    (if (= x1 x2)              ; When x1 = x2
      y1                       ; The y coordinate is y1,
      (let ((m                 ; otherwise interpolate
              (/ (- y2 y1)     ; as before.
                 (- x2 x1)))
             (b y1)
             (x (- x3 x1)))
        (+ (* m x) b))))
INTERPOLATE
> (interpolate 1 3 4 3 2)
3
```

2.10 List Processing

Contradictory as it might sound, a symbol does not have to have a value in order to be valuable for symbolic manipulation purposes. In order to refer to a symbol (rather than its value) it is often useful to get `eval` to suspend evaluation. This is done in Lisp with the `quote` function. `Quote` is used so often in Lisp that there is a convenient abbreviation; `'`*expression* is an abbreviation for `(quote` *expression*`)`. `Quote` causes `eval` to suspend evaluation on any expression.

```
> (quote sym)
SYM
> 'sym
SYM
> '(first second third)
(FIRST SECOND THIRD)
```

2.11 Memory

Using `quote`, we can construct lists of symbols if we know what those symbols are in advance. Using `cons` (for <u>cons</u>tructor), we can build lists more flexibly under program control. `Cons` takes two arguments and constructs what is called a *dotted pair* for reasons that will soon become apparent; construction involves allocating storage for two *pointers* that refer to the values of the two arguments.

```
> (setq x
    (cons 1 2))
(1 . 2)
> (setq y (cons 1
              (cons 2 ())))
(1 2)
```

The structures in memory corresponding to dotted pairs are called *cons cells* and are depicted graphically as joined boxes with pointers. Pointers

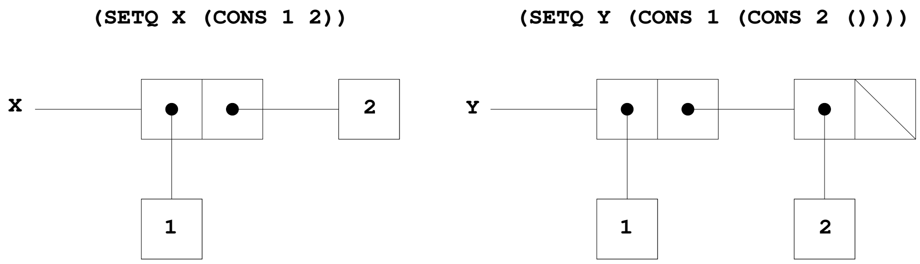


Figure 2.4: List structures in memory

to `nil` are depicted by a slash through the corresponding box. Figure **??** shows the structures resulting from evaluating the expressions above. `Car` and `cdr` allow us to select parts of list structures constructed using `cons`. The names `car` and `cdr` can be traced to the machine architecture of the computer that Lisp was first designed to run on.

```
> (car x)
1
> (cdr y)
(2)
> (car (cdr y))
2
```

`First` and `rest` are more mnemonic alternatives to `car` and `cdr`. `Second`, `third`, and `fourth` are also commonly defined in many dialects of Lisp, and, more generally, `(nth` i l`)` allows us to access the ith element of the list l.

One reason that we retain the archaic `car` and `cdr` is the related but very convenient abbreviations for nested `car`s and `cdr`s. In many dialects of Lisp, short sequences of `car`s and `cdr`s (usually up to four) (*e.g.,* `(car (cdr (car` *expression*`)))`) are abbreviated by functions of the form `c[a|d]`*`r` (*e.g.,* `(cadar` *expression*`)`). If these functions are not available in your dialect, you should find it easy to define them. The function `list` provides a somewhat more convenient method of constructing lists. `List` takes any number of arguments and returns a list of their values. The invocation `(list 1 2`

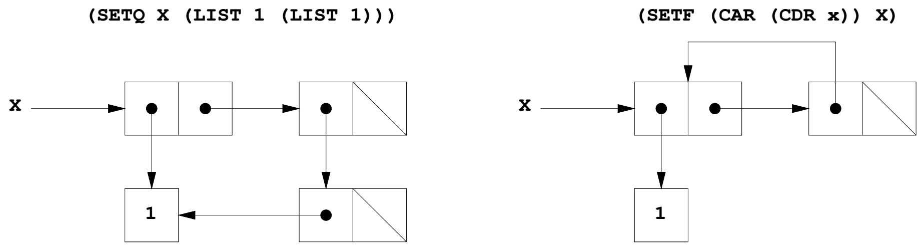


Figure 2.5: Modifying list structures in memory

`3 4)` results in the same structure as `(cons 1 (cons 2 (cons 3 (cons 4 ()))))`.

```
> (list 1 2 3 4)
(1 2 3 4)
> (list 1 (list 2 (list 3)))
(1 (2 (3)))
```

Lisp allows you to modify existing list structures by changing the contents of memory. The is especially useful when more than one symbol points to the same locations in memory. `Setf` takes a location in memory (such as provided by a symbol reference or by `car` and `cdr` invocations) and an expression, and changes what is stored at that location to be the value of the expression. `Setf` is said to *destructively* modify its first argument.

```
> (setq x (list 1 (list 2)))
(1 (2))
> (setf (car (car (cdr x))) 1)
1
> x
(1 (1))
```

`Setf` is more general than `setq` but the latter is often still used as a form of documentation to indicate that a symbol is being set rather than some more complicated memory modification. Be careful with `setf`; employed carelessly, it can often be the cause of subtle bugs in programs. `Setf` allows us to build circular list structures that are very useful for representation purposes; you do, however, have to me somewhat careful in displaying such structures.

```
> (setf (car (cdr x)) x)
(1 (1 (1 (1 (1 (1 (1 (1 (1 (1 (1 ...
```

The interpreter's response to the above invocation had to be interrupted or it would have continued printing indefinitely. Figure **??** displays the resulting structure graphically. Common Lisp has print procedures that perform a more reasonable job of printing circular list structures.

Lisp passes parameters using call by value as illustrated by the following exchange with the interpreter.

