LISP

PATRICK HENRY WINSTON
BERTHOLD KLAUS PAUL HORN

CONTENTS

PART I

1 UNDERSTANDING SYMBOL MANIPULATION

This book has two parts, each written to accomplish a particular purpose:

- The purpose of Part One of this book is to introduce the ideas of symbol manipulation and to teach the basics of LISP programming.
- The purpose of Part Two is to demonstrate LISP's muscle and to excite people about what LISP can do.

This brief chapter defines symbol manipulation, explains why LISP is the right symbol-manipulation language to learn, and previews the ideas that will be covered.

Symbol Manipulation is Like Working with Words and Sentences

What exactly is symbol manipulation? Everyone understands that computers can do arithmetic like crazy, but how can they do more? How, for example, can a computer do things that are intelligent or that seem to be? In part, the answer lies in the fact that the numbers stored in a computer can be thought of as a code for other things. It is true that from one perspective everything in a computer is a string of binary digits, ones and zeros, that everyone calls bits. Commonly, these binary digits are interpreted as a code for decimal digits. But from another perspective, groups of those same bits can be interpreted as characters, just as sequences of dots and dashes are interpreted as characters by people who have learned the Morse code.

Then, once it is understood that groups of bits can represent characters, it is

clear that groups of characters can represent words. And groups of words can represent sentences, and groups of sentences can represent paragraphs. By more and more grouping, eventually larger and larger structures are possible. Groups of bits eventually become sections, chapters, books, and encyclopedias.

In LISP similar interpretation and grouping notions are at work, although the units go by different names.

- In LISP, the fundamental things formed from bits are word-like objects called *atoms.*

- Groups of atoms form *lists.* Lists themselves can be grouped together to form higher-level lists. Indeed, the ability to form hierarchical groups is of fundamental importance.

- Atoms and lists collectively are called *symbolic expressions.* Working with them is what symbol manipulation using LISP is about. Indeed, symbol manipulation is sometimes called list processing.

A symbol-manipulation program uses symbolic expressions to remember and work with data and procedures, just as people use pencil, paper, and human language to remember and work with data and procedures. A symbol-manipulation program typically has sections that recognize particular symbolic expressions, tear old ones apart, and assemble new ones.

Here are two examples of symbolic expressions. The parentheses mark where lists begin and end. The first is a description of a structure built out of children's blocks. The second is a description of a certain university.

```
(ARCH (PARTS LINTEL POST1 POST2)
      (LINTEL MUST-BE-SUPPORTED-BY POST1)
      (LINTEL MUST-BE-SUPPORTED-BY POST2)
      (LINTEL A-KIND-OF WEDGE)
      (POST1 A-KIND-OF BRICK)
      (POST2 A-KIND-OF BRICK)
      (POST1 MUST-NOT-TOUCH POST2)
      (POST2 MUST-NOT-TOUCH POST1))

(MIT (A-KIND-OF UNIVERSITY)
     (LOCATION (CAMBRIDGE MASSACHUSETTS))
     (PHONE 253-1000)
     (SCHOOLS (ARCHITECTURE
                BUSINESS
                ENGINEERING
                HUMANITIES
                SCIENCE))
     (FOUNDER (WILLIAM BARTON ROGERS)))
```

Certainly these are not very scary. Both just describe something according to some conventions about how to arrange symbols. Here is another example, this time expressing a rule for determining whether some animal is a carnivore:

```
(RULE IDENTIFY6
      (IF (ANIMAL HAS POINTED TEETH)
          (ANIMAL HAS CLAWS)
          (ANIMAL HAS FORWARD EYES))
      (THEN (ANIMAL IS CARNIVORE)))
```

What we see is just another way of expressing the idea that an animal with pointed teeth, claws, and forward-pointing eyes is probably a carnivore. Using such a rule amounts to taking it apart, finding the conditions specified following the IF, checking to see if those pieces are on a list of believed assertions, and adding the conclusion following the THEN to a list of believed assertions. Using such a rule is an example of symbol manipulation.

Symbol Manipulation is Needed to Make Computers Intelligent

These days, there is a growing armamentarium of programs that exhibit what most people consider intelligent behavior. Nearly all of these intelligent or seemingly intelligent programs are written in LISP. Many have the potential of great practical importance. Here are some examples:

- Expert problem solvers. One of the first LISP programs did calculus problems at the level of university freshmen. Another early program did geometric analogy problems of the sort found in intelligence tests. Since then, newer programs have diagnosed infections of the blood, understood electronic circuits, evaluated geological evidence for mineral prospecting, and invented interesting mathematics. All are written in LISP.

- Common-sense reasoning. Much of human thinking seems to involve a small amount of reasoning using a large amount of knowledge. Representing knowledge means choosing a vocabulary of symbols and fixing some conventions for arranging them. Good representations make just the right things explicit. LISP is the language in which most research on representation is done.

- Learning. Not much work has been done on the learning of concepts by computer, but certainly most of what has been done also rests on progress in representation. LISP again dominates.

- Natural language interfaces. There is a growing need for programs that interact with people in English and other natural languages. Full understanding of natural languages by computer is probably a long time off, but practical systems have been built for asking questions about constrained domains ranging from moon rocks to the ships at sea to the inventory of a carpet company.

- Education and intelligent support systems. To interact comfortably with computers, people must have computers that know what people know and how to tell them more. No one wants a long-winded explanation after they know a lot. Nor does anyone want a telegram-like explanation when just beginning. LISP-based programs are beginning to make user models by analyzing what the user does. These programs use the models to trim or elaborate explanations.

- Speech and vision. Understanding how people hear and see has proved fantastically difficult. It seems that we do not know enough about how the physical world constrains what ends up on our ear drums and retinas. Nevertheless, progress is being made and much of it is made in LISP, even though a great deal of straight arithmetic-oriented programming is necessary. To be sure, LISP has no special advantages for arithmetic-oriented programming. But at the same time, LISP has no completely debilitating disadvantages for arithmetic either. This is surprising to some people because LISP was once molasses slow at such work.

Consequently, a person who wants to know about computer intelligence at some point needs to understand LISP if his understanding is to be complete. And there are certainly other applications, some of which are again surprising. At least the following deserve mention:

- Word processing. EMACS is a powerful text editor. The version that runs on the MULTICS operating system was written entirely in LISP. So was the editor called ZWEI, used on the LISP machine, a modern personal computer. Similarly, a number of text justifiers, programs that arrange text on pages, have been written in LISP.

- Symbolic mathematics. The MACSYMA system is a giant set of programs for applied mathematicians that enables them to handle algebra that would defy pencil and paper efforts.

- Systems programming. The LISP machine is a modern personal computer programmed from top to bottom in LISP. The operating system, the user utility programs, the compilers, and the interpreters are all written in LISP with a saving in cost of one or two orders of magnitude.

Given all these examples, it is no surprise that the following is accepted by nearly everyone:

- Symbol manipulation is an essential tool. Computer scientists and engineers must know about it.

LISP is the Right Symbol-Manipulation Language to Learn

There are too many programming languages. Fortunately, however, only a few are for symbol manipulation, and of these LISP is the most used. After LISP is understood, most of the other symbol-manipulation languages are easy to learn.

Why has LISP become the most used language for symbol manipulation, and lately, much more? There is some disagreement. All of the following arguments have adherents:

- The interaction argument. LISP is oriented toward programming at a terminal with rapid response. All programs and all data can be displayed or altered at will.

- The environment argument. LISP has been used by a community that needed and got the best in editing and debugging tools. For over two decades, people at the world's largest artificial intelligence centers have created sophisticated computing environments around LISP, making a combination that is unbeatable for writing big, intelligent programs.

- The features argument. LISP was designed for symbol manipulation from the start. In fact, LISP is an acronym for list processing language. Consequently, LISP has just the right features.

- The uniformity argument. LISP functions and LISP data have the same form. One LISP function can analyze another. One LISP function even can put together another and use it.

Happily, LISP is an easy language to learn. A few hours of study is enough to understand some amazing programs. Previous exposure to some other more common language is not necessary. Indeed such experience can be something of a handicap, for there can be a serious danger of developing a bad accent. Other languages do things differently and function-by-function translation leads to awkward constructions.

One reason LISP is easy to learn is that its syntax is extremely simple. Curiously, this came about by accident. John McCarthy, LISP's inventor, originally used a sort of old LISP that is about as hard to read as old English. At one point, however, he wished to use LISP in a context that required both the functions and the data to have the same syntactic form. The resulting form of LISP, which McCarthy intended to use only for developing a piece of mathematics, caught on and quickly became the standard.

2
BASIC LISP FUNCTIONS

The purpose of this chapter is to introduce LISP's basic symbol-manipulation functions. To do this, some functions are introduced that work on numbers and others are introduced that work on lists. The list-oriented functions extract parts of lists and build new lists.

LISP Means Symbol Manipulation

As with other computer languages, the best way to learn LISP is bravely, jumping right into interesting programs. We will therefore look occasionally at things that will not be completely understood with a view toward moving as quickly as possible into exciting applications.

To get started, imagine being seated in front of a computer terminal. You begin as if engaging in a conversation, with LISP tossing back results in response to typed input. Suppose, for example, that you would like some help adding numbers. The proper incantation would be this:

```
(PLUS 3.14 2.71)
```

LISP would agreeably respond:

```
5.85
```

This is a simple example of LISP's ability to handle arithmetic. Elementary examples of symbol manipulation are equally straightforward. Suppose, for example, that we are interested in keeping track of some facts needed in connection with understanding a children's story about, say, a robot. It might well

be important to remember that certain children are friends of the robot. Typically a name is required to denote such a group, and the name FRIENDS will do as well as any other. If Dick and Jane and Sally are friends of the robot, this fact could be remembered by typing this line:

```
(SET 'FRIENDS '(DICK JANE SALLY))
```

SET associates (DICK JANE SALLY) with FRIENDS. Do not worry about the quote marks that appear. They will be explained later.

Typing FRIENDS now causes the list of friends to be typed in response:

```
FRIENDS
  (DICK JANE SALLY)
```

There could be a similar list established for enemies:

```
(SET 'ENEMIES '(TROLL GRINCH GHOST))
```

Because friends and enemies tend to be dynamic categories in children's worlds, it is often necessary to change the category a particular individual is in. The ghost ceases to be an enemy and becomes a friend after typing two lines like this:

```
(SET 'ENEMIES (DELETE 'GHOST ENEMIES))

(SET 'FRIENDS (CONS 'GHOST FRIENDS))
```

The first line changes the remembered list of enemies to what it was minus the entry GHOST. The second line would be simpler if CONS were replaced by something more mnemonic like ADD, but we are stuck with historical convention. In any event, FRIENDS and ENEMIES have been changed such that we now get properly altered responses:

```
ENEMIES
  (TROLL GRINCH)

FRIENDS
  (GHOST DICK JANE SALLY)
```

Later we will see how to write a program that does the same job. In particular we will understand how the following creates a program named NEWFRIEND for changing a person from an enemy into a friend:

```
(DEFUN NEWFRIEND (NAME)
       (SET 'ENEMIES (DELETE NAME ENEMIES))
       (SET 'FRIENDS (CONS NAME FRIENDS)))
```

With NEWFRIEND, the previous elevation of the status of GHOST can be achieved more simply by typing only this:

```
(NEWFRIEND 'GHOST)
```

LISP Programs and Data Are Constructed out of S-expressions

Some important points should be noted. First, when left and right parentheses surround something, we call the result a list and speak of its elements. In our very first example, the list (PLUS 3.14 2.71) has three elements, PLUS, 3.14, and 2.71.

Note the peculiar location of the function PLUS, standing strangely before the two things to be added rather than between them as in ordinary arithmetic notation. In LISP the *function* to be performed is always given first, followed then by the things that the function is to work with, the *arguments.* Thus 3.14 and 2.71 are the arguments given to the function PLUS.

This so-called prefix notation facilitates uniformity and because the function name is always in the same place no matter how many arguments are involved.

Let us look at more examples. We will stick to arithmetic for the moment since arithmetic is initially more familiar to most people than symbol manipulation is. LISP's responses are indented by two spaces in order to distinguish them from the user's typed input:

```
(DIFFERENCE 3.14 2.71)
  0.43

(TIMES 9 3)
  27

(QUOTIENT 9 3)
  3

(MAX 2 4 3)
  4

(MIN 2 4 3)
  2

(ADD1 6)
  7

(SUB1 6)
  5
```

```
(SQRT 4.0)
  2.0

(EXPT 2 3)
  8

(MINUS 8)
  -8

(MINUS -8)
  8

(ABS 5)
  5

(ABS -5)
  5
```

Note that MAX selects the largest of the numbers given as arguments. As with MAX, the functions MIN, PLUS, TIMES can work on more than two arguments. DIFFERENCE will keep on subtracting and QUOTIENT will keep on dividing when given more than two arguments. There is never any confusion because the end of the argument sequence is signaled clearly by the right parenthesis. SQRT, of course, takes the square root, and ABS, the absolute value. EXPT calculates powers. SQRT, MINUS, ADD1, SUB1, and ABS all deal with a single argument.

Now consider this expression, in which PLUS is followed by something other than raw numbers:

```
(PLUS (TIMES 2 2) (QUOTIENT 2 2))
```

If we think of this as directions for something to do, it is easy to see that the subexpression (TIMES 2 2) evaluates to 4, (QUOTIENT 2 2) evaluates to 1, and these results fed in turn to PLUS, give 5 as the result. But if the whole expression is viewed as a kind of list, then we see that PLUS is the first element, the entire expression (TIMES 2 2) is the second element, and (QUOTIENT 2 2) is the third. Thus lists themselves can be elements of other lists. Said another way, we permit lists in which individual elements are lists themselves. In part the representational power of LISP derives from this ability to build nested structures out of lists.

Things like PLUS and 3.14, which have obvious meaning, as well as things like FOO, B27, and 123XYZ, are called atoms. Atoms that are not numbers are called *symbolic atoms.*

- Both atoms and lists are often called *s-expressions, s* being short for symbolic. S-expressions are the *data objects* manipulated by LISP.

Problems

Problem 2-1: Each of the following things may be an atom, a list, an s-expression, some combination, or a malformed expression. Identify each accordingly.

```
ATOM

(THIS IS AN ATOM)

(THIS IS AN S-EXPRESSION)

((A B) (C D))

3

(3)

(LIST 3)

(QUOTIENT (ADD1 3) (SUB1 3))

)(

((()))

(() ())

((())

())(

((ABC
```

Problem 2-2: Evaluate the following s-expressions:

```
(QUOTIENT (ADD1 3) (SUB1 3))

(TIMES (MAX 3 4 5) (MIN 3 4 5))

(MIN (MAX 3 1 4) (MAX 2 7 1))
```

LISP Handles Both Fixed and Floating Numbers

It would be diversionary to discuss at length the difference between the so-called *fixed-point* and *floating-point* numbers. Like many other programming languages LISP can handle both. Let it suffice to say that fixed-point numbers are used to represent integers, while floating-point numbers are used to represent reals. Programming beginners can ignore the difference for the moment. Others should know that FLOAT converts fixed-point numbers into floating-point ones, while FIX goes the other way, producing the largest integer that is less than or equal to its argument. Further details are given in Note 1 of Appendix 4.

We can now summarize the relationships between the various types of objects manipulated by LISP. This is done graphically in figure 2-1.

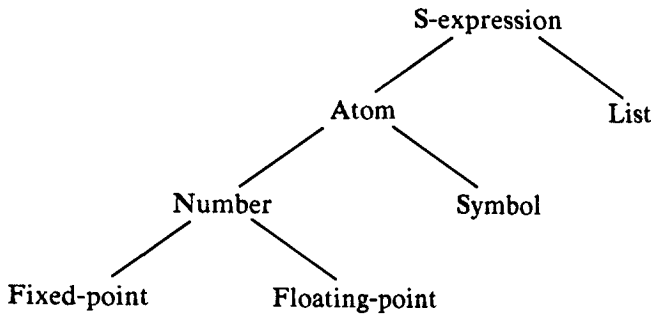


Figure 2-1: The relationships between various types of objects in LISP. An s-expression can be a list or an atom; an atom can be a symbol or a number; a number can be floating-point or fixed-point.

CAR and CDR Take Lists Apart

Examples from arithmetic are simple, but arithmetic does not expose the talent of LISP for manipulating s-expressions. Suppose we have an s-expression like (FAST COMPUTERS ARE NICE). We might like to chip off the first element leaving (COMPUTERS ARE NICE), or we might like to insert a new first element producing something like (BIG FAST COMPUTERS ARE NICE). It is time to look at such manipulations starting with basic techniques for dissecting and constructing lists. In particular we must understand the functions CAR, CDR, APPEND, LIST, and CONS. A regrettable historical convention has left two of these five key functions terribly nonmnemonic — their meaning simply has to be memorized.

Some examples will help explain how the basic CAR and CDR functions work. Again, do not worry about the quote marks that appear. They will be explained soon. To work. CAR returns the first element in a list:

```
(CAR '(FAST COMPUTERS ARE NICE))
  FAST

(CAR '(A B C))
  A
```

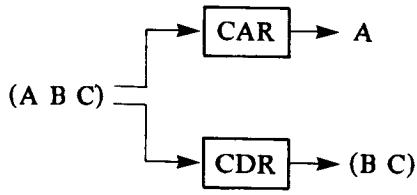
In the next example the argument given to CAR is the two-element list ((A B) C). The first element is itself a list, (A B), and being the first element of the argument, (A B) is returned by CAR.

```
(CAR '((A  B) C))
  (A B)
```

CDR does the complementary thing. It returns a list containing all but the first element.

```
(CDR '(FAST COMPUTERS ARE NICE))
  (COMPUTERS ARE NICE)

(CDR '(A B C))
  (B C)

(CDR '((A B) C))
  (C)
```

Note that CDR, unlike CAR, always returns a list. Remembering the following diagram may help keep the asymmetry of CAR and CDR straight:



Also note that when CDR is applied to a list with only one element, it returns the *empty list*, sometimes denoted by ().