```
> (defun foo (x)
    (setq x 0))
FOO
> (let ((x 1))
     (foo x)
     (princ x))
1
```

We can easily simulate other parameter passing conventions using pointers and `setf`. The following illustrates how to simulate call-by-reference parameter passing.

```
> (defun bar (x)
    (setf (car x) 0))
BAR
> (let ((x (list 1)))
       (bar x)
       (princ (car x)))
0
```

It should be noted, however, that there is no way in Lisp to achieve call-by-reference for symbols.

Lisp has a variety of boolean predicates that return `t` or `nil` to test the type of Lisp objects. `Listp` tests for lists, `consp` tests for dotted pairs, `numberp` tests for numbers, `oddp` and `evenp` for odd and even integers, `symbolp` for symbols, and `null` for the empty list `nil`. In addition, there are the obvious logical operators: `and`, `or`, and `not`. To compare list structures, there are `eq` and `equal`. `Eq` determines if its two arguments point to the same location in memory. `Equal` determines if its two arguments are structurally similar.

```
> (setq sym 'foo)
FOO
> (eq sym 'foo)
T
> (setq sym (list 'foo))
(FOO)
> (eq sym (list 'foo))
NIL
> (setq new sym)
(FOO)
> (eq new sym)
T
> (equal sym (list 'foo))
T
```

2.12 Functions with Local State

It is often useful to associate data (or *state*) with a particular function or set of functions. One way of doing this is to create one or more global variables and `setq` them to appropriate memory structures. As noted earlier, however, this use of global variables is generally frowned upon by purists. As an alternative, we might create an enclosing environment for the function or set of functions that introduces variables local to the environment that refer to the local state.

Functions and Extent, How Long Does it Last?

The time during which an environment exists is called its *extent*. So far, the extent of the environments we have talked about is brief, usually just the time it takes to execute a procedure. There is one notable exception. The extent of the global environment is the entire time the Lisp process is running. Whenever a Lisp object is created that is capable of making references to symbols, that object maintains a pointer to its immediate environment so that references can be made with respect to that environment. As long as that object exists, its associated environment exists. So far, the only objects we have encountered that are capable of making variable references are functions, but shortly we will learn how to create other such objects.

Functions with Different Environments

All of the functions, we have considered up until now have as their associated environment the global environment, but we can easily define functions with different environments. The following (novelty) function, `squarelast`, uses local state to remember the number corresponding to the value of its single argument the last time it was invoked and returns the square of that number. `Squarelast` is always one step behind the user.

```
> (let (x (y 1))                  ; Establish environment
    (defun squarelast (z)         ; Create Internal Function
      (setq x y)                  ; Variable manipulation
      (setq y z)
      (* x x)))                   ; Operation
SQUARELAST
> (squarelast 2)
1
> (squarelast 3)
4
```

A Function that Remebers

As a somewhat more interesting use of local state, we can define a pair of functions that remember pairs of numbers of the form $(x\ .\ f(x))$ for a function f, and, when asked to provide an estimate for an arbitrary value

x, returns the $f(x)$ if it has stored a pair (x . $f(x)$) or invokes the linear interpolation function described earlier otherwise.

```
> (let ((data ()))                              ; Establish Environment
    (defun remember (x y)                       ; Define Global Function
      (setq data (insert (cons x y) data)))     ; Creates a List of Pairs
    (defun estimate (x)                         ; Define Global Function
      (let ((pairs (nearest-pairs x data)))     ; Estimate's Local Environment
        (cond ((null pairs) x)                  ; With No List, No response
              ((null (rest pairs))
                     (rest (first pairs)))
              (t (interpolate
                     (first (first pairs))
                     (rest  (first pairs))
                     (first (rest pairs))
                     (rest  (rest pairs))
                  x))))))
ESTIMATE
```

We employ the pair of functions, `remember` and `estimate`, over time as samples become available and estimates are required. The extent of the environment associated with these functions is as long as one or the other of the two functions exist.

```
> (remember 1 1)
((1 . 1))
> (remember 2 3)
((1 . 1) (2 . 3))
> (remember 3 3)
((1 . 1) (2 . 3) (3 . 3))
> (estimate 2.4)
3
> (estimate 1.2)
1.1
```

2.13 Lambda and Functions as Arguments

Named functions defined with `defun` are not the only Lisp objects capable of referencing variables. Lisp also allows us to create unnamed functions called *lambda* functions. The expression `(function (lambda` *arguments body*`))` evaluates to a function defined with the formal parameters in *arguments* and the definition supplied in *body*. `Function`, like `quote`, does not evaluate its single argument; it expects either a `lambda` expression or a symbol defined as a function. Like `quote`, `function` has a handy abbreviation; Common Lisp allows you to write `#'`*expression* as an abbreviation for `(function` *expression*`)`. In Common Lisp, you cannot call such a function simply by having it appear as the first element of a list (as you can in the Scheme dialect), but, rather, you have to `funcall` or `apply` it.

`Funcall` takes as its first argument an expression that evaluates to a function and as many additional arguments as the function corresponding to the first argument has arguments of its own (*e.g.,* `(funcall #'cons 1 2)`). `Apply` is similar except that it takes only two arguments, the second being a list of the arguments to be supplied to the function corresponding the first argument (*e.g.,* `(apply #'cons (list 1 2))`). Here is a simple example illustrating `function` and `funcall`.

```
> (let ((x 0))
    (setq counter #'(lambda () (setq x (+ x 1)))))
#<Interpreted-Function (LAMBDA NIL (SETQ X (+ X 1))) 104BB76>
> (funcall counter)
1
> (funcall counter)
2
```

If we were to set the symbol `counter` to something new, then the environment created above would cease to exist terminating its extent.

Both named and lambda functions are often passed around as variables and in lists to be passed as arguments to `funcall`. Many Common Lisp functions take optional arguments corresponding to functions that are introduced with keywords. Keywords appear in argument lists as symbols whose first character is a colon (*e.g.,* `:test`). For instance, we mentioned that `member` uses `eq` to check for objects in a list; if instead of writing `(member` x l`)` we write `(member` x l `:test #'equal)`, then `member` will use `equal` as

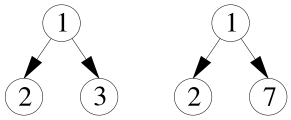


Figure 2.6: Labeled binary tree before and after label substitution

a test instead of eq.

2.14 Abstract Data Types

the implementation might change without their knowledge.

Suppose that we are writing programs that involve the manipulation of labeled binary trees. Rather than manipulating binary trees using primitives like cons, car, and cdr, we design a data abstraction using functions that we define and have mnemonic names. The following functions define an abstract data type with operations for creating, testing, accessing, and modifying instances of this data type.

```
(defun make-tree (label left right)
  (list 'labeled-binary-tree label left right))
(defun is-tree (x)
  (and (listp x) (eq (car x) 'labeled-binary-tree)))
(defun tree-label (tree) (cadr tree))
(defun tree-left (tree) (caddr tree))
(defun tree-right (tree) (cadddr tree))
(defun set-tree-label (tree value)
  (setf (cadr tree) value))
```

Having created this data abstraction, we can now forget about the details and program in terms of the abstraction. Here is a simple function that modifies the labels in a binary tree. Progn is a Lisp construct for collecting a sequence of expressions into a single form; eval evaluates each of the expressions in a progn form in turn, returning the value of the last one.

```
> (defun tree-sub (tree old new)
    (if (is-tree tree)
        (progn (if (eq old (tree-label tree))
                   (set-tree-label tree new))
               (tree-sub (tree-left tree) old new)
               (tree-sub (tree-right tree) old new))))
TREE-SUB
> (setq tree (make-tree 1 (make-tree 2 () ())
                          (make-tree 3 () ())))
(LABELED-BINARY-TREE 1 (LABELED-BINARY-TREE 2 NIL NIL)
                       (LABELED-BINARY-TREE 3 NIL NIL))
> (tree-sub tree 3 7)
7
> tree
(LABELED-BINARY-TREE 1 (LABELED-BINARY-TREE 2 NIL NIL)
                       (LABELED-BINARY-TREE 7 NIL NIL))
```

Figure **??** depicts the instances of the abstract data types created by the above invocations and the change that results from the label substitution.

This chapter provides only a glimpse of what Lisp has to offer in the way of procedural and data abstraction. Common Lisp and its extensions include a wide variety of techniques for structuring large programs and managing complex data types. Lisp provides a wonderful environment for experimenting with languages that support abstraction. To note just one area in which Lisp has had an impact, a great deal of research on object-oriented programming was carried out using Lisp. Today there are a number of packages that provide a basis for object-oriented programming within Common Lisp.

2.15 Mapcar, and other commands

Lambda functions are often created just to pass them to other functions. For instance, you might wish to compute the maximum of the corresponding entries in two lists.

```
> (mapcar #'(lambda (x y) (if (> x y) x y)) '(2 7 5) '(1 9 4))
(2 9 5)
```

Mapcan is like mapcar except that it appends the results using nconc. Mapc is like mapcar except that it does not do anything with the results; mapc is used only for its side effects.

The reduce function takes a function corresponding to a binary operation and a list and combines the elements of the list using the binary operation. Here are two examples illustrating reduce.

```
> (reduce #'+ `(1 2 3))
6
> (reduce #'(lambda (v w) (mapcar #'+ v w))
          '((1 0 0) (0 1 0) (0 0 1)))
(1 1 1)
```

The first example computes the sum of a list of numbers. The second example computes the sum of a list of vectors.

DO Loops Instead?

Here is another example of how a do loop might substitute for an expression involving a mapping function.

```
> (setq list '(1 2 3 4))
(1 2 3 4)
> (do ((args list (cdr args))
       (results nil (cons (oddp (car args)) results)))
      ((null args) (reverse results)))
(T NIL T NIL)
> (mapcar #'oddp list)
(T NIL T NIL)
```

In this particular case, the mapcar expression is more concise, but there are plenty of occasions in which the do loop will serve more appropriately than a mapping function.

2.16 Syntax

Other functions that operate on lists include: append which takes zero or more arguments that evaluate to lists and returns a new list which consists of all of the elements in those lists, and `nconc` which is similar to `append` except that it destructively modifies all of its arguments but the last.

```
> (setq x '(1 2) y '(3 4))
(3 4)
> (append x y)
(1 2 3 4)
> x
(1 2)
> (nconc x y)
(1 2 3 4)
> x
(1 2 3 4)
```

`Reverse` takes a list and returns a new list that consists of the elements of the old list arranged in reverse order. `Member` is a particularly useful Lisp function that takes two arguments corresponding to an arbitrary Lisp object and a list. If the object corresponding to the first argument is an element of the list corresponding to the second, then `member` returns that portion of the list (its *tail*) beginning with the first element of the list that is `eq` to that object. If the object is not an element of the list, `member` returns `nil`.

```
> (member 5 (append '(1 2 3) '(4 5) '(6 7 8)))
(5 6 7 8)
```

Using our new list-processing machinery, we now define a function `insert` that takes two arguments: a dotted pair of numbers (*e.g.,* `(1 . 2)`) and a list of dotted pairs of numbers (*e.g.,* `((1 . 2)(2 . 4)(4 . 8))`). We assume that the list of dotted pairs is sorted by the size of the first number in each pair with smaller numbers appearing earlier in the list. The list of dotted pairs represents samples in the form of a pair of numbers, x and $f(x)$, of a scalar function f of one variable. There is no need to include two pairs with the same first element; if the pairs all come from the same function and two pairs have the same (`=`) first element, then the second elements will also be the same and the dotted pairs will be `equal`. `Insert` creates a new sorted list that includes the pair corresponding to the first argument. It does so by pulling the list apart with car and cdr and putting it back together with cons, adding the new pair if necessary.