Evaluation is often Purposely Inhibited by Quoting

Now it is time to understand those quote marks that have been appearing. We will use the fact that CAR and CDR operations can be nested together just like the arithmetic functions. To pick out the second element of some list, the first function to use is CDR and the second is CAR. Thus if we want the second element of (A B C), it might seem reasonable to write this:

```
(CAR (CDR (A B C)))
```

There is a problem, however. We want CDR to take (A B C) and give back (B C). Then CAR would certainly return B, the second element in the original list. But how is LISP to know where the specification of what to do leaves off and the data to be manipulated begins? Look at the embedded list:

```
(A B C)
```

LISP might legitimately think that A is some sort of function, perhaps one defined by the user. Similarly, the following s-expression is certainly a list:

```
(CDR (A B C))
```

And its first element is surely CDR! Thus the following expression could well result in an answer of CDR:

```
(CAR (CDR (A B C)))
```

How far should the evaluation process go into an s-expression? LISP needs help in making this decision. The user specifies where to stop evaluation by supplying an evaluation-inhibiting signal in the form of a quote character, '. Thus the following expression returns B:

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

B is returned because the quote mark prevents LISP from wading in and thinking of (A B C) as an s-expression in which A is a function to be applied to B and C. Instead, (A B C) is given to CDR which then hands (B C) to CAR resulting finally in just plain B.

Moving the quote mark changes the result. If we type (CAR '(CDR (A B C))), then LISP does not try to take the CDR of anything but simply gives the s-expression (CDR (A B C)) to CAR as a list to work on, resulting in CDR since CDR is the first element.

Leaving out the quote mark altogether would result in an effort to use A as a function. There is no function supplied by LISP and if none had been defined by the user, LISP would report a so-called undefined function error.

- It is important to know that the scope of the quoting mechanism is exactly the immediately following s-expression. A quote in front of a list prevents any attempt at evaluating the list.

Sometimes it is useful to know about another, older way to stop evaluation. The s-expression to be protected against evaluation is simply bracketed on the left by a left parenthesis followed by the atom QUOTE and on the right by a matching right parenthesis. Thus the following stops evaluation of the list (A B C):

```
(QUOTE (A B C))
  (A B C)

(CAR (CDR (QUOTE A B C)))
  B
```

Note then that '(A B C) is equivalent to (QUOTE (A B C)).

Problems

Problem 2-3: Evaluate the following s-expressions:

```
(CAR '(P H W))

(CDR '(B K P H))

(CAR '((A B) (C D)))

(CDR '((A B) (C D)))

(CAR (CDR '((A B) (C D))))

(CDR (CAR '((A B) (C D))))

(CDR (CAR (CDR '((A B) (C D)))))

(CAR (CDR (CAR '((A B) (C D)))))
```

Problem 2-4: Evaluate the following s-expressions:

```
(CAR (CDR (CAR (CDR '((A B) (C D) (E F))))))

(CAR (CAR (CDR (CDR '((A B) (C D) (E F))))))

(CAR (CAR (CDR '(CDR ((A B) (C D) (E F))))))
```

```
(CAR (CAR '(CDR (CDR ((A B) (C D) (E F))))))

(CAR '(CAR (CDR (CDR ((A B) (C D) (E F))))))

'(CAR (CAR (CDR (CDR ((A B) (C D) (E F))))))
```

Problem 2-5: Write sequences of CARs and CDRs that will pick the atom PEAR out of the following s-expression:

```
(APPLE ORANGE PEAR GRAPEFRUIT)

((APPLE ORANGE) (PEAR GRAPEFRUIT))

(((APPLE) (ORANGE) (PEAR) (GRAPEFRUIT)))

(APPLE (ORANGE) ((PEAR)) (((GRAPEFRUIT))))

((((APPLE))) ((ORANGE)) (PEAR) GRAPEFRUIT)

((((APPLE) ORANGE) PEAR) GRAPEFRUIT)
```

Composing CARs and CDRs Makes Programming Easier

When many CARs and CDRs are needed to dig out some item from deep inside an s-expression, it is usually convenient to substitute a composite function of the form CxR, CxxR, CxxxR, or CxxxxR. Each x is either an A, signifying CAR, or D, signifying CDR. Thus (CADR '(A B C)) is completely equivalent to (CAR (CDR '(A B C))).

Atoms Have Values

So far we have seen how symbolic structures can be taken apart by evaluating lists that begin with CAR or CDR. We have also seen how arithmetic can be done by evaluating lists that begin with PLUS, DIFFERENCE, and other similar functions. Indeed it seems like LISP's goal is always to evaluate something and return a value. This is true for atoms as well as lists. Suppose we type a the name of an atom, followed by a space, and wait for LISP to respond:

```
X
```

On seeing X, LISP tries to return a value for it, just as it would if some s-expression like (PLUS 3 4) were typed. But for an atom, the *value* is something looked up somewhere, rather than the result of some computation as

when dealing with lists.

The value of a symbolic atom is established by a special function, SET. It causes the value of its second argument to become the value of the first argument. Typing (SET 'L '(A B)) results in a value of (A B) for the expression. But more importantly, there is a side effect because (A B) becomes the value of L. If we now type L, we see that (A B) comes back:

```
L
  (A B)
```

Thus the expression (SET 'L '(A B)) is executed mainly for the side effect of giving a value to the atom L.

Now since the value of L is (A B), L can be used in working through some examples of the basic list manipulating functions. These illustrate that LISP seeks out the value of atoms not only when they are typed in by themselves, but also when the atoms appear as arguments to functions.

```
L
  (A B)

'L
  L

(CAR L)
  A

(CAR 'L)
  ERROR

(CDR L)
  (B)

(CDR 'L)
  ERROR
```

Note that both CAR and CDR announce an error if asked to work on an atom. Both expect a list as their argument and both work only when they get one.

- All numbers are treated specially in that the value of a number is just the number itself.

APPEND, LIST, and CONS Construct Lists

While CAR and CDR take things apart, APPEND, LIST, and CONS put them

together. APPEND strings together the elements of all lists supplied as arguments:

```
(SET 'L '(A B))
  (A B)

(APPEND L L)
  (A B A B)

(APPEND L L L)
  (A B A B A B)

(APPEND '(A) '() '(B) '())
  (A B)

(APPEND 'L L)
  ERROR
```

Be sure to understand that APPEND runs the elements of its arguments together, but does nothing to those elements themselves. Note that the value returned in the following example is ((A) (B) (C) (D)), not (A B C D):

```
(APPEND '((A) (B)) '((C) (D)))
  ((A) (B) (C) (D))
```

LIST does not run things together like APPEND does. Instead, it makes a list out of its arguments. Each argument becomes an element of the new list.

```
(LIST L L)
  ((A B) (A B))

(LIST L L L)
  ((A B) (A B) (A B))

(LIST 'L L)
  (L (A B))

(LIST '((A) (B)) '((C) (D)))
  (((A) (B)) ((C) (D)))
```

CONS takes a list and inserts a new first element. CONS is a mnemonic for list constructor.

- Let us adopt a convention by which the words between angle brackets are taken as descriptions of what should appear in the position occupied.

Then the CONS function can be described as follows:

```
(CONS <new first element> <some list>)
```

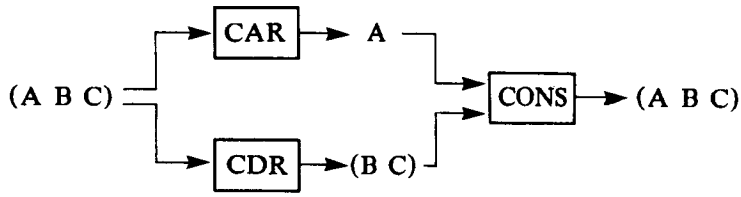
Thus we have these:

```
(CAR (CONS 'A '(B C)))
  A

(CDR (CONS 'A '(B C)))
  (B C)
```

And working again with L, whose value was established to be (A B), we have a sort of do-nothing combination of CONS, CAR, and CDR:

```
L
  (A B)

(CONS (CAR L) (CDR L))
  (A B)
```

Representing this inverse relationship between CONS and the pair CAR and CDR in a diagram, we have this:



Note that (CONS L '(C D)) yields the result ((A B) C D) in which the list (A B) has become the first element of a list that was just (C D) before. Study how APPEND, LIST, and CONS differ:

```
(APPEND '(A B) '(C D))
  (A B C D)

(LIST '(A B) '(C D))
  ((A B) (C D))
```

```
(CONS '(A B) '(C D))
  ((A B) C D)

(APPEND L L)
  (A B A B)

(LIST L L)
  ((A B) (A B))

(CONS L L)
  ((A B) A B)

(APPEND 'L L)
  ERROR

(LIST 'L L)
  (L (A B))

(CONS 'L L)
  (L A B)
```

Problems

Problem 2-6: Evaluate the following s-expressions in the order given:

```
(SET 'TOOLS (LIST 'HAMMER 'SCREWDRIVER))

(CONS 'PLIERS TOOLS)

TOOLS

(SET 'TOOLS (CONS 'PLIERS TOOLS))

TOOLS

(APPEND '(SAW WRENCH) TOOLS)

TOOLS

(SET 'TOOLS (APPEND '(SAW WRENCH) TOOLS))

TOOLS
```

LENGTH, REVERSE, SUBST, and LAST Round out a Basic Repertoire

LENGTH counts the number of elements in a list. REVERSE turns a list around. Both consider what they get to be a list of elements, not caring whether the elements are atoms or lists. Often they are used on lists that have lists as elements, but they do nothing with the insides of those elements. Assume, in the following examples, that the value of L is still (A B).

```
(LENGTH '(A B))
  2

(LENGTH '((A B) (C D)))
  2

(LENGTH L)
  2

(LENGTH (APPEND L L))
  4

(REVERSE '(A B))
  (B A)

(REVERSE '((A B) (C D)))
  ((C D) (A B))

(REVERSE L)
  (B A)

(REVERSE (APPEND L L))
  (B A B A)
```

SUBST takes three arguments, one of which is an s-expression in which occurences of a specified s-expression are to be replaced. The first argument is the new s-expression to be substituted in; the second is the s-expression to be substituted for; and the third is the s-expression to work on:

```
(SUBST <new s-expression> <old s-expression> <s-expression to work on>)
```

Study the following, in which the s-expressions to be substituted for are atoms:

```
(SUBST 'A 'B '(A B C))
  (A A C)
```

```
(SUBST 'B 'A '(A B C))
  (B B C)

(SUBST 'A 'X (SUBST 'B 'Y '(SQRT (PLUS (TIMES X X) (TIMES Y Y)))))
  (SQRT (PLUS (TIMES A A) (TIMES B B)))
```

LAST returns a list which contains only the last element of the list given as the argument:

```
(LAST '(A B C))
  (C)

(LAST '((A B) (C D)))
  ((C D))

(LAST 'A)
  ERROR
```

Problems

Problem 2-7: Evaluate the following s-expressions:

```
(LENGTH '(PLATO SOCRATES ARISTOTLE))

(LENGTH '((PLATO) (SOCRATES) (ARISTOTLE)))

(LENGTH '((PLATO SOCRATES ARISTOTLE)))

(REVERSE '(PLATO SOCRATES ARISTOTLE))

(REVERSE '((PLATO) (SOCRATES) (ARISTOTLE)))

(REVERSE '((PLATO SOCRATES ARISTOTLE)))
```

Problem 2-8: Evaluate the following s-expressions:

```
(LENGTH '((CAR CHEVROLET) (DRINK COKE) (CEREAL WHEATIES)))

(REVERSE '((CAR CHEVROLET) (DRINK COKE) (CEREAL WHEATIES)))

(APPEND '((CAR CHEVROLET) (DRINK COKE))
        (REVERSE '((CAR CHEVROLET) (DRINK COKE))))
```

Problem 2-9: Evaluate the following s-expressions:

```
(SUBST 'OUT 'IN '(SHORT SKIRTS ARE IN))

(SUBST 'IN 'OUT '(SHORT SKIRTS ARE IN))

(LAST '(SHORT SKIRTS ARE IN))
```

The Interpreter Evaluates S-expressions

When an expression is typed by a LISP user, it is automatically handed to the function EVAL. The diagram in figure 2-2 describes how this function works.

Note, in particular, that some functions get special handling. SETQ is such a function because it handles its arguments in a nonstandard way: SETQ is like SET except that SETQ makes no attempt to evaluate the first argument. SETQ is much more popular than SET.

```
(SETQ ZERO 0)
  0
```

Sometimes the pairs of arguments to several SETQs are run together and given to a single SETQ. The odd numbered arguments are not evaluated but the even ones are, as would be expected. Thus one SETQ can do the work that would otherwise require many:

```
(SETQ ZERO 0 ONE 1 TWO 2 THREE 3 FOUR 4
      FIVE 5 SIX 6 SEVEN 7 EIGHT 8 NINE 9)
  9

ZERO
  0

NINE
  9
```

EVAL Causes Extra Evaluation

Note that one can use the function EVAL explicitly to call for another round of evaluation beyond the one already employed to evaluate the arguments of a function. The following example will illustrate:

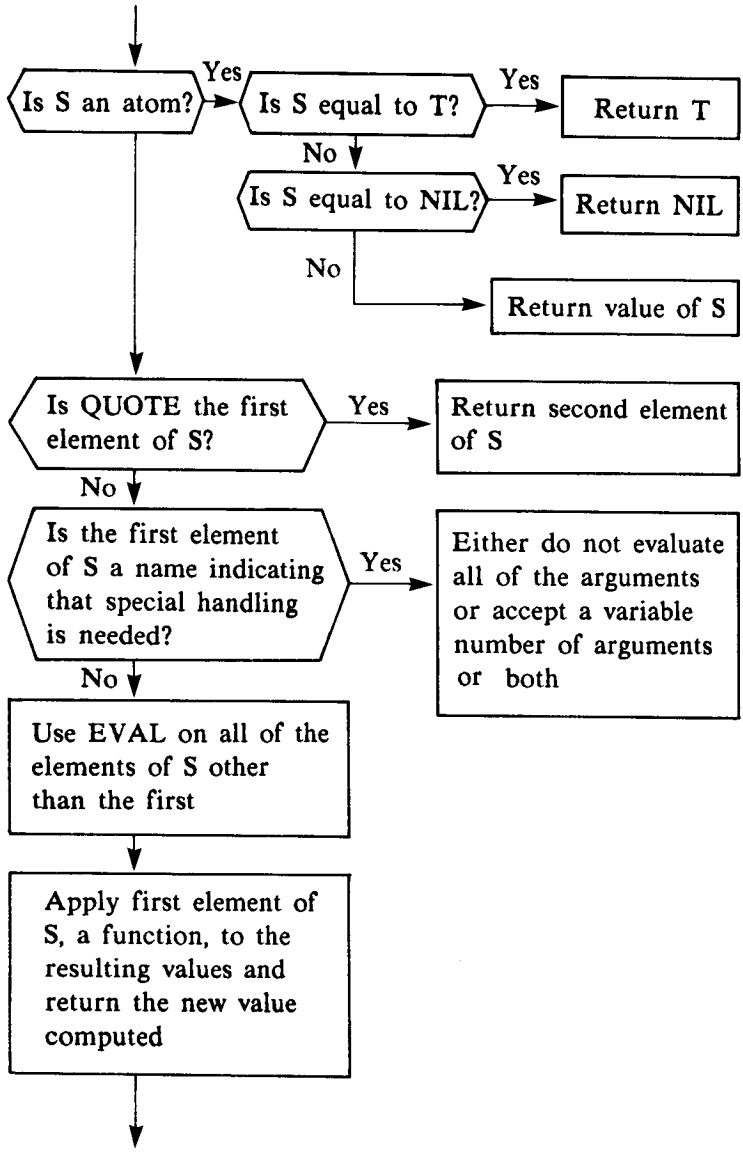


Figure 2-2: Definition of EVAL. This version assumes quoting is done by using (QUOTE <s-expression>) rather than '<s-expression>.

```
(SETQ A 'B)
  B

(SETQ B 'C)
  C

A
  B

B
  C

(EVAL A)
  C
```

The atom A is first evaluated because it is the unquoted argument to a function. The result is then evaluated because the function is EVAL. EVAL causes whatever the value is to be evaluated!

Later on we shall see that EVAL will be used when a program-writing program needs to use what it has written.

Problems

Problem 2-10: Evaluate the following s-expressions in the order given:

```
(SETQ METHOD1 'PLUS)

(SETQ METHOD2 'DIFFERENCE)

(SETQ METHOD METHOD1)

METHOD

(EVAL METHOD)

(SETQ METHOD 'METHOD1)

METHOD

(EVAL METHOD)

(EVAL (EVAL '(QUOTE METHOD)))
```

Summary

- LISP means symbol manipulation.
- LISP programs and data are constructed out of s-expressions.
- LISP handles both fixed and floating numbers.
- CAR and CDR take lists apart.
- Evaluation is often purposely inhibited by quoting.
- Composing CARs and CDRs makes programming easier.
- Atoms have values.
- APPEND, LIST, and CONS construct lists.
- LENGTH, REVERSE, SUBST, and LAST round out a basic repertoire.
- The interpreter evaluates s-expressions.
- EVAL causes extra evaluation.

3
DEFINITIONS PREDICATES CONDITIONALS AND SCOPING

The first purpose of this chapter is to explain how you can create your own functions. The second purpose is to show how tests make it possible to do things conditionally.