```
> (defun insert (new pairs)
    (cond ((null pairs) (cons new ()))
          ((= (car new) (car (car pairs))) pairs)
          ((< (car new) (car (car pairs))) (cons new pairs))
          (t (cons (car pairs) (insert new (cdr pairs))))))
INSERT
> (insert (cons 3 4) (list (cons 1 2) (cons 2 4) (cons 5 6)))
((1 . 2) (2 . 4) (3 . 4) (5 . 6))
> (insert (cons 1 2) (insert (cons 3 4) ()))
((1 . 2) (3 . 4))
```

Insert uses what is called cdr recursion as the function is applied recursively to the cdr of the list corresponding to its second argument until that list is nil. It is also often useful to recurse on both the car and the cdr of objects corresponding to complex nested list structures. The following function recurses on both the car and cdr of a list to test for the appearance of a given symbol.

```
> (defun search (symbol expression)
    (cond ((null expression) nil)
          ((symbolp expression) (eq expression symbol))
          (t (or (search symbol (car expression))
                 (search symbol (cdr expression))))))
SEARCH
> (search 'fred '(student (name fred) (year junior)))
T
```

In anticipation of an example introduced in the next section, we now define a function that given a number and a list of pairs of numbers returns nil if the number is greater than the greatest of the first elements of the pairs or less than the least of them. If there exists a pair whose car is = to the number, then the function returns a dotted pair consisting of the existing pair with the = car and nil (*e.g.*, ((1 . 2))). Otherwise, the function returns a dotted pair consisting of the two consecutive existing pairs such that the `car` of the first pair is greater than the number and the `car` of the second is less than the number (*e.g.,* `((1 .  2) 2 .  4))`.

```
> (defun nearest-pairs (x pairs)
    (cond ((null pairs) nil)
          ((= x (caar pairs)) (cons (car pairs) ()))
          ((or (< x (caar pairs)) (null (cdr pairs))) ())
          ((< x (caadr pairs)) (cons (car pairs) (cadr pairs)))
          (t (nearest-pairs x (cdr pairs)))))
NEAREST-PAIRS
> (setq pairs (list (cons 1 2) (cons 2 4) (cons 5 6)))
((1 . 2) (2 . 4) (5 . 6))
> (nearest-pairs 2 pairs)
((2 . 4))
> (nearest-pairs 3 pairs)
((2 . 4) 5 . 6)
```

Generating Output

We have already seen one function, `princ`, for generating formatted output from programs. Using `princ` and another function `terpri` which takes no arguments and results in a line feed, you can handle most of your output needs. There are less primitive alternatives available in Common Lisp, however, this will be discussed in a later section.

2.16.1 Format, for Special Output

`Format` is a complicated printing and formatting utility. The expression (`format` *destination string arguments*) is a common way of invoking `format` where *string* generally includes embedded directives that control tabbing, line feeds, and the printing of *arguments*. Directives are specified by the tilde character ( ) followed by one or more additional characters. The following examples illustrate just a few of the ways that `format` can be used.

```
> (format nil "~D is an integer; ~A is a symbol" 17 'foo)
"17 is an integer; FOO is a symbol"
> (format nil "~4,2F is a real number" 1.23456)
"1.23 is a real number"
> (let ((x 6) (y 1.2))
    (format nil "~D times ~4,2F is ~4,2F" x y (* x y)))
"6 times 1.20 is 7.20"
> (format nil "Here~%is a line break.")
"Here
is a line break."
```

The directive `4,2F` is used to print a fixed-format, floating-point number with a minimum display width of four characters including the decimal point and two digits after the decimal point. The directive `A` is used to print an arbitrary object just as it would be printed by `princ`. If the destination is `nil`, then `format` returns the formatted string, otherwise `format` returns `nil` and sends the string to the specified destination. If the destination is `t`, then `format` prints to the *standard output* which is generally the terminal or display device that the Lisp interpreter is using.

Common Lisp provides a variety of other printing routines for handling errors and interacting with the interpreter. We mention one that may prove useful in dealing with the Lisp interpreter. By executing `(setq *print-pretty* t)`, deeply nested list structures will appear much more readable.

2.17 Debugging Programs

For the most part, debugging a Lisp program is no different from debugging any other program. If your code fails to behave as expected, then start by checking for the most common errors, such as misspelled function names, unbalanced parentheses, wrong number of arguments to a function, inappropriate arguments, or using a variable before it is set.

Most modern implementations of Lisp provide elaborate debugging tools, but discussion of these tools is outside the scope of this text. In the following, we consider some simple tools and techniques that are available in all implementations of Common Lisp and that should suffice for the exercises in this text.

To print out information while your code is running, you can use either `princ` or `format`. One of the simplest debugging aids is to liberally add print statements throughout your code to track changes in the value of variables.

Using the Common Lisp `trace` utility, you can keep track of when and with what arguments certain functions are called. The following script illustrates how to trace two of the functions defined earlier in this chapter.

```
> (trace raise square)
(RAISE SQUARE)
> (square (raise 2 2))
1 Enter RAISE 2 2
| 2 Enter RAISE 2 1
|   3 Enter RAISE 2 0
|   3 Exit RAISE 1
| 2 Exit RAISE 2
1 Exit RAISE 4
1 Enter SQUARE 4
1 Exit SQUARE 16
16
```

We can also turn off tracing on one or more functions using `untrace`.