Caution: the words *procedure*, *function*, and *program* will be used throughout this book. For our purpose, it is not necessary to be precise about the differences. Generally speaking, though, a *procedure* is a recipe for action. A *function* is a kind of procedure for which it is convenient to think in terms of an output called the value and inputs called the arguments. Mathematical procedures, in particular, are often called functions. A *program* is an embodiment of a procedure in a particular programming language.

DEFUN Enables a User to Make New Functions

We now have some ingredients. Let us see how they can be combined into new functions using DEFUN. It has the following syntax:

```
(DEFUN <function name>
       (<parameter 1>  <parameter 2> ... <parameter n>)
       <process description>)
```

As before, the angle brackets delineate descriptions of things (the descriptions may denote atoms, lists, or even fragments as appropriate).

DEFUN does not evaluate its arguments. It just looks at them and establishes a function definition, which can later be referred to by having the function name appear as the first element of a list to be evaluated. The function name must be a symbolic atom. When DEFUN is used, like any function, it gives back a value.

The value DEFUN gives back is the function name, but this is of little consequence since the main purpose of DEFUN is to establish a definition, not to return some useful value.

The value a function gives back when used is called the *value returned.* Anything a function has done that persists after it returns its value is called a *side effect.* DEFUN and SETQ both have side effects. The side effect of DEFUN is to set up a function definition. The side effect of SETQ is to give a value to an atom.

The list following the function name is called the function's *parameter list.* Each parameter is a symbolic atom which may appear in the <process description> part of the definition. Its value is determined by the value of one of the arguments when the function is called.

Let us consider an example immediately. Using DEFUN it is easy to define a function that converts temperatures given in degrees Fahrenheit to degrees Celsius:

```
(DEFUN F-TO-C (TEMP)
       (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
  F-TO-C
```

When F-TO-C is used, it appears as the first element in a two-element list. The second element is F-TO-C's argument. After the argument is evaluated, it becomes the temporary value of the function's *parameter.* In this case, TEMP is the parameter, and it is given the value of the argument while F-TO-C is being evaluated. It is as if the operations shown in figure 3-1 are carried out.

Suppose, for example, that the value of SUPER-HOT is 100. Then consider this:

```
(F-TO-C SUPER-HOT)
```

To evaluate this expression, SUPER-HOT is evaluated, handing 100 to the function F-TO-C. On entry to F-TO-C, 100 becomes the temporary value of TEMP. Consequently when the body of the function is evaluated, it is as if the following were evaluated:

```
(QUOTIENT (DIFFERENCE 100 32) 1.8)
```

The value returned is 37.77. If TEMP had a value before F-TO-C was evaluated, that value is restored. If TEMP had no value before F-TO-C was evaluated, it has no value afterward either.

- Sometimes several functions use the same parameter. If one such function calls another, which calls another, and so on, it is necessary for LISP to keep track of several values of the common parameter. This makes it possible to restore the parameter's values properly as the function evaluations are completed later.

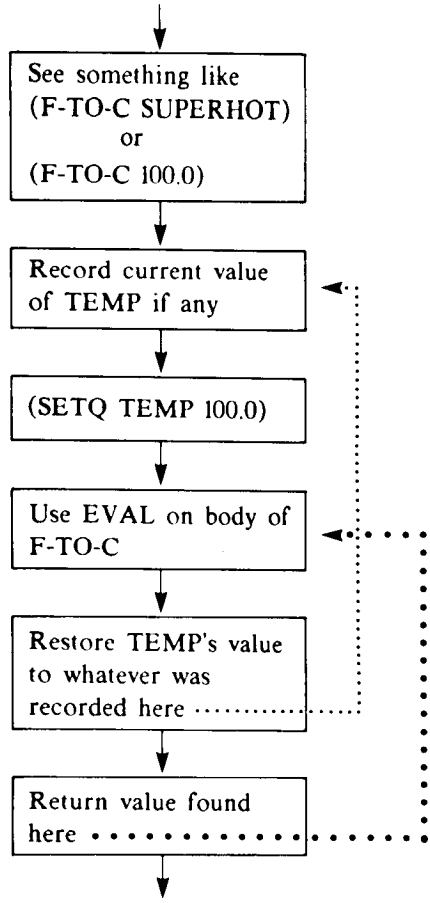


Figure 3-1: When entering a function, the parameters receive values which are used in the evaluation of the body. The old values of these parameters are saved so that they can be restored when leaving.

Now consider another example, this time involving a function that does a symbolic computation, rather than a numerical one. This new function exchanges the first and second elements of a two-element list:

```
(DEFUN EXCHANGE (PAIR)
       (LIST (CADR PAIR) (CAR PAIR)))                    ;Reverse elements.
  EXCHANGE
```

Note the so-called *comment*. LISP totally ignores semicolons and anything that appears after them on the same line. This makes it possible to annotate programs without interfering with their operation. Liberal use of comments is considered good programming practice.

To see how EXCHANGE works, suppose the value of SINNERS is (ADAM EVE). Then to evaluate (EXCHANGE SINNERS), the first thing LISP does is evaluate the argument, SINNERS. The value of SINNERS then becomes the temporary value of PAIR while EXCHANGE is doing its job. Consequently, (CADR PAIR) is EVE, (CAR PAIR) is ADAM, and the following is evident:

```
(EXCHANGE SINNERS)
  (EVE ADAM)
```

The next example introduces a function with two parameters, rather than just one. Its purpose is to compute percentage. More specifically, the value returned is to be the percentage by which the second argument given to the function is greater than the first.

```
(DEFUN INCREASE (X Y)
        (QUOTIENT (TIMES 100.0 (DIFFERENCE Y X)) X))     ;Nested operations.
  INCREASE

(INCREASE 10.0 15.0)
  50.0
```

In the example, the temporary value of X becomes 10.0 on entry, and the temporary value of Y becomes 15.0. Note that there is never any confusion about how to match X and Y with 10.0 and 15.0 — LISP knows the correct way to match them up because DEFUN's parameter list specifies the order in which parameters are to be paired with incoming arguments.

Finally, note that a function definition can employ any number of s-expressions. The last one always determines the value returned. The others therefore are useful only in that they cause some side effect, as in this altered definition of F-TO-C:

```
(DEFUN F-TO-C (TEMP)
        (SETQ TEMP (DIFFERENCE TEMP 32))          ;Subtract.
        (QUOTIENT TEMP 1.8))                      ;Divide.
```

In this second version, TEMP is used as an input parameter and as a temporary anchor for the difference between the input and 32. The DEFUN creates a program that specifies two sequential steps.

Note that one can build up a set of functions incrementally. Each can be entered and tested in turn. If a function is found to be faulty, it can be replaced right away by simply using DEFUN again.

Problems

Problem 3-1: Some people are annoyed by the nonmnemonic character of the critical functions CAR, CDR, and CONS. Define new functions FIRST, REST, and INSERT that do the same things. Note that SECOND, THIRD, and similar functions are equally easy to create.

Problem 3-2: Define ROTATE-L, a function that takes a list as its argument and returns a new list in which the former first element becomes the last. The following illustrates:

```
(ROTATE-L '(A B C))
  (B C A)

(ROTATE-L (ROTATE-L '(A B C))
  (C A B)
```

Problem 3-3: Define ROTATE-R. It is to be like ROTATE-L except that it is to rotate in the other direction.

Problem 3-4: A palindrome is a list that has the same sequence of elements when read from right to left that it does when read from left to right. Define PALINDROMIZE such that it takes a list as its argument and returns a palindrome that is twice as long.

Problem 3-5: When converting between degrees Fahrenheit and degrees Celsius, it is useful to note that —40 Fahrenheit equals —40 Celsius. This observation makes for the following symmetric conversion formulas:

$$C = (F + 40) \;/\; 1.8 - 40$$
$$F = (C + 40) * 1.8 - 40$$

Define conversion functions, F-TO-C and C-TO-F, using these formulas.

Problem 3-6: Define ROOTS, a function with three parameters, A, B, and C. ROOTS is to return a list of the two roots of the polynomial $ax^2 + bx + c$, using this formula:

$$x = [-b \pm (b^2 - 4ac)^{1/2}] \;/\; (2a)$$

Assume that the roots are real.

A Predicate Is a Function that Returns T or NIL

More complicated definitions require the use of functions called predicates. A *predicate* is a function that returns one of two special atoms, T or NIL. These values of T and NIL correspond to logical true and false.

- Note that T and NIL are special atoms in that their values are preset to T and NIL. That is, the value of T is T and the value of NIL is NIL.

Consider ATOM, for example. ATOM is a predicate that tests its argument to see if it is an atom. To facilitate showing how ATOM and other important predicates behave, let us make the value of the atom DIGITS be a list of the names of the numbers from zero to nine:

```
(SETQ DIGITS '(ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE))
  (ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE)
```

Now ATOM can be demonstrated:

```
(ATOM 'DIGITS)
  T

(ATOM 'FIVE)
  T

(ATOM DIGITS)
  NIL

(ATOM '(ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE))
  NIL
```

BOUNDP expects a symbolic atom as its argument. It tests if its argument has a value. If it does, it returns T. Otherwise it returns NIL.

```
(BOUNDP 'DIGITS)
  T

(BOUNDP 'ZERO)
  NIL

(BOUNDP (CAR DIGITS))
  NIL
```

EQUAL is another fundamental predicate. It takes two arguments and returns T if they are the same. Otherwise it returns NIL:

```
(EQUAL DIGITS DIGITS)
  T

(EQUAL 'DIGITS 'DIGITS)
  T

(EQUAL DIGITS '(ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE))
  T

(EQUAL DIGITS 'DIGITS)
  NIL
```

NULL checks to see if its argument is an empty list:

```
(NULL '())
  T

(NULL T)
  NIL

(NULL DIGITS)
  NIL

(NULL 'DIGITS)
  NIL
```

MEMBER tests to see if one s-expression is an element of another. It would make sense for MEMBER to return T if the first argument is an element of the following list and NIL otherwise. Actually MEMBER returns the fragment of the list that begins with the first argument if the first argument is indeed an element of the list. This is a special case of a common programming convenience. The general idea is that a function can return either NIL or something that is both nonNIL and useful for feeding further computation.

```
(MEMBER 'FIVE DIGITS)
  (FIVE SIX SEVEN EIGHT NINE)

(MEMBER 'TEN DIGITS)
  NIL
```

The first argument must be an element of the second argument. It is not enough for the first argument to be buried somewhere in the second argument, as in this example:

```
(MEMBER 'FIVE '((ZERO TWO FOUR SIX EIGHT) (ONE TWO THREE FOUR FIVE)))
  NIL
```

The following use of MEMBER exploits the fact that MEMBER returns something other than T when an element is present. It determines the number of digits after the first instance of the atom FIVE in the list DIGITS.

```
(SUB1 (LENGTH (MEMBER 'FIVE DIGITS)))
  4
```

For some further examples, let us establish values for the elements of DIGITS:

```
(SETQ ZERO 0 ONE 1 TWO 2 THREE 3 FOUR 4
      FIVE 5 SIX 6 SEVEN 7 EIGHT 8 NINE 9)
  9
```

Now using these values, we can look at some predicates that work on numbers. NUMBERP tests its argument to see if it is a number:

```
(NUMBERP 3.14)
  T

(NUMBERP FIVE)
  T

(NUMBERP 'FIVE)
  NIL

(NUMBERP DIGITS)
  NIL

(NUMBERP 'DIGITS)
  NIL
```

GREATERP and LESSP expect their arguments to be numbers. GREATERP tests them to see that they are in strictly descending order. LESSP checks to see that they are in strictly ascending order. Both may be given any number of arguments.

```
(GREATERP FIVE 2)
  T

(GREATERP 2 FIVE)
  NIL
```

```
(LESSP 2 FIVE)
  T

(LESSP FIVE 2)
  NIL

(LESSP 2 2)
  NIL

(GREATERP FIVE FOUR THREE TWO ONE)
  T

(GREATERP THREE ONE FOUR)
  NIL

(GREATERP 3 1 4)
  NIL
```

ZEROP expects a number. It tests its argument to see if it is zero:

```
(ZEROP ZERO)
  T

(ZEROP 'ZERO)
  ERROR

(ZEROP FIVE)
  NIL
```

MINUSP tests whether a number is negative:

```
(MINUSP ONE)
  NIL

(MINUSP (MINUS ONE))
  T

(MINUSP ZERO)
  NIL
```

Note that several predicates end in P, a mnemonic for predicate. The predicate ATOM is an unfortunate exception that would tend to suggest that LIST is a predicate too.

Problems

Problem 3-7: Define EVENP, a predicate that tests whether a number is even. You will need REMAINDER, a function that returns the remainder produced by dividing the first argument by the second (see Appendix 4, Note 1).

Problem 3-8: Define PALINDROMEP, a predicate that tests its argument to see if it is a list that has the same sequence of elements when read from right to left that it does when read from left to right.

Problem 3-9: Define RIGHTP, a predicate that takes three arguments. The arguments are the lengths of the sides of a triangle that may be a right triangle. RIGHTP is to return T if the sum of the squares of the two shorter sides is within 2% of the square of the longest side. Otherwise RIGHTP is to return NIL. You may assume the longest side is given as the first argument.

Problem 3-10: Define COMPLEXP, a predicate that takes three arguments, A, B, and C, and returns T if $b^2 - 4ac$ is less than zero.

AND, OR, and NOT are used to do Logic

NOT returns T only if its argument is NIL.

```
(NOT NIL)
  T

(NOT T)
  NIL

(NOT 'DOG)
  NIL

(SETQ PETS '(DOG CAT))
  (DOG CAT)

(NOT (MEMBER 'DOG PETS))
  NIL

(NOT (MEMBER 'TIGER PETS))
  T
```

AND and OR make composite tests possible. AND returns nonNIL only if all arguments are nonNIL. OR returns nonNIL if any argument is nonNIL. Both take any number of arguments.

```
(AND T T NIL)
  NIL

(OR T T NIL)
  T

(AND (MEMBER 'DOG PETS) (MEMBER 'TIGER PETS))
  NIL

(OR (MEMBER 'DINGO PETS) (MEMBER 'TIGER PETS))
  NIL
```

AND and OR do not treat all of their arguments the same way:

- AND evaluates its arguments from left to right. If a NIL is encountered, NIL is returned immediately. Any remaining arguments are not even evaluated. Otherwise AND returns the value of its last argument.

In other words, anything other than NIL behaves like T as far as logical considerations are concerned. This is a great feature. An OR behaves similarly:

- OR evaluates its arguments from left to right. If something other than NIL is encountered, it is returned immediately. Any remaining arguments are not even evaluated. Otherwise OR returns NIL.

Consider the following examples, remembering that MEMBER returns the remainder of its second argument if it finds its first argument in that argument:

```
(AND (MEMBER 'DOG PETS) (MEMBER 'CAT PETS))
  (CAT)

(OR (MEMBER 'DOG PETS) (MEMBER 'TIGER PETS))
  (DOG CAT)
```

Predicates Help COND Select a Value among Alternatives

The predicates are most often used to determine which of several possible s-expressions should be evaluated. The choice is most often determined by predicates in conjunction with the branching function COND. COND is therefore an extremely common function. Regrettably, it has a somewhat peculiar syntax. The function name COND is followed by a number of lists, each of which contains a test and something to return if the test succeeds. Thus the syntax is as follows:

```
(COND (<test 1> ... <result 1>)
      (<test 2> ... <result 2>)
      .
      .
      .
      (<test n> ... <result n>))
```

Each list is called a *clause.* The idea is to search through the clauses evaluating only the first element of each until one is found whose value is nonNIL. Then everything else in the successful clause is evaluated and the last thing evaluated is returned as the value of the COND. Any expressions standing between the first and the last elements in a COND clause must be there only for side effects since they certainly cannot influence the value of the COND directly. There are two special cases:

- If no successful clause is found, COND returns NIL.
- If the successful clause consists of only one element, then the value of that element itself is returned. Said another way, the test and result elements may be the same.

It may seem strange, incidentally, that T is not strictly required to trigger a clause, and anything other than NIL will do. This is a desirable feature because it allows test functions whose outcomes are not limited to T and NIL values. Since nonNIL is the same as T as far as COND is concerned, a test function can often return something that is both nonNIL and potentially useful. MEMBER is such a function.

While on the subject of strange things, one particular one deserves special attention:

- The empty list, (), and NIL are equivalent in all respects. For example, they satisfy the equal predicate: (EQUAL NIL '()) returns T. (By convention, the empty list is printed out as NIL.)

Some say the reason that the equivalence of (), and NIL was originally arranged has to do with the instruction set of the ancient 709 computer. Others believe the identity was always known to be a programming convenience. No one seems to know for sure. In any case, the first element in a COND clause is frequently an atom whose value is a list that may be empty. If the list is empty, it does not trigger the COND clause since it acts like NIL. On the other side, occasional bugs derive from the fact that (ATOM '()) is T.

Note that NULL and NOT are actually equivalent functions because NULL returns T only if its argument is an empty list and NOT returns T only if its argument is NIL.

Also, while on the subject of the empty list, it is important to know what happens when CAR or CDR gets one. They may or may not be happy depending

on the particular LISP implementation. One argument is that CAR returns the first element of a list and should complain if there is none. The same argument holds that CDR returns the rest of a list after the first is removed and should complain if there is nothing to remove.

A better argument is that both CAR and CDR should return NIL if given an empty list by reason of programming convenience. Otherwise it is constantly necessary to test a list before working on it.

- The CAR and CDR of the LISP used in the programs in this book both return NIL when given an empty list.

COND Enables DEFUN to do More

Now the ingredients are ready so let us bake the cake. Suppose we want to have a function that adds a new element to the front of a list only if it is not already in the list. Otherwise the value returned is to be the unaltered list. Clearly the desired function must do a test and act according to the result. This will do:

```
(DEFUN AUGMENT (ITEM BAG)
       (COND ((MEMBER ITEM BAG) BAG)                ;Already in?
             (T (CONS ITEM BAG))))                  ;Add ITEM to front.
  AUGMENT
```

Note again that the list of parameters shows how arguments are to be used by the function. Without displaying the names somewhere in the definition in the order they are to be given to the function, there would be no way of deciding if (AUGMENT '(A B) '(X Y)) should yield ((A B) X Y) or ((X Y) A B).

Problems

Problem 3-11: In some LISPs, trying to take the CAR or CDR of NIL causes an error. Define NILCAR and NILCDR in terms of CAR and CDR such that they work like CAR and CDR, but return NIL if given NIL as their argument no matter what CAR and CDR do.

Problem 3-12: Some people prefer the Fahrenheit scale to the Celsius scale, because they find it aesthetically pleasing that 0^{o} and 100^{o} are pinned to temperatures that bracket the temperature spectrum of temperate climates, 0^{o} being ridiculously cold and 100^{o} being ridiculously hot.

Define CHECK-TEMPERATURE, a function that is to take one argument, such that it returns RIDICULOUSLY-HOT if the argument is greater than 100, RIDICULOUSLY-COLD if the argument is less than 0, and OK otherwise.

Variables May Be Free or Bound

In LISP the process of relating arguments to parameters on entering a function is called *lambda binding*. The following function is strangely written in order to clarify some details:

```
(DEFUN INCREMENT (PARAMETER)                    ;PARAMETER is bound.
       (SETQ PARAMETER (PLUS PARAMETER FREE))   ;FREE is free.
       (SETQ OUTPUT PARAMETER))                 ;OUTPUT is free.
  INCREMENT
```

Evidently INCREMENT is to add the value of FREE to its argument, returning the result after arranging for the result to be the value of OUTPUT. Now let us try something:

```
(SETQ PARAMETER 15)
  15

(SETQ FREE 10)
  10

(SETQ OUTPUT 10)
  10

(SETQ ARGUMENT 10)
  10

(INCREMENT ARGUMENT)
  20

OUTPUT
  20

PARAMETER
  15

ARGUMENT
  10
```

The value of OUTPUT was permanently altered while that of PARAMETER was not. PARAMETER was temporarily changed — it became 10 on entry to INCREMENT, and then became 20 by virtue of the SETQ. PARAMETER gets a value on entry and gets restored on exit because it appears in the function's parameter list. We say that PARAMETER is bound with respect to INCREMENT while OUTPUT is free in the same context.

- A *bound variable*, with respect to a function, is an atom that appears in the function's parameter list.

- A *free variable*, with respect to a function, is an atom that does not appear in the function's parameter list.

- It makes no sense to speak of a variable as bound or free unless we also specify with respect to what function the variable is free or bound.

- Bound variable values must be saved so that they may be restored. If a bound variable is also used as a free variable, its value is the current one, not any that may have been saved.

It is sometimes nontrivial to figure out what the value of a free variable is. This problem is discussed in detail in Chapter 23 and in Note 3 of Appendix 4.

Note also that changing the value of PARAMETER using SETQ did not change the value of ARGUMENT, even though PARAMETER inherited its original value from ARGUMENT. In some programming languages ARGUMENT and PARAMETER would get tied together, and as a result, the value of ARGUMENT would change as well.

Problems

Problem 3-13: Define CIRCLE such that it returns a list of the circumference and area of a circle whose radius is given. Assume PI is to be a free variable with the appropriate value.

LISP is Neither Call-by-reference nor Call-by-value

In LISP, the notion of what a variable involves is somewhat different from that of most programming languages. In LISP, we say that atoms are *bound* to objects. The objects are s-expressions. We may think of these s-expressions as the values and the atoms as the variables. In most other languages, however, things are a little different. The basic entities are variables which can be in different states. In the case of a fixed-point variable, for example, the different states correspond to the different numbers that can be represented.

When the argument to a function is in the form of the name of a variable, there are two common options, referred to as *call by reference* and *call by value.* Strictly speaking, LISP can be neither call by value nor call by reference since the notion of what a LISP variable is does not correspond exactly to the notion of variable for which call by value and call by reference have a natural meaning.

Programming beginners can ignore the subtleties involved for now. The details are explored in Note 2 of Appendix 4.

Free-variable Values are Determined Dynamically, not Lexically

A collection of bindings is called an *environment.* The object to which an atom is bound can be found by looking in the environment. If a language uses *dynamic scoping*, as LISP does, the values of free variables are determined by the so-called activation environment, the environment in force when the function requiring the free-variable values is used. If a language uses *lexical scoping* instead, then the values of free variables are determined by the so-called definition environment, the environment in force when the function requiring the free-variable values was defined.

Again, programming beginners can ignore the subtleties involved for now. The details are explored in Note 3 of Appendix 4.

Function Names can be Used as Arguments

Sometimes it is useful for the value of an atom to be a function name. Then the function FUNCALL makes it possible to retrieve the value and use the function.

Let the value of BANKING-FUNCTION be the atom INTEREST, let the value of INTEREST-RATE be 0.1, and let the definition of INTEREST use INTEREST-RATE:

```
(SETQ BANKING-FUNCTION 'INTEREST)
  INTEREST

(SETQ INTEREST-RATE 0.1)
  0.1

(DEFUN INTEREST (BALANCE)
       (TIMES BALANCE INTEREST-RATE))
```

Note that INTEREST-RATE is a free variable with respect to the function INTEREST. Now BANKING-FUNCTION can be used as the first argument to FUNCALL. FUNCALL finds its value, namely INTEREST, and uses the rest of the arguments to feed INTEREST:

```
(FUNCALL BANKING-FUNCTION 100.0)
  10.0
```

Evidently all of the following produce exactly the same result:

```
(INTEREST 100.0)

(FUNCALL 'INTEREST 100.0)

(FUNCALL BANKING-FUNCTION 100.0)
```

Actually, since the first argument to FUNCALL is evaluated, anything at all that is subject to evaluation can be there, including a COND. Thus the following would key the interest function used to the type of account:

```
(SETQ ACCOUNT-TYPE 'NORMAL)
  NORMAL

(FUNCALL (COND ((EQUAL ACCOUNT-TYPE 'NORMAL) 'INTEREST)
               ((EQUAL ACCOUNT-TYPE '90DAY) 'HIGHINTEREST)
               (T 'NOINTEREST))
         100.0)
  10.0
```

Certainly the same effect could be obtained in another way, but there are many situations in which the use of FUNCALL is appropriate, rather than contrived, as we shall see later when talking about data-driven programming.

With two possible banking functions, INTEREST and NEWBALANCE, it is possible to define TRANSACT, a function whose first argument is intended to be the name of a function:

```
(DEFUN TRANSACT (FUNCTION-NAME BALANCE)
       (FUNCALL FUNCTION-NAME BALANCE))
```

This leads to the following examples:

```
(TRANSACT 'INTEREST 100.0)
  10.0

(TRANSACT 'NEWBALANCE 100.0)
  110.0
```

Summary

- DEFUN enables a user to make new functions.
- A predicate is a function that returns T or NIL.
- AND, OR, and NOT are used to do Logic.
- Predicates help COND select a value among alternatives.
- COND enables DEFUN to do more.
- Variables may be free or bound.

- LISP is neither call-by-reference nor call-by-value
- Free variable values are determined dynamically, not lexically.
- Function names can be used as arguments.

4
RECURSION AND ITERATION

The first purpose of this chapter is to understand how a function can use itself in its own definition. This proves useful because it enables a procedure to solve a problem by simplifying it slightly and handing the simplified problem off to one or more exact copies of itself, much like people do when working in a mature bureaucracy. This is called recursion. It is one way of doing something repeatedly.

The second purpose is to see how a function can do something repeatedly without using recursion. This is called iteration. Iteration should be done according to rules that ordinarily improve program readability.

Programming Requires Control Structure Selection

A *control structure* is a general scheme by which a program can go about getting things done. Recursion and iteration are examples of control structures. In general, the choice of a control structure should be determined by the problem under consideration. Sometimes a mathematical function is specified in a way that suggests how it should be computed. Sometimes the way a problem is represented determines the proper thing to do. Sometimes either recursion or iteration will do equally well.

Recursion Allows Programs to Use Themselves

To calculate the n-th power of some number, m, it is sufficient to do the job for the n—1-th power because this result multiplied by m is the desired result for the n-th power:

$$m^n = m * m^{n-1} \quad \text{for} \quad n > 0$$
$$m^0 = 1$$

Definitions that describe a process partly in terms of fresh starts on simpler arguments are said to be *recursive.* The power function, as just specified, suggests a recursive definition. Let us look at a recursive definition for POWER expressed in LISP. Since it cannot handle fractional exponents, it is somewhat weaker than the EXPT function supplied by LISP itself:

```
(DEFUN POWER (M N)
       (COND ((ZEROP N) 1)                          ;n = 0 ?
             (T (TIMES M (POWER M (SUB1 N))))))     ;Recurse.
```

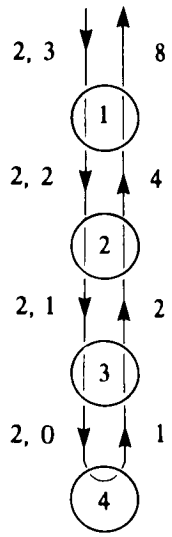


Figure 4-1: A simulation often helps illuminate the strategy of a recursive program. Here each copy of the POWER function reduces the value of N by one and passes it on until N is reduced to zero. The arrows show how control moves from one numbered application of POWER to another. Arguments are shown on the downward pointing arrows; values returned, on the upward.

Using figure 4-1, we can easily follow the history of M and N as LISP evaluates (POWER 2 3). The conventional device used to show how the recursion works is an inverted tree where the branches under a node represent fresh entries to the

function as required by the computation. Each entry is listed with its order in the sequence of entries to the function. Downward portions of the flow of control show the argument or arguments carried down while upward portions indicate the values returned.

In the POWER example, each place in the tree sprouts only one new branch because each copy of POWER can create only one new copy. Such recursive programs are generally easy to convert into programs that do not call themselves and therefore are not recursive.

The Fibonacci function provides a different example. The Fibonacci function is defined as follows:

$$f(n) = f(n-1) + f(n-2) \quad \text{for} \quad n > 1$$
$$f(0) = 1$$
$$f(1) = 1$$

Putting this into LISP, we have this:

```
(DEFUN FIBONACCI (N)
       (COND ((ZEROP N) 1)                                  ;n = 0
             ((EQUAL N 1) 1)                                ;n = 1
             (T (PLUS (FIBONACCI (DIFFERENCE N 1))
                      (FIBONACCI (DIFFERENCE N 2))))))
```

Figure 4-2 shows FIBONACCI computing a result when given an argument of 4.

Computing Fibonacci numbers this way is more interesting than computing powers because each copy of FIBONACCI can create two new copies, not just one. Computing Fibonacci numbers is an example of a problem that would be naturally suited to recursion but for the fact that there happens to be a more efficient way to do the computation.

Having seen recursion at work on two simple numerical problems, we next pretend that LISP does not come equipped with the function MEMBER already. We define MEMBER to be a function that tests to see if its first argument is an element of its second.

```
(DEFUN MEMBER (ITEM S)
       (COND ((NULL S) NIL)                 ;List empty?
             ((EQUAL ITEM (CAR S)) S)       ;First element wins?
             (T (MEMBER ITEM (CDR S)))))    ;Recurse.
```

Evidently, MEMBER considers two situations to be basic: either the second argument, S, is an empty list, resulting in a value of NIL for MEMBER; or the first element of S is equal to the first argument, ITEM. If neither of the basic situations is in effect, MEMBER gives up and hands a slightly simplified problem to a copy of itself. The new copy has a list to work with that is one element shorter.

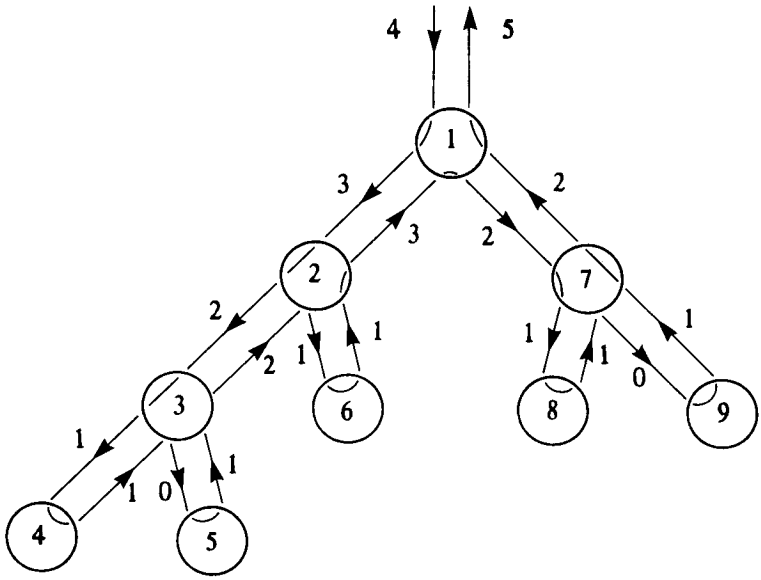


Figure 4-2: A simulation of FIBONACCI. Note that recursion activates two copies each time the argument proves to be hard to handle.

Figure 4-3 shows MEMBER at work on the following problem:

```
(MEMBER 'COMPUTERS '(FAST COMPUTERS ARE NICE))
  (COMPUTERS ARE NICE)
```

Note that presenting the LISP definition of MEMBER is a way of describing what MEMBER does. Describing MEMBER by presenting a definition that uses only more primitive LISP functions is a useful alternative to describing it in English. This amounts to defining part of LISP using LISP itself. In Chapter 23, this idea will be developed to a seemingly impossible extreme, with the whole of LISP evaluation incestuously defined in terms of a LISP program.

Note also that MEMBER delivers a simplified problem to a copy of itself, except in those situations considered basic, but it never does anything to a result produced by a copy. Results are just passed on. Such functions are called tail recursive.

- A function is *tail recursive* if values are passed upward without alteration as the recursion unwinds.

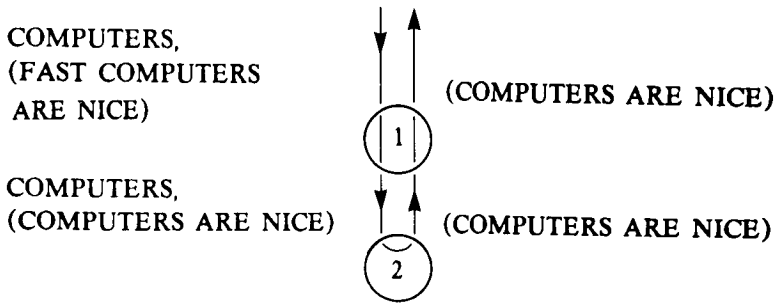


Figure 4-3: A simulation of MEMBER at work on the atom COMPUTERS and the list (FAST COMPUTERS ARE NICE).

POWER and FIBONACCI are not tail recursive, nor is COUNTATOMS, the next example. COUNTATOMS counts the atoms in some given s-expression:

```
(DEFUN COUNTATOMS (S)
       (COND ((NULL S) 0)                          ;List empty?
             ((ATOM S) 1)                          ;Atom rather than a list?
             (T (PLUS (COUNTATOMS (CAR S))         ;T forces recursion.
                      (COUNTATOMS (CDR S))))))
```

The first line announces that the function COUNTATOMS of one argument, S, is about to be defined. The first two clauses of the COND enumerate the very simplest cases, returning 0 for empty lists and 1 for atoms. If COUNTATOMS gets NIL as its argument, then it returns 0 since NIL is equivalent to () and the empty-list test is done before the atom test.

The third clause handles other situations by converting big problems into smaller ones. Lists are broken up using CAR and CDR and COUNTATOMS is applied to both resulting fragments. Since every atom in the list is either in the CAR or the CDR, every atom gets counted. The PLUS combines the results. Eventually, after perhaps many, many applications of itself, COUNTATOMS reduces the hard cases to something that either the first or the second COND clause can handle.

At this point it is helpful to see how COUNTATOMS can take an s-expression apart and reduce it successively to the simple cases. As shown in figure 4-4, the particular expression whose atoms are to be counted is (TIMES X (SQRT 4)). Note that the data expression itself is in the form of an expression one might think of executing for a value. Here then is an example of a program, COUNTATOMS, examining some other piece of program and performing a computation on it.

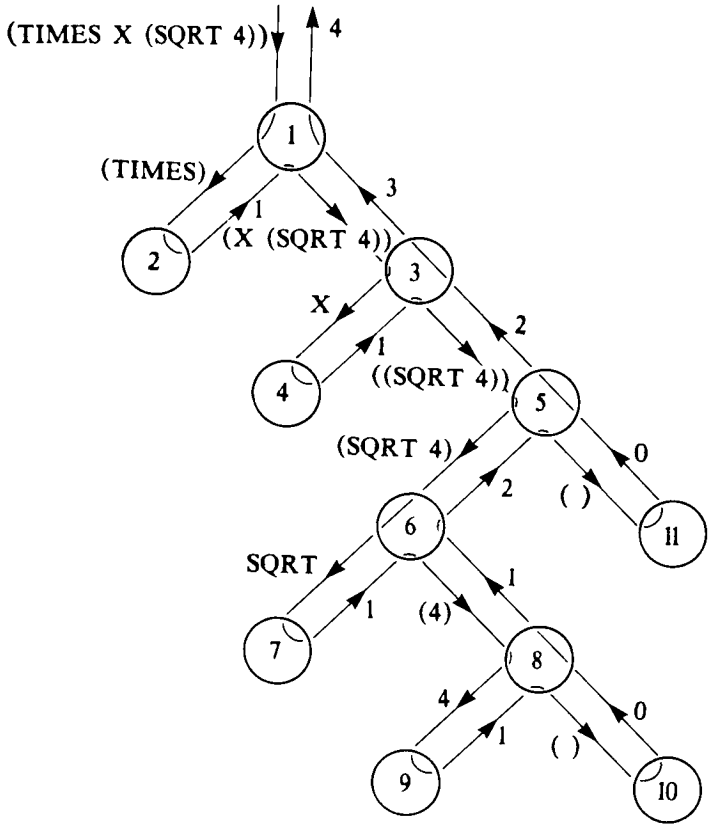


Figure 4-4: Again a simulation helps illuminate the strategy of a recursive program. Here the s-expression (TIMES X (SQRT 4)) is broken up into its constituent pieces by the function COUNTATOMS. The arrows show how control moves from one numbered application of the function to another. Arguments are shown on the downward pointing arrows; values returned, on the upward.