```
> (untrace raise)
(RAISE)
> (square (raise 2 2))
1 Enter SQUARE 4
1 Exit SQUARE 16
16
```

Sometimes tracing does not provide enough information or provides more than we want. In such cases, it is often useful to single-step though the evaluation of a program. The `step` utility allows just this. The details of `step` just as the details of `trace` will depend upon the particular implementation of Common Lisp, but the following script will give you some idea of what is available.

```
> (setq n 3)
3
> (step (square n))
  (SQUARE N) -> :h
    :n    Evaluate current expression in step mode.
    :s    Evaluate current expression without stepping.
    :x    Finish evaluation, but turn Stepper off.
    :p    Print current expression.
    :b    Enter the Debugger.
    :q    Exit to Top Level.
    :h    Print this text.
  (SQUARE N) -> :n
    (FUNCTION SQUARE) -> :n
    #<Interpreted-Function (NAMED-LAMBDA SQUARE (X)
                            (BLOCK SQUARE (* X X))) 100DCD6>
    N = 3
    (BLOCK SQUARE (* X X)) -> :n
      (* X X) -> :n
        (FUNCTION *) -> :n
        #<Compiled-Function * 4ABA76>
        X = 3
        X = 3
```

In the above script, `:h` and `:n` were typed by the user in response to the `->` prompt.

It is nearly impossible to write a program that evaluates without error. Whenever an error occurs, you will enter the debugger, some confusing information will be displayed, and you will see an alternate prompt. Usually typing something like `?` or `help` will give you a list of options that will enable you to continue from the error or give you some idea of what caused the error.

Errors and debugging are inevitable in programming. Take some time to familiarize yourself with the debugging tools available with your implementation of Common Lisp.

2.18 Return to the original Example

We use the following data abstraction to represent possible reports on the values of sensor and control parameters.

```
(defun param (tuple) (first tuple))
(defun value (tuple) (second tuple))
```

For example, `(forward new)` corresponds to a report that the `forward` sensor has a value of `near`.

The same sensor will report different values at different times. To keep track of the time of a report, we associate with each report an integer referred to as a *time stamp*. The following data abstraction is used for time-stamped sensor and parameter reports.

```
(defun tuple (item) (first item))
(defun stamp (item) (second item))
```

For example, `((forward far) 1012)` indicates that the `forward` sensor reported `far` at time `1012`.

Our control algorithm maintains a list of the most recent sensor and control parameter reports. The first routine, `update`, that we consider takes a list of new reports and a list of old reports and combines them. The old list includes reports from all of the sensors and control parameters but the new list may not. One approach to combining the reports is to consider each of the old reports and check to see if it is updated in the new reports.

The Common Lisp `assoc` function takes two arguments corresponding to an expression and a list of lists with two elements each. The second argument is called an *association list* and allows us to map between symbolic expressions (*e.g.,* `((type truck) (color red) (year 1950))`). In its simplest uses, `assoc` takes a symbol and a list of pairs of symbols and returns the first pair whose `car` is `eq` to the first argument or `nil` if no such pair exists. For instance, `(assoc 'left '((left far) (right near)))` would return `(left far)`.

`Assoc` allows optional arguments specified by keywords. Note that a list of time-stamped reports will contain pairs of the form `((left far) 1)`.

```
> (assoc '(left far) '(((left far) 1)))
NIL
> (assoc '(left far) '(((left far) 1)) :test #'equal)
((left far) 1)
```

In searching for an updated report, we are looking for a new report on the same parameter with a possibly different value.

```
> (assoc '(left far) '(((left near) 1)) :test #'equal)
NIL
> (assoc '(left far) '(((left far) 1))
         :test #'(lambda (x y) (eq (car x) (car y))))
((left far) 1)
```

Using another keyword argument, we can index into the structure of the first element of the pairs in the association list to achieve the same result; `(assoc 'left '(((left far) 1) :key #'car)` returns `((left far) 1)`. Instead of testing the first argument of `assoc` against the `car` of each pair in the association list, we test the first argument against the result of applying the function introduced by `:key` to the `car` of each pair in the association list.

We are making progress, but we have a slight problem. In our discussion of `assoc`, we assume that tuples are implemented as a list of two elements and that reports are implemented as a list of a tuple and an integer. This happens to be true but an important advantage of introducing a data abstraction is that we should not have to think about how it is implemented. By employing `assoc` as we did above, we violate the data abstraction by taking advantage of a particular implementation. If someone were to change the implementation (*e.g.,* used arrays instead of lists or put the time stamp first instead of last), then we would have to track down all of the places where `assoc` was used and make changes in accord with the new implementation.

To maintain the abstraction, we introduce two functions that perform associations on lists of reports. The first function searches for a report with a tuple that has a particular parameter; the second function searches for a report with a particular tuple.

```
(defun param-assoc (param reports)
  (assoc param reports :key #'param))
(defun tuple-assoc (tuple reports)
  (assoc tuple reports :test #'equal))
```

Note that both implementations violate the data abstraction for reports by assuming that the `car` of a report is a tuple. By using `param` instead of `car` in the first function, we do, however, respect the data abstraction for tuples. Because the above implementations violate the abstraction for reports, they should be located with the implementation for reports and documented appropriately so that a programmer changing the implementation for reports will also change the implementation for the two association functions. With a little more work, we could respect the abstraction for both reports and tuples but in this case careful documentation and a little discipline should suffice to support easy maintenance of the code.

Now we can specify the algorithm for `update` as follows. We call `update` with two lists of time-stamped reports. For each time-stamped report in the list of old reports, check to see if there is a new report with the same parameter; if so, add the new report to the list of reports to be returned by `update` and otherwise add the old report to the list of reports to be returned. This algorithm is implemented in Lisp as follows.

```
(defun update (new old)
  (mapcar #'(lambda (item)
              (or (param-assoc (param (tuple item)) new)
                  item))
          old))
```

The form `(or` *first second*`)` is an alternative method of computing `(let ((result` *first*`)) (if (null result)` *second* `result))`.

Consider another possible implementation of `update`. Suppose in this case that neither the old nor the new reports necessarily include reports on all of the sensor and control parameters and that the new reports may even include several reports on the same parameter at different times. In the following implementation, we use the `dolist` iterative construct and a subroutine `fuse` that takes a single report and combines it with the old reports. As in the previous algorithm we call `update` with two lists of time-stamped reports. In this case, however, we step through the new reports one at a time updating the list of old reports using the `fuse` subroutine.

```
(defun update (new old)
  (dolist (item new old) (setq old (fuse item old))))
```

`Fuse` takes a report and a list of reports and substitutes its first argument for the first report in the list reporting on the same parameter if that report has an earlier time stamp than the first argument or adds the first argument to the end of the list of reports if there is no such report. `Fuse` uses `cdr` recursion to take the old list of reports apart with `car` and `cdr` and put it back together with `cons`. It either substitutes the new report for an old one with the same parameter or if no such old report exists tacks the new report on the end of the old reports.

```
(defun fuse (report old)
  (cond ((null old) (list report))
        ((eq (param (tuple report)) (param (tuple (car old))))
         (if (> (stamp report) (stamp (car old)))
           (cons report (cdr old)) old))
        (t (cons (car old) (fuse report (cdr old))))))
```

The following provides an example showing how `update` works.

```
> (update '(((forward near) 2) ((left near) 2) ((left far) 1))
          '(((forward far) 0) ((left away) 0) ((right near) 1)
            ((rear far) 0) ((speed slow) 1)))
(((LEFT NEAR) 2) ((FORWARD NEAR) 2) ((RIGHT NEAR) 1)
 ((REAR FAR) 0) ((SPEED SLOW) 1))
```

Here is an alternative implementation of `fuse` using `param-assoc` and a function `remove` provided in Common Lisp. `Remove` takes an expression and a list; the result is a new list that has the same elements as the original except those `eq` to the expression (*e.g.,* `(remove 1 '(1 2 3 1))` returns `(2 3)`). `Remove` takes optional keyword arguments (*e.g.,* `(remove '(1 2) '(a (1 2) b c) :test #'equal)` returns `(a b c)`).

```
(defun fuse (report old)
  (let ((a (param-assoc (param (tuple report)) old)))
    (cond ((null a) (cons report old))
          ((< (stamp report) (stamp a)) old)
          (t (cons a (remove a old))))))
```

We could have used `(substitute report a old)` instead of `(cons report (remove a old))`. `Substitute` substitutes its first argument for all occurrences of its second argument appearing as elements of the list corresponding to its third argument. `Subst` makes substitutions in nested list structures (*e.g.,* `(substitute 1 0 '(0 1 (1 0)))` returns `(1 1 (1 0))` whereas `(subst 1 0 '(0 1 (1 0)))` returns `(1 1 (1 1))`).

It should be noted that most of the Common Lisp functions that we describe in this text are more versatile than indicated. For instance, many of the functions mentioned take more keyword arguments than those described. In addition, many of the functions that apply to lists also apply to a more general *sequence* data type that encompasses lists and one-dimensional arrays.

Now that we can update our sensor reports, we need to determine what control actions to take. The control strategy is encoded in a set of rules. We use the following data abstraction for rules.

```
(defun conditions (rule) (car rule))
(defun action (rule) (cadr rule))
```

The conditions correspond to sensor/value pairs and actions correspond to control-parameter/value pairs. We say that a rule is *applicable* given a set of reports if each of the conditions have associated `equal` reports. The following functions implement a predicate to determine whether or not a rule is applicable.

```
(defun applicablep (rule reports)
  (aux-applicablep (conditions rule) reports))
(defun aux-applicablep (tuples reports)
  (or (null tuples)
      (and (tuple-assoc (car tuples) reports)
           (aux-applicablep (cdr tuples) reports))))
```

`Applicablep` takes a rule and a list of time-stamped reports and uses `cdr` recursion to check if each condition in the rule's list of conditions is corresponds to some report in the list of reports. The auxiliary function is used to set up the recursion by introducing the variable `conditions` that we wish to recurse upon. Such auxiliary functions are common in implementing recursive procedures.

We can also implement `applicablep` using a special mapping function that behaves like a boolean function. The expression `(every` *test arguments*`)` returns `t` if *test* returns non-`nil` when applied to each cross section of *arguments* (*e.g.,* `(every #'oddp '(3 5 9))` and `(every #'eq '(a 1) '(a 1))` both return `t`).

```
(defun applicablep (rule reports)
  (every #'(lambda (tuple) (tuple-assoc tuple reports))
         (conditions rule)))
```

The following function tests each rule in a set of rules of rules and acts according to the applicable rules.

```
(defun react (rules reports)
  (dolist (rule rules reports)
    (if (applicablep rule reports)
      (setq reports (fuse (act (action rule)) reports)))))
```

Acting in our simple implementation just consists of adding a time stamp to the action of a rule. The Common Lisp function `get-internal-real-time` returns an integer representing the current time.

```
(defun act (action) (list action (get-internal-real-time)))
```

Here is a simple example showing the result of reacting.

```
> (react '((((forward near) (jleft far)) (turn left)))
         '(((forward near) 0) ((turn right) 1) ((jleft far) 1)))
(((TURN LEFT) 2214639) ((FORWARD NEAR) 0) ((JLEFT FAR) 1))
```

Note that, if there are two applicable rules with conflicting actions, the above function will apply `act` to each of the actions. In a more realistic implementation, we would provide some means of resolving conflicts involving applicable rules.

We provide an alternative implementation of `react` that allows us to demonstrate another useful mapping function. `Mapcan` behaves like `mapcar` except that it combines its results using `nconc` (*e.g.,* `(mapcan #'cdr '((0) (1 a) (2) (3 b) (4 c))` returns `(a b c)` the same as `(apply #'nconc (mapcar #'cdr '((0) (1 a) (2) (3 b) (4 c))))`).