At each stage the argument to COUNTATOMS gets broken into two smaller pieces. Once the answers for both pieces are at hand, PLUS adds the results together and returns the value to a higher-level place further up the tree.

Note that the T in COUNTATOMS insures that the last clause will be triggered if the others are not. A T is often seen in the last clause of a COND where it clearly establishes that the evaluation will not run off the end. The T is not really necessary, however, since the same values would result were it left out.

```
(DEFUN COUNTATOMS (S)
       (COND ((NULL S) 0)                          ;List empty?
             ((ATOM S) 1)                          ;Atom rather than a list?
             ((PLUS (COUNTATOMS (CAR S))           ;No T, first is also last
                    (COUNTATOMS (CDR S))))))
```

This works because COND triggers on anything but NIL, not just T. PLUS can never produce a NIL, and the clause therefore triggers just as if the T were there. Since the clause has only one element, that element is both first and last and provides both the test and the value returned.

Using a T is better programming practice because it clearly signals the fact that the programmer expects the last clause to be used when all else fails. Using a T clearly indicates that there can be no falling through the COND completely with the default value of NIL.

Problems

Problem 4-1: Describe the evident purpose of the following function:

```
(DEFUN MYSTERY (S)
       (COND ((NULL S) 1)
             ((ATOM S) 0)
             (T (MAX (ADD1 (MYSTERY (CAR S)))
                     (MYSTERY (CDR S))))))
```

Problem 4-2: Describe the evident purpose of the following function:

```
(DEFUN STRANGE (L)
       (COND ((NULL L) NIL)
             ((ATOM L) L)
             (T (CONS (STRANGE (CAR L))
                      (STRANGE (CDR L))))))
```

Problem 4-3: Define SQUASH, a function that takes an s-expression as its argument and returns a nonnested list of all atoms found in the s-expression. Here is an example:

```
(SQUASH '(A (A (A (A B))) (((A B) B) B) B))
  (A A A A B A B B B B)
```

Problem 4-4: The version of FIBONACCI given in the text is the obvious, but wasteful implementation. Many computations are repeated. Write a version which does not have this flaw. It may help to have an auxiliary function.

Problems about Sets and Binary Trees

The next group of problems involves functions that work with sets. A *set* is a collection of elements, each of which is said to be a member of the set. A set can be represented as a list, with each element of the set represented by an atom. Each atom occurs only once in the list, and no significance is attached to the order of the atoms in the list.

Problem 4-5: Define UNION. The union of two sets is a set containing all the elements that are in either of the two sets.

Problem 4-6: Define INTERSECTION. The intersection of two sets is a set containing only the elements that are in both of the two sets.

Problem 4-7: Define LDIFFERENCE. The difference of two sets, the IN set and the OUT set, is what remains of the set IN after all elements that are also elements of set OUT are removed.

Problem 4-8: Define INTERSECTP, a predicate which tests whether two sets have any elements in common. It is to return NIL if the two sets are disjoint.

Problem 4-9: Define SAMESETP, a predicate which tests whether two sets contain the same elements. Note that the two lists representing the sets may be arranged in different orders.

A *binary tree* can be defined recursively as either a leaf, or a node with two attached binary trees. Such a binary tree can be represented using atoms for the leaves and three-element lists for the nodes. The first element of each list is an atom representing a node, while the other two elements are the sub-trees attached to the node. Thus

```
(N-1 (N-2 L-A L-B) (N-3 (N-4 L-C L-D) (N-5 L-E (N-6 L-F L-G))))
```

is the representation of a particular binary tree with six nodes (N-1 to N-6) and seven leaves (L-A to L-G). The following group of problems make use of this representation.

Problem 4-10: A mobile is a form of abstract sculpture consisting of parts that move. Usually it contains objects suspended in mid-air by fine wires hanging from horizontal beams. We can define a particularly simple type of mobile recursively as either a suspended object, or a beam with a sub-mobile hanging from each end. If we assume that each beam is suspended from its midpoint, we can represent such a mobile as a binary tree. Single suspended objects are represented by numbers equal to their weight, while more complicated mobiles can be represented by a three-element list. The first element is a number equal to the weight of the

beam, while the other two elements represent sub-mobiles attached at the two ends of the beam.

A mobile should be balanced. This means that the two mobiles suspended from opposite ends of each beam must be equal in weight. Define MOBILEP, a function which determines whether a mobile is balanced. It returns NIL if it is not, and its total weight if it is. So for example:

```
(MOBILEP '(6 (4 (2 1 1) 4) (2 5 (1 2 2)))))
  30.
```

Problem 4-11: Binary trees can be used to represent arithmetic expressions, as for example:

```
(* (+ A B) (- C (/ D E)))
```

One can write a compiler, or program for translating such an arithmetic expression into the machine language of some computer, using LISP. Suppose for example that the target machine has a set of sequentially numbered registers which can hold temporary results. The machine also has a MOVE instruction for getting values into these registers, and ADD, SUB, MUL and DIV instructions for arithmetically combining values in two registers. The above expression, for example, could be translated into the following:

```
((MOVE 1 A)
 (MOVE 2 B)
 (ADD 1 2)
 (MOVE 2 C)
 (MOVE 3 D)
 (MOVE 4 E)
 (DIV 3 4)
 (SUB 2 3)
 (MUL 1 2))
```

The result is left in register number one. Define COMPILE, which performs this translation.

Problem 4-12: Let the "weight" of a binary tree equal one when it is just a leaf. If the binary tree is not a leaf, its weight is the larger of the weights of the two sub-trees if these are not equal. If the weights of the two sub-trees are equal, the weight of the tree is one larger than the weight of either sub-tree. The tree representing the arithmetic expression shown in the previous problem, for example, has weight three. Define WEIGHT, a function which computes the weight of a tree.

Problem 4-13: The number of registers used to compute an arithmetic function depends on whether one computes the left or the right sub-trees first. The function WEIGHT, defined in the previous problem, actually determines the minimum number of registers required to compute a particular arithmetic expression. It is clear that our compiler is not optimal, since it used four registers for a tree with weight three. Assume that another instruction, COPY, is available for moving the contents of one register into another. Improve COMPILE, using WEIGHT, to minimize the number of registers used.

Problems about "C" Curves and Dragon Curves

William Gosper first drew our attention to certain recursively defined drawings. Consider the following recursive program:

```
(DEFUN C-CURVE (LENGTH ANGLE)
       (COND ((LESSP LENGTH MIN-LENGTH) (PLOT-LINE LENGTH ANGLE))
             (T (C-CURVE (QUOTIENT LENGTH (SQRT 2.0))
                         (PLUS ANGLE (QUOTIENT PI 4.0)))
                (C-CURVE (QUOTIENT LENGTH (SQRT 2.0))
                         (DIFFERENCE ANGLE (QUOTIENT PI 4.0))))))
```

Assume that (PLOT-LINE LENGTH ANGLE) plots a straight line of length LENGTH at an angle ANGLE with respect to some standard reference axis. The line is drawn from wherever the last line ended. Also assume that the value of the free variable PI is π. It may help to simulate the result with paper and pencil for cases where the recursion is only two or three layers deep. Compare the results with figure 4-5.

Problem 4-14: Write the procedure PLOT-LINE using LINE, a procedure which takes four arguments X-START, Y-START, X-END and Y-END, specifying the coordinates of the end points of a line to be drawn. Use the free variables X-OLD and Y-OLD to remember where the previous line ended. Assume that the functions SIN and COS calculate the sine and cosine of their single arguments.

Problem 4-15: Lines end up being drawn in only one of eight directions.(Actually only four for given values of LENGTH and MIN-LENGTH.) Write PSEUDO-SIN which takes an integer and produces the result previously offered up by SIN. The corresponding angle is $\pi/4$ times the integer.

Problem 4-16: One can view the recursive step in C-CURVE as one in which a straight line is replaced by a pair of shorter straight lines at right angles to each other, connecting the end points of the original line. There are two ways of doing this, depending on which side of the original line one decides to place the elbow so formed. In C-CURVE the shorter lines are consistently placed on one side of the

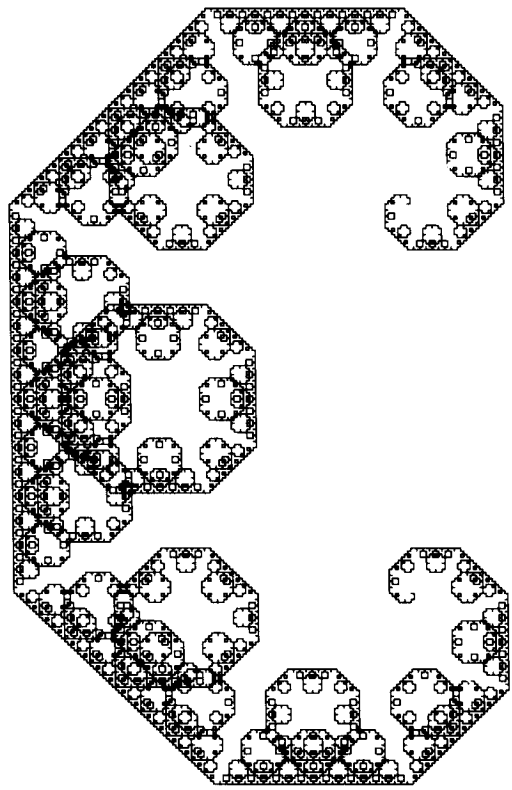


Figure 4-5: The "C" curve, a recursively defined drawing.

line being replaced. Write DRAGON-CURVE, where the first and second recursive calls place the elbows on opposite sides of their respective lines. The drawing produced by such a program is shown in figure 4-6.

Problems about Rewriting Logical Expressions

When working with digital hardware, people often use more than the minimum number of logic element types than are absolutely needed. This is convenient because it is tedious to work in terms of a minimal set. At the same time, only a

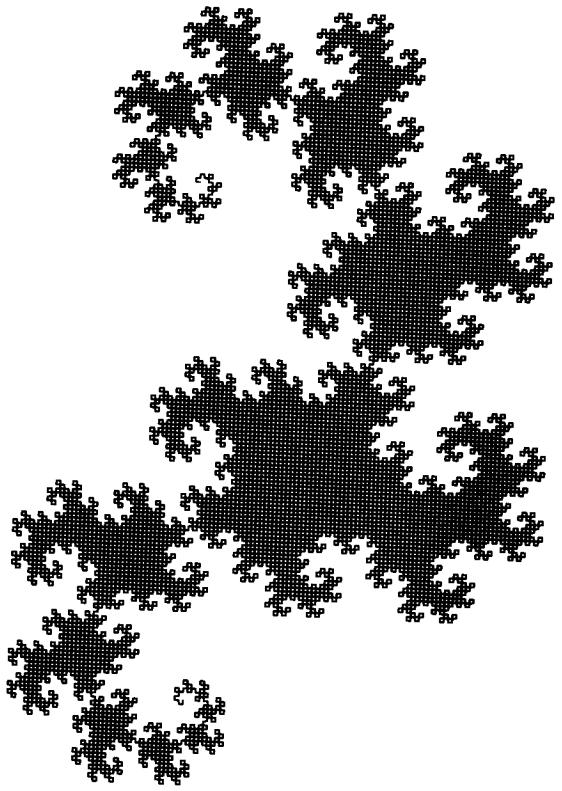


Figure 4-6: The "Dragon" curve, another recursively defined drawing.

minimal set may be available for actually building electronic circuits. There is then a need for automatic translation from the form preferred by the designer to that required when actually building something. In particular, designers may prefer to think in terms of logical operations like NOT, AND, OR, and XOR, while the basic hardware modules may be NAND gates.

The function REWRITE will translate from one form to the other by rewriting the logical operation at the head of the expression handed to it. The arguments to this logical operator are translated using recursive calls to REWRITE. Well-known identities between logical expressions are used in the process.

```
(DEFUN REWRITE (L)
       (COND ((ATOM L) L)
             ((EQUAL (CAR L) 'NOT)
              (LIST 'NAND
                    (REWRITE (CADR L)) T))
             ((EQUAL (CAR L) 'AND)
              (LIST 'NAND
                    (LIST 'NAND
                    (REWRITE (CADR L))
                    (REWRITE (CADDR L)))
                    T))
             ((EQUAL (CAR L) 'OR)
              (LIST 'NAND
                    (LIST 'NAND (REWRITE (CADR L)) T)
                    (LIST 'NAND (REWRITE (CADDR L)) T)))
             ((EQUAL (CAR L) 'XOR)
              (LIST 'NAND
                    (LIST 'NAND
                          (LIST 'NAND
                                (LIST 'NAND (REWRITE (CADR L)) T)
                                (LIST 'NAND (REWRITE (CADDR L)) T))
                          (LIST  'NAND
                                 (REWRITE (CADR L))
                                 (REWRITE (CADDR L))))
                    T))
             (T (LIST 'ERROR L))))
```

Problem 4-17: The above implementation of REWRITE is sometimes wasteful. In certain cases, composite operands may be rewritten several times. Develop a new version of REWRITE that avoids this by applying REWRITE recursively to lists constructed using NAND and untranslated arguments of the logical operations.

Dealing with Lists often Calls for Iteration using MAPCAR

A somewhat more elegant way to define programs like COUNTATOMS is through the functions MAPCAR and APPLY, two new functions that are very useful and very special in the way they handle their arguments.

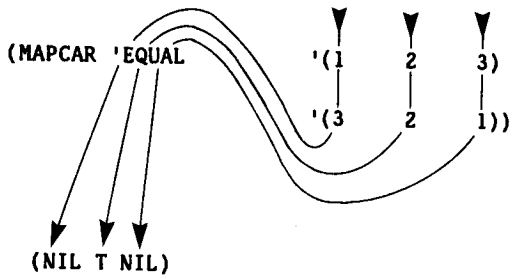
Iterate is a technical term meaning to repeat. MAPCAR can be used to iterate when the same function is to be performed over and over again on a whole list of things. Suppose, for example, that it is useful to add one to each number in a list of numbers. Then from (1 2 3) we would want to get (2 3 4).

To accomplish such transformations with MAPCAR, one supplies the name of the function together with a list of things to be handed one after the other to the function.

```
(MAPCAR 'ADD1                    ;Function to work with.
        '(1 2 3))                ;Arguments to be fed to the function.
  (2 3 4)
```

The MAPCAR is said to cause iteration of ADD1 since the MAPCAR causes ADD1 to be used over and over again.

There is no restriction to functions of one variable, but if the function is a function of more than one variable, there must be a corresponding number of lists of things to feed the function. As shown in the following example, MAPCAR works like an assembly machine, taking one element from each list of arguments and assembling them together for the function:



Consider now a common error that shows why the function APPLY is necessary. Suppose we want to add up a list of numbers, L.

```
(SETQ L '(4 7 2))
  (4 7 2)
```

Do we want to evaluate (PLUS L)? No! That would be wrong. PLUS expects arguments that are numbers. But here PLUS is given one argument that is a list of numbers rather than an actual number. PLUS does not know what to do with the unexpected argument. It is as if we tried to evaluate (PLUS '(4 7 2)) instead of (PLUS 4 7 2).

To make PLUS work, we must use APPLY, which takes two arguments, a function name and a list, and arranges to have the function act on the elements in the list as if they appeared as proper arguments. Thus (APPLY 'PLUS L) is a special case of the general form:

```
(APPLY <function description>
       <list of arguments>)
```

Now using APPLY and MAPCAR we can work up a more elegant way of counting atoms:

```
(DEFUN COUNTATOMS (S)
       (COND ((NULL S) 0)
             ((ATOM S) 1)
             (T (APPLY 'PLUS (MAPCAR 'COUNTATOMS S)))))
```

As suggested by the first two COND clauses, simple cases are handled as before. And once again, the objective is to reduce the more complex expressions to the simple ones. Only now MAPCAR is used to simplify s-expressions rather than CAR and CDR. This version of COUNTATOMS takes advantage of MAPCAR's talent in going after every element of a list with a specified function, in this case a recursive application of COUNTATOMS itself.

Now, assuming for a moment that the function works, we see that the MAPCAR comes back with a list of numbers that must be added together. This is why APPLY appears. PLUS wants numbers, not a list of numbers. APPLY does the appropriate interfacing and hands the list of numbers to PLUS as if each element in the list were itself an argument to PLUS.

Simulating this version of COUNTATOMS is easier than simulating the other one since the recursion is less deep and complicated. See figure 4-7.