```
(defun react (rules reports)
  (update (mapcan #'(lambda (rule)
                      (and (applicablep rule reports)
                           (list (act (action rule)))))
                  rules)
          reports))
```

The function `run` applies `react` and `update` in a cycle using the lisp iterative construct `dotimes`. Assume that `collect` returns the latest sensor reports.

```
(defun run (rules reports)
  (dotimes (index 100 reports)
    (setq reports (react rules (update (collect))))))
```

The above function performs 100 cycles with `index` set to $0, 1, 2, \ldots, 99$. Alternatively, we might implement `run` using two mutually recursive functions.

```
(defun run (rules reports)
  (aux-update rules (collect) reports 0))
(defun aux-update (rules new old i)
  (if (< i 100) (aux-react rules (update new old) i)))
(defun aux-react (rules reports i)
  (aux-update rules (collect) (react rules reports) (+ i 1)))
```

As illustrated above, Common Lisp allows for a lot of variety in implementing algorithms. In the following chapters, we often choose the simplest or most concise implementation rather than the most efficient.

2.19 Complexity and Expressivity

We are interested in developing 'efficient' algorithms for symbolic reasoning. Usually, we are satisfied if the procedures we write perform a total number of steps some low-order polynomial function of the size of the inputs. In this text, we assume that you have at least heard about asymptotic complexity and *big-O* notation. For instance, you might know that you can compute the minimum of a list of n integers with $O(n)$ comparisons, or sort the list using $O(n \log n)$ comparisons. Problems such as sorting, shortest path, minimum-cost spanning tree are all said to be easy because they can be computed with a small (polynomial in the size of the problem description) number of basic steps.

There are plenty of problems that are not easy. The best known exact solutions to the general traveling salesperson problem require an exponential number of steps. The yes-or-no version of this problem[5] is said to be in the

[5]The classical formulation of the traveling salesperson problem as a yes-or-no decision problem is as follows. Given a positive integer K and a complete graph whose vertices represent cities and whose edges represent routes between cities labeled with the distance between adjacent cities, determine if there exists a tour of length at most K corresponding

class of *NP-complete* problems [Garey & Johnson, 1979].

In the following chapters, we will be faced with other problems in the class of NP-complete problems. NP-completeness does not mean that we should despair. Sometimes we will be able to find reasonable approximations (*e.g.,* find a tour that is within a small constant factor of the minimum tour) that suffice for our purposes. For other problems, it might acceptable to find a good solution with high probability (*e.g.,* 'most of the time' we find a tour that is 'not too long.'). Algorithm complexity will figure prominently in our discussion of representation issues.

Expressiveness concerns what you can or cannot represent in a given representation without regard to computation. Lisp as a programming language is as expressive as you could wish for given that it is powerful enough to encode a universal Turing machine. Along with increased expressivity, however, comes other possible drawbacks. It is impossible to determine for an arbitrary Lisp program whether it will terminate or not. There are times when it will seem reasonable to sacrifice expressivity in order to ensure that our algorithms do not run indefinitely or take an inordinate amount of time to return a result. Tradeoffs of this sort involving expressivity and complexity will surface time and again in the following chapters.

If you are interested in learning more about Common Lisp, you should consult Steele's description of the language [Steele, 1984]. If you are interested in learning more about how to program in Lisp, you might consider a text specific to Common Lisp (*e.g.,* [Wilensky, 1986]) or one of the many general introductions to Lisp (*e.g.,* [Touretsky, 1984]). For an excellent introduction to programming in Lisp using a dialect of Lisp called *Scheme* consider the text by Abelson and Sussman [Abelson & Sussman, 1986].

---

to a sequence of vertices and edges that visits each city. Given that the yes-or-no problem is hard, the corresponding optimization problem (*i.e.,* find the minimum length tour) is hard.

2.20 Exercises

1. Write your own versions of the Lisp functions, `member`, `append`, `nconc`, `reverse`, and `nth`. Rename them so that you do not redefine functions that are critical to the proper functioning of your Lisp environment. For `append` and `nconc`, it is enough to implement versions for two arguments.

2. Create a set of recursive functions for manipulating sets. You should write functions for taking the union, intersection, and complement of sets and testing whether an item is an element of a set. `Union` and `intersection` are already defined in Common Lisp so you should use alternative names for the same reason mentioned in the previous exercise.

3. You are given a *dictionary* in the form of a list of 'exploded' symbols (*e.g.,* `(d e f u n)`, `(s e t q)`, `(s e t f)`). Write a recursive function, `lookup`, that takes a prefix of an 'exploded' symbol (*e.g.,* `(s e t)`) and a dictionary and returns the list of all items in the dictionary that match this prefix.

4. Write a recursive function, `distance`, to compute the *distance* between two bit vectors of the same length represented as lists of ones an zeros (*e.g.,* `(1 0 1 1)`, `(0 1 0 1)`) using as a metric the number of bits that are different (*e.g.,* three in the case of `(1 0 1 1)` and `(0 1 0 1)`). Write a recursive function, `closest`, that finds two vectors that are closest in distance in a list of bit vectors. Discuss what would have to be done if the vectors were of different length and composed of upper case letters instead of ones and zeros. In particular, discuss what distance metric you would use if you were trying to implement a user-friendly spell checker that offers suggestions as to what the user

might have been trying to spell.

5. In the following exercise, you are to implement a rule of logical inference. It happens to be a generalization of modus ponens which you will hear more about in Chapter 3 but you do not need to know the details for this exercise. Suppose that your logician employer has asked you to implement the following specification. Write a recursive function `infer` that takes a conjunction represented as a list of the form (Q_1 Q_2 ... Q_m) and a rule of the form (Q (P_1 P_2 ... P_n)) and returns `nil` if Q does not appear in the list and otherwise returns a new list in which Q is replaced by P_1 P_2 ... P_n. Assume that a symbol appears at most once in a conjunction. Here are some examples illustrating the expected behavior of `infer`.

```
> (infer '(D E A F) '(A (B C)))
(D E B C F)
> (infer '(D E A F) '(A ()))
(D E F)
> (infer '(D) '(A (B C)))
NIL
```

6. Rewrite `interpolate`, `nearest-pairs`, `remember` and `estimate` using a data abstraction for pairs of the form (x . $f(x)$).

7. Write a recursive function that constructs a labeled binary tree of depth n. Label the root and its subtrees with the integers 0 through 2^n so that no two subtrees have the same label and that, given two subtrees of different depth, the label of the one with the greater depth is larger than the other.

8. Write a pair of recursive functions, `lookup` and `enter`, that refer to the same local state variable corresponding to an appropriate data structure for the dictionary in Exercise ??. `Lookup` has the same basic

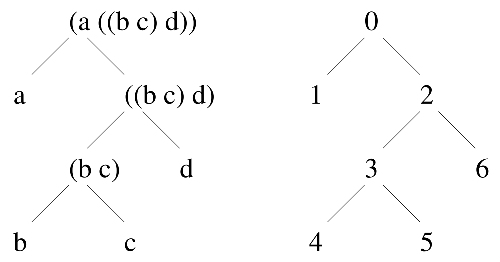


Figure 2.7: A binary tree and its depth-first numbering

input/output behavior as the function of Exercise **??** except that it does not require the second argument. `Enter` takes a single argument corresponding to a new function name and enters it into the dictionary. Use a tree data structure for the dictionary in which function names are stored at the leaves and symbols corresponding to the letters in the function names are stored at internal nodes. Discuss why this tree representation is better than representing the dictionary as a flat list of function names.

9. In Chapter 4, we consider an optimization technique that involves simulating evolution. In this technique, Lisp objects are combined and mutated as abstract forms of genetic material. In this exercise, we consider a method for indexing into nested list structures that might be used for selecting subexpressions for mutation or for combining genetic material during reproduction.

   Write a recursive Lisp function `index` that takes two arguments: a nested list structure of a particular restricted form and an integer. Any nonnumeric symbol is of the restricted form and any list structure of the form (x y) is of the restricted form if x and y are of the restricted form. So `a`, `(a b)`, and `(a (b c))` are all of the restricted form, but `1` and `(a b c)` are not. Note that every restricted form can be represented as a binary tree. Figure ?? shows the binary tree for the restricted form `(a ((b c) d))` along with the *depth-first numbering* of the nodes in the binary tree. Given a restricted form and an integer, `index` returns the subtree with the corresponding depth-first numbering. Here are some examples demonstrating `index`.

```
> (index '(a ((b c) d)) 6)
D
> (index '(a ((b c) d)) 3)
(B C)
> (index '(a ((b c) d)) 0)
(A ((B C) D))
```

Note that the fact that every list has exactly two elements simplifies the recursion considerably. You can assume that the second argument to `index` corresponds to the depth-first ordering number of a subtree of the first argument.

10. Discuss the problems involved in defining `defun` in terms of `lambda` and `mapcar` in terms of `apply`.

11. Consider the following variant forms of rules for our rule-based control system.

    (a) Suppose that the conditions of rules are specified as boolean combinations of sensor reports (*e.g.,* `(and (forward near) (or (jleft near) (jright near)))`).

    (b) Suppose that parameters have numerical values and we allow inequalities in the conditions of rules (*e.g.,* `((forward < 2) (jleft > 1) (speed > 0))`).

    In each case, sketch how you would implement `applicablep`.

12. To access the parameter associated with a time-stamped report, we would evaluate `(param (tuple datum))`. We could define a function

to directly access the parameter associated with a given report, but we would have to name it something other than `param`. In object-oriented languages, objects map *messages* to procedures called *methods*. Two different types of objects can map the same message to different methods. Lisp provides a variety of object-oriented extensions that would allow this sort of flexibility. In this exercise, we sketch a simple object-oriented extension to Lisp. The extension is based on the following convention for sending messages to objects, `(send` *object message*`)`. In the following implementation, an object is just a list of message/method pairs, where methods are implemented as closures. As a simple example, consider how to implement sensor and control parameter reports as objects.

```
(defun make-tuple (p v)
  (let ((param p) (value v))
    (list (list 'param #'(lambda () param))
          (list 'value #'(lambda () value)))))
```

Note the use of the `let` statement to maintain local state. Now we implement time-stamped reports, providing a method for directly accessing the parameter associated with a report.

```
(defun make-datum (x y)
  (let ((tuple x) (stamp y))
    (list (list 'tuple #'(lambda () tuple))
          (list 'stamp #'(lambda () stamp))
          (list 'param #'(lambda () (send tuple 'param))))))
```

`Send` takes an object and a message, looks up the method associated with the message and uses `funcall` to invoke the method.

```
(defun send (object message)
  (let ((pair (assoc message object)))
    (if (null pair) (princ "Undefined!")
        (funcall (cadr pair)))))
```

Here we show how we can use the same message to refer to different types of objects.

```
> (progn (setq x (make-tuple 'forward 'near)
               y (make-datum x 0))
         (eq (send x 'param) (send y 'param)))
T
```

Implement binary trees as objects. You will have to extend the implementation above to handle methods that take arguments. One approach is to give `send` a third argument corresponding to a list of method arguments and then use `apply` instead of `funcall` to invoke the method on the list of arguments.