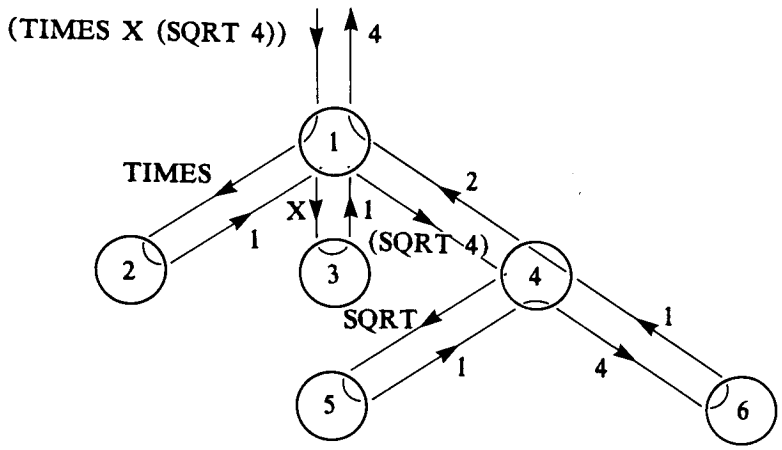


Figure 4-7: A version of COUNTATOMS using MAPCAR instead of CARs and CDRs exhibits less recursion. There are only six entries on three levels rather than eleven on six.

Now it is easy to modify COUNTATOMS to do other things. For example, to determine the depth of an s-expression, the following simple modifications to COUNTATOMS are all that is necessary: change the name to DEPTH, change the empty list and atom results, change PLUS to MAX, and insert ADD1:

```
(DEFUN DEPTH (S)
       (COND ((NULL S) 1)
             ((ATOM S) 0)
             (T (ADD1 (APPLY 'MAX (MAPCAR 'DEPTH S))))))
```

Figure 4-8 shows how DEPTH works on the same expression previously used to illustrate COUNTATOMS. Again, at each branching, we see how MAPCAR splits up an expression into simpler elements to be worked on.

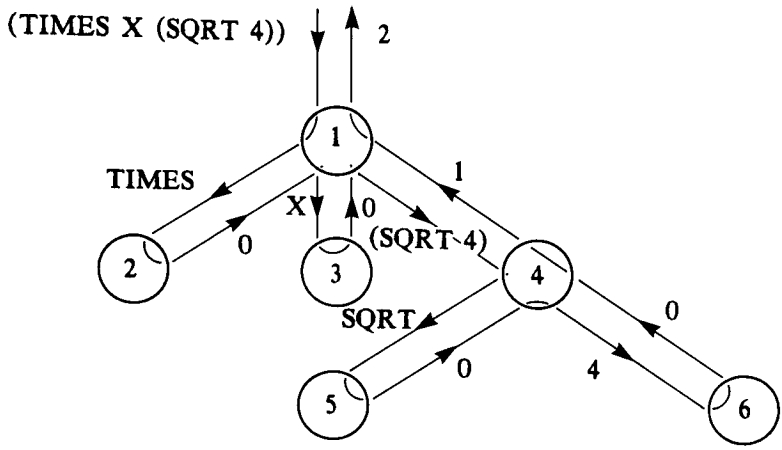


Figure 4-8: The recursion pattern for DEPTH is the same as the one for COUNTATOMS.

The iteration involved in the use of MAPCAR is a special case of iteration since it is restricted to the repeated application of a function to the elements of a list. In the next section we look at PROG, a function that makes it possible to do iteration in general.

Problems

Problem 4-18: Define DYNAMIC-RANGE, a function that takes one argument, a list of numbers, and returns the ratio of the biggest to the smallest.

PROG Creates Variables and Supports Explicit Iteration

PROG is a popular function, particularly with those who do a lot of numerical computation, partly because it creates new variables and partly because it provides one way to write procedures that loop, or said more elegantly, that iterate.

PROG also can be used just to paste together several s-expressions that are to be executed in sequence. This is not done much because DEFUN accepts sequences of s-expressions, not just one. In the old days of early LISP, however, DEFUN and certain other functions were allowed to have only one s-expression. PROG was necessary whenever a sequence was desired.

The syntax for PROG is easier to explain through an immediate example rather than through discussion. We will do the power function, this time specified in a way that suggests iteration rather than recursion:

$$m^n = m * m \quad \ldots \quad m$$

Here, then, is another way to write POWER:

```
(DEFUN POWER (M N)
       (PROG (RESULT EXPONENT)
             (SETQ RESULT 1)                              ;Initialize.
             (SETQ EXPONENT N)                            ;Initialize.
             LOOP                                         ;Start loop.
             (COND ((ZEROP EXPONENT) (RETURN RESULT)))    ;Test.
             (SETQ RESULT (TIMES M RESULT))               ;Reset.
             (SETQ EXPONENT (SUB1 EXPONENT))              ;Reset.
             (GO LOOP)))                                  ;Repeat.
```

Several things must be explained. Keep in mind that the objective is to multiply *n* *m*-s together. This will be accomplished by passing repeatedly through the part of the PROG just after LOOP.

- The arguments to a PROG are mostly s-expressions that are evaluated one after the other. The values are thrown away, so evaluations are only good for side effects. If control runs off the end of a PROG, then NIL is returned, just as with COND.

- The first position in a PROG is always occupied by a list of parameters that are all bound on entering the PROG and restored to old values on exit. If there are no parameters, NIL or () must be in first position. Each parameter is given an initial value of NIL automatically.

- Whenever the function RETURN is reached when evaluating a PROG, the PROG is terminated immediately. The value of the terminated PROG is the value of the argument to the RETURN that stopped the PROG.

- Any atom appearing as an argument to a PROG is considered to be a position marker. These atoms, called *tags*, are not evaluated. They mark places to which control can be transferred by GO functions. (GO <tag>) transfers control to the s-expression following <tag>. A tag can be any atom, by the way; it need not be a symbolic atom.

- PROGs can be nested, like other functions, but it is only possible to go to a tag that is in the same PROG as the GO.

It is clear that the power function works by looping through the tag named LOOP until the variable N is counted down to zero. Each time through the loop, RESULT is changed through multiplication by *m*. The COND tests for the stop condition, $n=0$, and executes a RETURN when the test succeeds. RESULT starts with a value of NIL as all PROG variables do, but is set to 1 before the loop is entered.

Incidentally, it is sometimes useful to have a way of combining several expressions into a sequence as PROG does, but without the parameters and looping possibilities. PROG2, and PROGN do this. The value returned is the value of the second argument in the case of PROG2, and the last in the case of PROGN. Some LISP implementations provide PROG1, which returns the value of the first argument.

PROG-based Iteration should be done with Care

One reason PROG-based iteration can be hard to understand is that the description of a loop can be too spread out. Variable initialization, variable incrementing SETQs, and termination tests with RETURNs can be scattered everywhere. Worse yet, multiple PROG tags and GOs insure that the actual path a program takes may look like a badly tangled up length of string.

Consequently it is useful to have iteration functions with a syntax that requires all of the loop description to be stated at the beginning, before the body of the loop and not interdigitated with it. PROG begins with a variable list. DO-type functions begin with a variable list, initialization information, descriptions of how to increment the variables each time around the loop, and a loop termination test. There are no tags and no GOs. It can be shown that anything using PROG loops can be rewritten using a DO-type loop.

Most modern LISPs have some sort of DO function built in. If there is no DO, it is easy enough to create one, as we will do in a later chapter. But even without a DO, it is possible to use PROGs in a way that closely adheres to the do-not-misuse-GO philosophy:

- Initialize PROG variables immediately after the variable list.

- Then place the PROG's single tag. Let it have some obvious name like LOOP.
- Then place the PROG's single termination-testing COND. It will have only one clause. That clause will have a RETURN at the end.
- Then place the body of the loop next.
- Finally, at the end of the loop, place variable incrementing instructions and conclude with (GO LOOP).

This compresses loop description into two places, one at the beginning of the loop and one at the end, which is as close as PROG can be to the ideal of having everything in one place.

Note that the function POWER, given above, illustrates the suggested semistructuring of PROG loops. Curiously, it has no body. Everything is handled in the variable initialization and incrementing part of the PROG. In fact, this lack of a body is rather common.

Problems and their Representation Determine Proper Control Structure

We can now state some rough guidelines for determining how to select a way to cause repetition from the repertoire of control structures introduced so far.

- The definition of a mathematical function may suggest an appropriate control structure. If so, use it.
- If solving a problem involves diving into list structure, recursion probably makes sense.
- If solving a problem involves doing something to every element in a list, iteration using MAPCAR is usually the right thing.

Keep in mind that these rules are only rough guidelines to be augmented as experience increases. In particular, we will soon add the following:

- When working with arrays, try iteration using PROGs (or DOs if DO is available).

Problems

Problem 4-19: Define FACTORIAL, using the PROG version of POWER as a model. Factorial of n is to be one if n is zero and n times the factorial of n-1 otherwise.

Problem 4-20: Some elements of a set may be equivalent. Such equivalences can be expressed using a list of pairs of equivalent elements. If A is equivalent to B, then B is equivalent to A. Further, if A is equivalent to B, and B is equivalent to C, then A is also equivalent to C. *Equivalence classes* are subsets, all elements of which are equivalent. In the case just described, the set (A B C) forms an equivalence class. Define COALESCE, a function which takes a set of pairwise equivalences and returns a set of equivalence classes.

A typical application of COALESCE is the following:

```
(COALESCE '((A E) (Z F) (M B) (P K)
            (E I) (F S) (B D) (T P)
            (I O) (S V) (D G) (K P)
            (O U) (V Z) (G M) (P T)))
  ((A E I O U) (F S V Z) (B D G M) (K P T))
```

It may be helpful to employ UNION and INTERSECTP, defined in earlier examples in this chapter. Also, the solution may make use of both recursion and iteration.

Summary

- Programming requires control structure selection.
- Recursion allows programs to use themselves.
- Dealing with lists often calls for iteration using MAPCAR.
- PROG creates variables and supports explicit iteration.
- PROG-based iteration should be done with care.
- Problems and their representation determine proper control structure.

References

The conversion of recursive programs into iterative ones is discussed by Auslander and Strong [1976] as well as Darlington and Burstall [1976], and Burstall and Darlington [1977]. Burge [1975] discusses recursive programming at length.

Algorithms involving sets are analysed by Aho, Hopcroft, and Ullman [1974, pg. 124]. The equivalence class problem can be solved using a transitive closure algorithm found in Aho, Hopcroft, and Ullman [1974, pg. 199]. Binary trees are discussed in the same book [1974, pg. 113].

Knuth [1974] discusses structured programming with GO-TO statements.

5
PROPERTIES A-LISTS ARRAYS AND ACCESS FUNCTIONS

So far we only know how to give a symbolic atom a value and to get the value back. The purpose of this chapter is to introduce new methods for remembering and recalling things. Our first objective is to generalize the notion of value, making it possible to give an atom many different values, each of which is associated with an explicitly named property. Then we will work on certain lists called association lists, a-lists for short. One of the features of a-lists is that they enable groups of atom values to be remembered, suppressed, or restored together, all at once. And finally, we will race past the use of arrays, showing that it is possible to store and retrieve information by row and column numbers.

Properties and Property Values Generalize the Notion of Atom and Value

Symbolic atoms can have *properties.* Property names and property values are left to the imagination of the user. An example is the FATHER property. The value of the FATHER property of the atom ROBERT could be JAMES, for example. The PARENTS property of ROBERT could be a list of two atoms, each naming one of ROBERT's parents. In general, the value of a property can be any s-expression.

Note, incidentally, that the word value is being used in two senses: first, we talk of the value of some particular property of an atom; and second, we talk of the value of an atom.

PUTPROP and GET are the Masters of Property Lists

To retrieve property values, GET is used:

```
(GET 'ROBERT 'FATHER)
  JAMES

(GET 'ROBERT 'SURNAME)
  NIL
```

To place or replace a property value, the complementary function, PUTPROP, does the job:

```
(PUTPROP 'ROBERT 'WINSTON 'SURNAME)
  WINSTON

(PUTPROP 'JAMES '(ROBERT ALBERT) 'SONS)
  (ROBERT ALBERT)
```

Note that the value returned by PUTPROP is the same as the value it attaches to the atom given as the first argument. It is also good to know that GET returns NIL if no property with the given name exists yet. This suggests first of all that NIL is a poor choice for the value of a property, since the result returned by GET cannot be distinguished from what GET returns when it does not find the property asked for. It also means that as far as GET is concerned, properties can be removed this way:

```
(PUTPROP <atom name> NIL <property name>)
```

It is clearer to use the function REMPROP with just the atom name and property name as arguments.

```
(REMPROP <atom name> <property name>)
```

DEFPROP is a sort of companion of PUTPROP. It uses the arguments the same way but does not evaluate them. Also, DEFPROP returns its first argument, rather than its second, mostly for obscure historical reasons. Thus the following have equivalent side effects:

```
(PUTPROP 'PATRICK 'WINSTON 'SURNAME)
  WINSTON

(DEFPROP PATRICK WINSTON SURNAME)
  PATRICK
```

After either we get,

```
(GET 'PATRICK 'SURNAME)
  WINSTON
```

Problems

Problem 5-1: Suppose each city in a network of cities and highways is represented by an atom. Further suppose that each city is to have a property named NEIGHBORS. The value of the NEIGHBORS property is to be a list of the other cities for which there is a direct highway connection.

Define a function CONNECT that takes two cities as arguments and puts each on the property list of the other under the NEIGHBORS property. Write CONNECT such that nothing changes if a connection is already in place.

Problem 5-2: Assume that if a person's father is known, the father's name is given as the value of the FATHER property. Define GRANDFATHER, a function that returns the name of a person's paternal grandfather, if known, or NIL otherwise.

Problem 5-3: Define ADAM, a function that returns the most distant male ancestor of a person through the paternal line. If no male ancestor is known, the function is to return the person given as its argument.

Problem 5-4: Define ANCESTORS, a function that returns a list consisting of the person given as its argument together with all known ancestors of the person. It is to work through both the FATHER and MOTHER properties. Assume related people do not have children.

Problem 5-5: Suppose X and Y are properties used to specify the location of cities relative to some reference point. Assuming a flat earth, write a function that calculates the distance between two cities. Remember that SQRT calculates square roots.

ASSOC Retrieves Pairs from Association Lists

An *association list*, a-list for short, is a list of pairs. The following makes TODAY and YESTERDAY into a-lists recording medical facts:

```
(SETQ YESTERDAY '((TEMPERATURE 103) (PRESSURE (120 60)) (PULSE 72)))
  ((TEMPERATURE 103) (PRESSURE (120 60)) (PULSE 72))

(SETQ TODAY '((TEMPERATURE 100) (PRESSURE (120 60)) (PULSE 72)))
  ((TEMPERATURE 100) (PRESSURE (120 60)) (PULSE 72))
```

ASSOC is a function of two arguments. The first argument to ASSOC is often called the *key*. ASSOC looks for its key in the a-list supplied as the second argument. ASSOC moves down the a-list until it finds a list element whose CAR is equal to the key. The value of the ASSOC is the entire element so discovered, key and all, or NIL if the key is never found. These examples illustrate:

```
(SETQ CHART YESTERDAY)
  ((TEMPERATURE 103) (PRESSURE (120 60)) (PULSE 72))

(ASSOC 'TEMPERATURE CHART)
  (TEMPERATURE 103)

(SETQ CHART TODAY)
  ((TEMPERATURE 100) (PRESSURE (120 60)) (PULSE 72))

(ASSOC 'TEMPERATURE CHART)
  (TEMPERATURE 100)

(ASSOC 'COMPLAINTS CHART)
  NIL
```

Problems

Problem 5-6: Write FETCH, a function that takes an atom and an a-list. If the atom is found as the first element of an item on the a-list, then the second item is to be returned. Otherwise FETCH is to return a question mark. (A question mark makes a perfectly good atom.) The following examples illustrate:

```
(SETQ CHART '((TEMPERATURE 100) (PRESSURE (120 60)) (PULSE 72)))
  ((TEMPERATURE 100) (PRESSURE (120 60)) (PULSE 72))

(FETCH 'TEMPERATURE CHART)
  100

(FETCH 'COMPLAINTS CHART)
  ?
```

Problem 5-7: Write LISTKEYS, a function that takes an a-list and returns a list of all the keys in it. Recall that the keys are the things that ASSOC checks its first argument against.

Problem 5-8: Write TREND, a function that takes two a-lists that record temperature, among other things. TREND is to return either IMPROVING or STABLE or SINKING depending on whether the patient's temperature is moving toward normal, at normal, or moving away. Assume the first a-list records the older data. Further assume that there is always an entry for temperature on both a-lists. Use FETCH as defined in a previous problem, if you like.

STORE and ARRAY are used with Arrays

Conceptually, an *array* is a data structure in which information is located in slots, each of which is associated with a particular set of numbers called indices.

Suppose, for illustration, that the measured brightness values at each point of an image are to be stored and retrieved. Any particular point has an x and a y coordinate, both of which are numbers. It is natural to store the brightness values in an array, using the x and y coordinates to specify exactly where the values should go.

If an image has been stored in an array named IMAGE, then (IMAGE 314 271) will retrieve the image point whose x coordinate is 314 and whose y coordinate is 271. Note that the array name is used just as if it were a function. Which index corresponds to which coordinate is, of course, quite arbitrary. In the case of matrices, it is customary to let the first index be the row number and the second the column number.

For arrays, unlike for lists, the size must be announced in advance of use. This is done using the function ARRAY:

```
(ARRAY IMAGE T 1024 1024)
  IMAGE
```

This states that IMAGE is an array, that there are two indices, and that both indices range over 1024 values (from 0 to 1023, oddly enough, not 1 to 1024). (The T signifies that the array slots are to be able to hold arbitrary s-expressions. Instead, we could have used the atoms FIXNUM or FLONUM to indicate that the array will be used only for fixed or floating point numbers respectively.)

Once IMAGE has been created, information can be stored in it. This is done as in the following example which puts 88 in the array location specified:

```
(STORE (IMAGE 314 271) 88)
  88
```

A different syntax, something like (STORE IMAGE 314 271 88) might make more sense, but the actual syntax has advantages that have to do with how arrays are implemented.

Note that while the array in this example happens to hold only numbers, LISP array slots may hold arbitrary s-expressions.

Problems

Problem 5-9: A matrix can be represented as a list, with each row a sublist. Define STUFF-MATRIX, a procedure which makes an array called MATRIX, of appropriate size, and then fills it in from a matrix represented as a list of lists. Here is a test case for STUFF-MATRIX:

```
(STUFF-MATRIX  '((0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
                 (0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0)
                 (0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0)
                 (0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0)
                 (0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0)
                 (0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0)
                 (0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0)
                 (0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0)
                 (0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0)
                 (0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0)
                 (0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0)
                 (0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0)
                 (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)))
```

Access Functions Simplify Data Interactions

Access functions are functions designed to simplify the chore of retrieving and changing desired data. Access functions for retrieval are built out of LISP primitives like GET, ASSOC, CAR, and CDR. It is often good to get at data by way of access functions, rather than using the LISP primitives directly, for these two reasons:

- Access functions make programs more transparent. Given mnemonic names, access functions are easier to find and understand than the equivalent combinations of primitives.

- Access functions make programs easier to modify. Given some need to change the way data is arranged, it is easier to change the definitions of a few access functions than to find and change each place that data is used.

Suppose, for example, that someone has decided that each person's social security number is to be the third item on a list to be found under the NUMBERS key in an a-list to be found under the IDENTIFIERS property of the atom of the same name. To be sure, whenever the social security number of the person whose name is the value EMPLOYEE is needed, it is possible to get it using LISP primitives:

```
  .
  .
  .
(CADDR (ASSOC 'NUMBERS (GET 'IDENTIFIERS EMPLOYEE)))
  .
  .
  .
```

But it is probably better to get away from this by defining an access function:

```
(DEFUN SOCIAL-SECURITY-NUMBER (PERSON)
       (CADDR (ASSOC 'NUMBERS (GET 'IDENTIFIERS PERSON))))
```

Then programs will contain only instances of the access function rather than the possibly confusing combination of a CADDR, ASSOC, and GET:

```
.
.
.
(SOCIAL-SECURITY-NUMBER EMPLOYEE)
.
.
.
```

A set of access functions can become quite elaborate. The so-called frames representation language, discussed in a later chapter, is an example of a set of access functions that is so elaborate that it deserves to be thought of as an embedded language.

Summary

- Properties and property values generalize the notion of atom and value.
- PUTPROP and GET are the masters of property lists.
- ASSOC retrieves pairs from association lists.
- STORE and ARRAY are used with arrays.
- Access functions simplify data interactions.

6
USING LAMBDA DEFINITIONS

The descriptions of functions are recorded using DEFUN so that they can be recalled later using their name. In this chapter, we will see that it is possible to use a function description in places where ordinarily there would be a function name. This is useful in a variety of ways.

LAMBDA Defines Anonymous Functions

Consider the problem of checking which elements in a list FRUITS are apples. One way would be to start by defining APPLEP:

```
(DEFUN APPLEP (X) (EQUAL X 'APPLE))
```

Then we can write this:

```
(SETQ FRUITS '(ORANGE APPLE APPLE APPLE PEAR GRAPEFRUIT))
  (ORANGE APPLE APPLE APPLE PEAR GRAPEFRUIT)

(MAPCAR 'APPLEP FRUITS)
  (NIL T T T NIL NIL)
```

This is a little painful if there is only one occasion to test for fruits. Passing over a list with a simple function should not require going off somewhere else to define a function. Why not make the programmer's intention more transparent by laying out the function at the spot where it is to be used.

```
(MAPCAR (DEFUN APPLEP (X) (EQUAL X 'APPLE)) FRUITS)
  (NIL T T T NIL NIL)
```

DEFUN works because MAPCAR wants a function name and the value of DEFUN is the name of the function defined. But to avoid the proliferation of useless names, the name can be dropped if it is used only here! In this event, the lack of a function name is signaled by using something different from DEFUN to define the procedure. For historical reasons this new function definition is called a *lambda expression* and the atom LAMBDA appears instead of DEFUN. The correct way to use a local definition would therefore be as in this example:

```
(MAPCAR '(LAMBDA (X) (EQUAL X 'APPLE)) FRUITS)
  (NIL T T T NIL NIL)
```

- No one would actually use DEFUN in a MAPCAR anyway since such a practice would be hopelessly inefficient — setting up a function with the inevitable overhead happens each time the section of program is used. The DEFUN was used here only as a way of introducing lambda-style definitions.

A longer but perhaps more informative name for introducing local definitions would be DEFINE-ANONYMOUS. We will stick to LAMBDA for compatibility, but if confusion ever sets in, a good heuristic for understanding LAMBDA is to translate it mentally to DEFINE-ANONYMOUS.

Note that lambda definitions appearing in a MAPCAR require a quote to suppress evaluation, just as a function name does. (Actually, under some circumstances, it is better to use FUNCTION instead of QUOTE. In particular, if a so-called compiler is used to translate raw LISP programs into a machine oriented form, then FUNCTION tells the compiler to translate the surrounded lambda definition, rather than just take it as an s-expression to be left as is.)

Note also that a lambda definition can go just about anywhere a function name can go. In particular, a lambda-style description can be the first element in a list to be evaluated, even though using a function name is vastly more common. The following are exactly equivalent, given the existing definition of APPLEP:

```
(APPLEP 'APPLE)
  T

((LAMBDA (X) (EQUAL X 'APPLE)) 'APPLE)
  T

(FUNCALL '(LAMBDA (X) (EQUAL X 'APPLE)) 'APPLE)
  T
```

For further illustration, suppose we want to actually count the number of apples on the list FRUITS. There are several methods involving MAPCAR. The first

involves building a new function around APPLEP that returns 1 or 0 instead of T or NIL:

```
(DEFUN APPLEP-1-0 (Y)
       (COND ((APPLEP Y) 1)
             (T 0)))
```

Then it is easy to count the apples by using MAPCAR to run APPLEP-1-0 down the list, adding the results by applying PLUS:

```
(MAPCAR 'APPLEP-1-0 FRUITS)
  (0 1 1 1 0 0)

(APPLY 'PLUS
       (MAPCAR 'APPLEP-1-0 FRUITS))
  3
```

Alternatively, the action now in APPLEP-1-0 can be specified locally, using a lambda-style definition in the MAPCAR:

```
(APPLY 'PLUS
       (MAPCAR '(LAMBDA (Y) (COND ((APPLEP Y) 1)
                                  (T 0)))
               FRUITS))
  3.
```

Indeed, the guts of APPLEP can be brought in too:

```
(APPLY 'PLUS
       (MAPCAR '(LAMBDA (X) (COND ((EQUAL X 'APPLE) 1)
                                  (T 0)))
               FRUITS))
  3.
```

Of course it is easy to bottle all of this up and make a function out of it:

```
(DEFUN COUNTAPPLES (FRUITS)
       (APPLY 'PLUS
              (MAPCAR '(LAMBDA (X) (COND ((EQUAL X 'APPLE) 1)
                                         (T 0)))
                      FRUITS)))

(COUNTAPPLES FRUITS)
  3
```

Incidentally, lambda-style definitions can have any number of s-expressions, just like ordinary definitions. Only the last determines the value. The rest are evaluated for side effects.

LAMBDA is Often Used to Interface Functions to Argument Lists

The next example develops a function that determines how many times atoms with a specified property are found in some given s-expression. Unlike COUNTAPPLES, this new function is expected to work on any s-expression, not just lists of atoms.

The plan is to make a recursive function that can work its way into the s-expression that is given. This preparatory function counts the instances of APPLE:

```
(DEFUN COUNTAPPLES (S)
       (COND ((EQUAL S 'APPLE) 1)          ;It is an apple.
             ((ATOM S) 0)                  ;No, it is not.
             (T (APPLY 'PLUS
                       (MAPCAR 'COUNTAPPLES S)))))
```

Now we attempt to generalize this by adding another variable, making a function that will count any item, not just apples. The new function counts the instances of the first argument appearing in the second argument. Our first attempt fails:

```
(DEFUN COUNT (ITEM S)
       (COND ((EQUAL S ITEM) 1)                ;It is one of them.
             ((ATOM S) 0)                      ;No, it is not.
             (T (APPLY 'PLUS
                       (MAPCAR 'COUNT S)))))   ;Blunder!
```

This is wrong because COUNT is defined as a function of two arguments, ITEM and S. If it does not get two, it is unhappy. But the MAPCAR only has one list of things to channel into COUNT. Hence disaster. There is a fatal mismatch between the parameter list COUNT is defined with and the arguments it is supplied by the MAPCAR.

The solution is to make a new function, using COUNT, that has only one parameter. Lets call this COUNTAUX:

```
(DEFUN COUNTAUX (S) (COUNT ITEM S))

(DEFUN COUNT (ITEM S)                        ;Uses auxiliary function.
       (COND ((EQUAL S ITEM) 1)
             ((ATOM S) 0)
             (T (APPLY 'PLUS (MAPCAR 'COUNTAUX S)))))
```

More likely, however, COUNTAUX would not be defined. Instead a local lambda-style definition would be used:

```
(DEFUN COUNT (ITEM S)                          ;Uses LAMBDA.
       (COND ((EQUAL S ITEM) 1)
             ((ATOM S) 0)
             (T (APPLY 'PLUS (MAPCAR '(LAMBDA (E)
                                        (COUNT ITEM E))
                                     S)))))
```

Here LAMBDA matches COUNT, a function of two variables, to one list of MAPCAR arguments. Note that ITEM is a free variable with respect to the lambda expression. Using 'COUNT in place of the lambda definition is an example of a common error.

Problems

Problem 6-1: Define PRESENTP, a function that determines if a particular atom exists anywhere in an s-expression. It differs from MEMBER because MEMBER only looks for top-level instances. Symbolic-mathematics systems make use of such a function to determine if an s-expression contains a particular variable. Consider these, for example:

```
(SETQ FORMULA '(SQRT (QUOTIENT (PLUS (EXPT X 2) (EXPT Y 2)) 2)))

(PRESENTP 'X FORMULA)
  T

(PRESENTP 'Z FORMULA)
  NIL
```

MAPCAN Facilitates Filtering

While we are at it, this is a good time to explain MAPCAN, a function closely related to MAPCAR. Often MAPCAN can be used to do *filtered accumulations.* To see what this means, suppose there is a list of groceries, and for some bizarre reason, we want to pick out the fruits and make a list out of them. When defined, the function KEEPFRUITS will do it:

```
(KEEPFRUITS '(BROCCOLI MILK APPLE BREAD BUTTER PEAR STEAK))
  (APPLE PEAR)
```

Assume FRUITP is a predicate that checks to see if an atom is a fruit, perhaps by inspecting its FRUIT property. Assume KEEPFRUITS's variable is named GROCERIES. Then it is natural to think of using FRUITP on every element in GROCERIES. MAPCAR can arrange this for us, but the result is not quite what we want:

```
(MAPCAR 'FRUITP GROCERIES)
  (NIL NIL T NIL NIL T NIL)
```

Another idea is to arrange a local lambda definition so that the actual fruits get into the result. We are on the way with this:

```
(MAPCAR '(LAMBDA (E) (COND ((FRUITP E) E) (T NIL)))
        GROCERIES)
  (NIL NIL APPLE NIL NIL PEAR NIL)
```

It is too bad the result is not a bit different, for then APPLY, working with APPEND, could do a nice thing:

```
(APPLY 'APPEND '(NIL NIL (APPLE) NIL NIL (PEAR) NIL))
  (APPLE PEAR)
```

But of course it is easy to modify the lambda definition to produce what APPEND needs:

```
(MAPCAR '(LAMBDA (E) (COND ((FRUITP E) (LIST E))
                           (T NIL)))
        GROCERIES)
  (NIL NIL (APPLE) NIL NIL (PEAR) NIL)
```

The combination of an APPEND together with a MAPCAR will do the job. The function used by MAPCAR must return the desired elements packaged up by LIST. The function returns NIL when applied to other elements.

Now for MAPCAN, finally. MAPCAN acts much as if it were a combination of MAPCAR and APPEND. (Actually, MAPCAN acts as if it were a combination of MAPCAR and NCONC, a function that will be explained later. This makes MAPCAN a possibly dangerous function for novices since the NCONC can cause unexpected things to happen.) Thus these are equivalent for our present purpose:

```
(APPLY 'APPEND
       (MAPCAR '(LAMBDA (E) (COND ((FRUITP E) (LIST E))
                                  (T NIL)))
               GROCERIES))
   (APPLE PEAR)
```

```
(MAPCAN '(LAMBDA (E) (COND ((FRUITP E) (LIST E))
                           (T NIL)))
        GROCERIES)
    (APPLE PEAR)
```

Thus MAPCAN is convenient when it is good to filter a list. Using it, KEEPFRUITS, the function we started out to define, is done like this:

```
(DEFUN KEEPFRUITS (GROCERIES)
       (MAPCAN '(LAMBDA (E) (COND ((FRUITP E) (LIST E))
                                  (T NIL)))
               GROCERIES))
```

Style is a Personal Matter

When there are choices, people will argue. For example, some people prefer alternatives to the MAPCAN-type filtered accumulation done by KEEPFRUITS. Here are some that do the same job, but with the result in reverse order:

```
(DEFUN KEEPFRUITS (GROCERIES)            ;Recursive instead of iterative.
      (COND ((NULL GROCERIES) NIL)
            ((FRUITP (CAR GROCERIES))
             (CONS (CAR GROCERIES) (KEEPFRUITS (CDR GROCERIES))))
            (T (KEEPFRUITS (CDR GROCERIES)))))

(DEFUN KEEPFRUITS (GROCERIES)            ;Uses PROG for iteration.
       (PROG (RESULT)
             LOOP
             (COND ((NULL GROCERIES) (RETURN RESULT))
                   ((FRUITP (CAR GROCERIES))
                    (SETQ RESULT (CONS (CAR GROCERIES) RESULT))))
             (SETQ GROCERIES (CDR GROCERIES))
             (GO LOOP)))

(DEFUN KEEPFRUITS (GROCERIES)            ;Uses DO (described briefly later).
       (DO ((RESULT)
            (POSSIBILITIES GROCERIES (CDR POSSIBILITIES)))
           ((NULL POSSIBILITIES) RESULT)
           (COND ((FRUITP (CAR POSSIBILITIES))
                  (SETQ RESULT (CONS (CAR POSSIBILITIES) RESULT))))))
```

Similarly, there are a variety of ways to hold values temporarily. Suppose, for example, that LISTFRUITS is to return the fruits in a list if there are fewer than ten. If there are more, it is to return TOO-MANY. The obvious definition is

wasteful in that it uses KEEPFRUITS twice:

```
(DEFUN LISTFRUITS (GROCERIES)              ;Uses KEEPFRUITS twice.
       (COND ((GREATERP (LENGTH (KEEPFRUITS GROCERIES)) 10)
              'TOO-MANY)
             (T (KEEPFRUITS GROCERIES))))
```

The inefficiency can be avoided in a number of ways:

```
(DEFUN LISTFRUITS (GROCERIES)              ;Holds on using PROG parameter.
       (PROG (TEMP)
             (SETQ TEMP (KEEPFRUITS GROCERIES))
             (COND ((GREATERP (LENGTH TEMP) 10) (RETURN 'TOO-MANY))
                   (T (RETURN TEMP)))))

(DEFUN LISTFRUITS (GROCERIES)              ;Holds on using LAMBDA parameter.
       ((LAMBDA (TEMP)
                (COND ((GREATERP (LENGTH TEMP) 10) 'TOO-MANY)
                      (T TEMP)))
        (KEEPFRUITS GROCERIES)))

(DEFUN LISTFRUITS (GROCERIES)              ;Uses LET (described briefly later).
       (LET ((TEMP (KEEPFRUITS GROCERIES)))
            (COND ((GREATERP (LENGTH TEMP) 10) 'TOO-MANY)
                  (T TEMP))))
```

In this book, we try to stay flexible, believing that style is often a matter of personal taste and that each style has its rightful place depending on circumstances.

- We have avoided DO and LET in this book because these functions tend to be less standardized than others at the moment. This is unfortunate since many of the examples would be clearer if DO and LET were used.

Summary

- LAMBDA defines anonymous functions.
- LAMBDA is often used to interface functions to argument lists.
- MAPCAN facilitates filtering.
- Style is a personal matter.

7
PRINTING READING AND ATOM MANIPULATION

The purpose of this chapter is to introduce PRINT and READ, functions that help other functions communicate with users. Without PRINT, the only way a user can learn about what a LISP function is doing is to wait for a value to appear. Without READ, the only way a function can get information is through its arguments and free variables. PRINT and READ therefore open the door to much more communication. Happily, both are very simple.

PRINT and READ Facilitate Communication

PRINT evaluates its single argument and prints it on a new line. Thus the following function will print out the squares of the integers until something drops dead:

```
(DEFUN BORE-ME ()
       (PROG (N)
             (SETQ N 0)
             LOOP
             (PRINT (TIMES N N))          ;Print square of integer.
             (SETQ N (ADD1 N))            ;Increment integer.
             (GO LOOP)))                  ;Do it forever.
  BORE-ME

(BORE-ME)
  0
  1
```

```
4
9
16
25
36
.
.
.
```

In BORE-ME, the value returned by PRINT is not used. In fact, the value of an application of PRINT is not very useful in this book's LISP because PRINT happens to return T as its value:

```
(PRINT 'EXAMPLE)
  EXAMPLE
  T
```

When (READ) is encountered, LISP stops and waits for the user to type an s-expression. That s-expression, without evaluation, becomes the value of (READ). Using READ by itself therefore causes total inactivity until the user types something. In this example, the user types EXAMPLE (followed by a space):

```
(READ)EXAMPLE
  EXAMPLE
```

Note that READ prints nothing to indicate that it is waiting, not even a carriage return.

Problems about Stacking Disks on Pins

We now tackle the celebrated Tower-of-Hanoi problem. An ancient myth has it that in some temple in the Far East, time is marked off by monks engaged in the transfer of 64 disks from one of three pins to another. The universe as we know it will end when they are done. The reason we do not have to concern ourselves about the cosmological implications of this is that their progress is kept in check by some clever rules: the monks can move only one disk at a time; the disks all have different diameters; and no disk can ever be placed on top of a smaller one.

The insight leading to the correct sequence of moves comes from realizing that a set of n disks can be transferred from pin A to pin B in these stages: first move the top $(n-1)$ disks from A to the spare pin C; then move the large bottom disk from A to B; and finally, move the $(n-1)$ disks from the spare pin, C, onto pin B. Naturally, moving the $(n-1)$ disks can be done by the same trick, using the third pin (not involved in the transfer) as workspace. By means of recursion, we have postponed the actual work until n equals one. It is possible to verify that in each case the transfer of the single disk constitutes a legal move.

```
(DEFUN TOWER-OF-HANOI (N) (TRANSFER 'A 'B 'C N))          ;N disks on A first.

(DEFUN MOVE-DISK (FROM TO)
       (LIST (LIST 'MOVE 'DISK 'FROM FROM 'TO TO)))      ;Build instruction.

(DEFUN TRANSFER (FROM TO SPARE NUMBER)
       (COND ((EQUAL NUMBER 1) (MOVE-DISK FROM TO))      ;Transfer one disk.
             (T (APPEND (TRANSFER FROM                   ;Move from FROM
                                  SPARE                  ;to SPARE
                                  TO                     ;using TO as space
                                  (SUB1 NUMBER))         ;(n-1) disks.
                        (MOVE-DISK FROM TO)              ;Move lowest disk.
                        (TRANSFER SPARE                  ;Move from SPARE
                                  TO                     ;to TO
                                  FROM                   ;using FROM as space
                                  (SUB1 NUMBER))))))     ;(n-1) disks.
```

This is what the resulting list of instructions looks like for a typical case:

```
(TOWER-OF-HANOI 3)
  ((MOVE DISK FROM A TO B) (MOVE DISK FROM A TO C) (MOVE DISK FROM B TO C)
   (MOVE DISK FROM A TO B) (MOVE DISK FROM C TO A) (MOVE DISK FROM C TO B)
   (MOVE DISK FROM A TO B))
```

Problem 7-1: The above solution may not be very convincing, since everything seems to be done blindly, without checking whether disks are ever placed on smaller disks, or for that matter, whether any disks are left on the pin from which they are supposed to be removed. What is needed is a simulation of the pin transfer steps. Consequently, there is a need to print detailed information about the steps as they are taken.

It would also be more informative if the final instructions included some identification of which disk is being moved. Let us assume that the disks are numbered in order of increasing size. Further, let A, B, and C be lists of numbers representing the stacks of disks currently on each of the three pins. Rewrite MOVE-DISK to include checking on the legality of proposed moves, appropriate modification of the lists A, B, and C, as well as generation of more informative output as suggested by the following:

```
(SETQ A '(1 2 3)  B NIL  C NIL)

(TOWER-OF-HANOI)
```

should print:

```
(MOVE 1 FROM A TO B)
(MOVE 2 FROM A TO C)
(MOVE 1 FROM B TO C)
(MOVE 3 FROM A TO B)
(MOVE 1 FROM C TO A)
(MOVE 2 FROM C TO B)
(MOVE 1 FROM A TO B)
```

Special Conventions Make Weird Atom Names Possible

Sometimes it is useful to have atoms that consist wholly or partly of characters that ordinarily are not allowed in atoms. Spaces and parentheses are examples of such characters. Placing vertical bars around such special characters is the way to do this in our implementation of LISP:

```
(SETQ ATOM1 '|(|)
  |(|

(SETQ ATOM2 '|WEIRD ATOM|)
  |WEIRD ATOM|

(PRINT ATOM1)
  |(|
  T

(PRINT ATOM2)
  |WEIRD ATOM|
  T
```

An older method uses a slash, /, to indicate that the following character is to lose any special properties it might otherwise have had. We could have used '/(for ATOM1 and 'WEIRD/ ATOM for ATOM2 in the above examples. (For output, LISP uses the vertical bar convention, not slashes.)

Atoms can be Broken Apart, Put Together, and Created

It is sometimes useful to take the names of atoms apart or to make new atom names. Later, in another chapter, we will have reason to do this so that patterns read by LISP can be more cosmetically pleasing. In particular, we will use atom-manipulation functions to split up atoms like >RESULT into two new atoms, > and RESULT.

EXPLODE appears in order to break up an atom into a list of its constituent characters, each appearing as a single character atom:

```
(EXPLODE 'ATOM)
  (A T O M)

(EXPLODE 'X)
  (X)
```

IMPLODE performs the complementary operation of running a list of single-character atoms together into a single atom:

```
(IMPLODE '(A B C))
  ABC
```

And if there is a program that wants a fresh atom name without bothering the user to get it, GENSYM should be used. In some sense, GENSYM is an input function that creates new atoms rather than reading them in:

```
(GENSYM)
  G0001

(GENSYM)
  G0002
```

In another chapter, these GENSYMed atoms will be used to build tree structures that record how the functions in a block-world manipulation system call one another to solve problems. The trees will make it possible to answer questions about how things were done, as well as why and when.

Exotic Input/Output Functions Lie Beyond PRINT and READ

It is possible to go through a career of LISP programming without using any printing and reading functions other than PRINT and READ. Eventually, however, most people like to use other functions that enable better-looking, more presentable input and output. It makes no sense to dwell on these other printing and reading functions, however, because they tend to vary from system to system. The following are given mainly to show what is usually available by one name or another.

TERPRI is a function of no arguments that starts a new line. PRIN1 is like PRINT except that PRIN1 does not start a new line. PRINC is like PRIN1 except that vertical bars, if any, are suppressed. Thus:

```
(PRINT ATOM2)
  |WEIRD ATOM|
  T
```

```
(PRIN1 ATOM2)|WEIRD ATOM|
  T

(PRINC ATOM2)WEIRD ATOM
```

Note that since PRIN1 and PRINC do not start a new line, the things they print are on the same line the user types on. READCH reads one character and returns it as an atom:

```
(READCH)X
  X
```

Formatted Printing is Easily Arranged

As an example of what can be done with vertical bars and exotic printing functions, let us consider a program that takes two arguments, a list of atoms to be printed, and a list of columns where printing is to start. This function is named FORTRANPRINT inasmuch as it provides features that are similar to some of those provided by FORTRAN and FORTRAN-like languages.

```
(DEFUN FORTRANPRINT (ATOMS COLUMNS)
       (PROG (N)
             (SETQ N 0)
             (TERPRI)                              ;Print blank line.
             NEXTATOM
             (COND ((OR (NULL ATOMS) (NULL COLUMNS))
                    (RETURN T)))
             NEXTSPACE
             (COND ((LESSP N (CAR COLUMNS))        ;Need another space?
                    (PRINC '| |)
                    (SETQ N (ADD1 N))
                    (GO NEXTSPACE)))
             (PRINC (CAR ATOMS))                   ;Print next atom.
             (SETQ N (PLUS N                       ;Allow for atom.
                           (LENGTH (EXPLODE (CAR ATOMS)))))
             (SETQ ATOMS (CDR ATOMS))
             (SETQ COLUMNS (CDR COLUMNS))
             (GO NEXTATOM)))
```

Here is an example in which several FORTRANPRINTs are used to print a timetable in columns determined by the value of TABS:

```
(DEFUN TABLE (TABS)
       (TERPRI)
       (FORTRANPRINT '(CONCORD LINCOLN WALTHAM CAMBRIDGE BOSTON) TABS)
       (TERPRI)
       (FORTRANPRINT '(6:11 6:17 6:29 6:41 6:55) TABS)
       (FORTRANPRINT '(6:46 6:52 7:05 7:17 7:31) TABS))
  TABLE

(TABLE '(0 12 24 36 48))

CONCORD      LINCOLN      WALTHAM      CAMBRIDGE    BOSTON

6:11         6:17         6:29         6:41         6:55
6:46         6:52         7:05         7:17         7:31
  T
```

Problems

Problem 7-2: Define ECH01, a function that reads s-expressions and returns them without evaluation, and define ECH02, a function that returns with evaluation.

Problem 7-3: It is often handy to have a way of printing information without too many parentheses to confuse things. Define a function P that takes one argument and prints the atoms in it as a single nonnested list. This illustrates:

```
(SETQ S '(A (B (C D) E) (F (G (H I) J) K) L))
  (A (B (C D) E) (F (G (H I) J) K) L)

(P (LIST '(THE ATOMS IN S ARE) S))
  (THE ATOMS IN S ARE A B C D E F G H I J K L)
```

Then define PC, a function that takes two arguments and prints only if the first evaluates to nonNIL. When PC does print, it is to print the second argument as a nonnested list. And finally define RQ, a function that prints its argument as a nonnested list, reads an s-expression given by the user, and returns its value. Note that PC is a useful debugging function and RQ is handy for requesting values from the user from deep inside some program. These examples illustrate:

```
(SETQ N 'ROBBIE)
  ROBBIE

(PC (NOT (NUMBERP N)) (LIST 'WARNING N '(IS NOT A NUMBER)))
  (WARNING ROBBIE IS NOT A NUMBER)
```

```
(RQ '(PLEASE SUPPLY A VALUE FOR PI))
  (PLEASE SUPPLY A VALUE FOR PI) 3.14159
  3.14159
```

Problem 7-4: Define P1, a function that behaves like P except that it prints no parentheses at all:

```
(SETQ S '(A B C D E F G H I J K L))
  (A B C D E F G H I J K L)

(P1 (LIST '(THE ATOMS IN S ARE) S))
  THE ATOMS IN S ARE A B C D E F G H I J K L
```

Problem 7-5: The examples throughout this book indicate that PRINT indents two spaces before starting. This is done for clarity only, and in actual LISP systems, PRINT does not indent. Define BOOKPRINT, a function that does.

Problem 7-6: Define ATOMCAR and ATOMCDR. The first returns the first character in a given atom, and the second returns all of the characters but the first.

Problem 7-7: Define PRINT-MATRIX, a procedure which prints each row of the array called MATRIX on a separate line. The two arguments given to the procedure specify the number of rows and the number of columns in the array.

Problem 7-8: A program's requests for information and the user's replies can be arranged neatly on the screen of a display terminal. A list can conveniently describe the desired format by specifying the rows and columns in which input and output is to appear. The form for a sublist specifying where some words are to be printed might be as follows:

```
(<list of words> <row> <column>)
```

Similarly the following sublist would specify the location where information typed by the user is to appear:

```
(<word> <row> <column>)
```

The symbolic atom <word> is set to the number or word typed by the user. Define FORM-ENTRY, a procedure which prints and reads according to such formatting instructions. Assume that items are ordered, so that a cursor can always be positioned by spacing forward and going to the next line (a cursor is a mark that indicates where the next character is to appear on the screen). Further assume that TOP-OF-SCREEN clears the display screen and positions the cursor in the top left hand corner.

The following is a test case for your procedure:

```
(FORM-ENTRY '(((LAST NAME:) 1. 0.)
              (LAST-NAME 1. 11.)
              ((FIRST NAME:) 1. 30.)
              (FIRST-NAME 1. 43.)
              ((AGE:) 3. 0.)
              (AGE 3. 6.)
              ((SEX:) 3. 14.)
              (SEX 3. 20.)
              ((OCCUPATION:) 3. 30.)
              (OCCUPATION 3. 43.)
              ((YOUR ADDRESS NEXT:) 6. 0.)
              ((STREET:) 9. 0.)
              (STREET 9. 9.)
              ((NUMBER:) 9. 30.)
              (NUMBER 9. 39.)
              ((CITY:) 10. 0.)
              (CITY 10. 9.)
              ((STATE:) 10. 30.)
              (STATE 10. 38.)
              ((ZIP CODE:) 10. 52.)
              (ZIP 10. 63.)
              ((THANK YOU VERY MUCH) 13. 0.))
```

Problem 7-9: It is essential, when using LISP, to have some tools to help get the layout of s-expressions right. Otherwise the parentheses become snares. LISP systems therefore have various functions to do what is called prettyprinting. Write a simple PRETTYPRINT, a function that takes one s-expression and prints it neatly by printing the first element, then the second, and then the third, neatly lining up the third and the rest under the second.

```
(<element-1> <element-2>
             <element-3>
             .
             .
             .
             <element-n>)
```

Each element itself should be prettyprinted so that the whole s-expression has a transparent appearance.

Project: Write a simple real time editor in LISP.

Summary

- PRINT and READ facilitate communication.
- Special conventions make weird atom names possible.
- Atoms can be broken apart, put together, and created.
- Exotic input/output functions lie beyond PRINT and READ.
- Formatted printing is easily arranged.

References

Pretty-printing is discussed by Goldstein [1973], Oppen [1979], as well as Hearn and Norman [1979]. A powerful real time editor written in LISP is described by Greenberg [1979].

PROLOG PROGRAMMING FOR ARTIFICIAL INTELLIGENCE

Ivan Bratko

E. Kardelj University · J. Stefan Institute
Yugoslavia

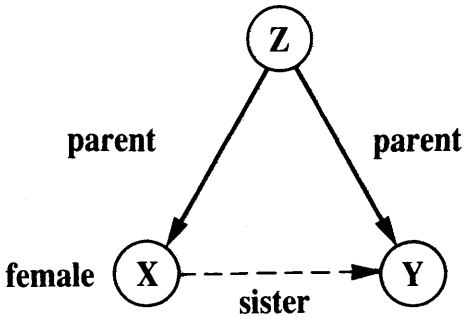
ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
Don Mills, Ontario · Amsterdam · Sydney · Singapore · Tokyo
Madrid · Bogota · Santiago · San Juan

CONTENTS

PART ONE

THE PROLOG LANGUAGE



1 An Overview of Prolog

This chapter reviews basic mechanisms of Prolog through an example program. Although the treatment is largely informal many important concepts are introduced.

1.1 An example program: defining family relations

Prolog is a programming language for symbolic, non-numeric computation. It is specially well suited for solving problems that involve objects and relations between objects. Figure 1.1 shows an example: a family relation. The fact that Tom is a parent of Bob can be written in Prolog as:

parent(tom, bob).

Here we choose **parent** as the name of a relation; **tom** and **bob** are its argu-

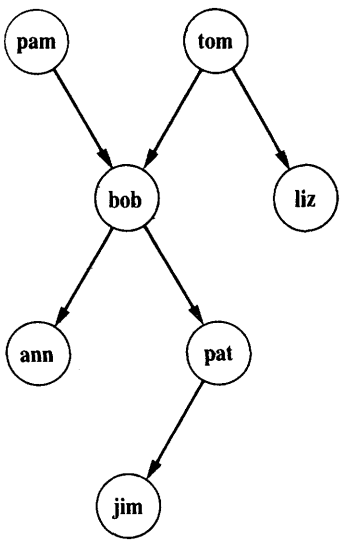


Figure 1.1 A family tree.

ments. For reasons that will become clear later we write names like **tom** with an initial lower-case letter. The whole family tree of Figure 1.1 is defined by the following Prolog program:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

This program consists of six *clauses*. Each of these clauses declares one fact about the **parent** relation.

When this program has been communicated to the Prolog system, Prolog can be posed some questions about the **parent** relation. For example, Is Bob a parent of Pat? This question can be communicated to the Prolog system by typing into the terminal:

```
?- parent( bob, pat).
```

Having found this as an asserted fact in the program, Prolog will answer:

```
yes
```

A further query can be:

```
?- parent( liz, pat).
```

Prolog answers

```
no
```

because the program does not mention anything about Liz being a parent of Pat. It also answers 'no' to the question

```
?- parent( tom, ben).
```

because the program has not even heard of the name Ben.

More interesting questions can also be asked. For example: Who is Liz's parent?

```
?- parent( X, liz).
```

Prolog's answer will not be just 'yes' or 'no' this time. Prolog will tell us what is the (yet unknown) value of X such that the above statement is true. So the

answer is:

X = tom

The question Who are Bob's children? can be communicated to Prolog as:

?- parent(bob, X).

This time there is more than just one possible answer. Prolog first answers with one solution:

X = ann

We may now want to see other solutions. We can say that to Prolog (in most Prolog implementations by typing a semicolon), and Prolog will find other answers:

X = pat

If we request more solutions again, Prolog will answer 'no' because all the solutions have been exhausted.

Our program can be asked an even broader question: Who is a parent of whom? Another formulation of this question is:

Find X and Y such that X is a parent of Y.

This is expressed in Prolog by:

?- parent(X, Y).

Prolog now finds all the parent-child pairs one after another. The solutions will be displayed one at a time as long as we tell Prolog we want more solutions, until all the solutions have been found. The answers are output as:

X = pam
Y = bob;

X = tom
Y = bob;

X = tom
Y = liz;

...

We can stop the stream of solutions by typing, for example, a period instead of a semicolon (this depends on the implementation of Prolog).

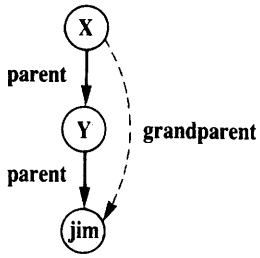


Figure 1.2 The **grandparent** relation expressed as a composition of two **parent** relations.

Our example program can be asked still more complicated questions like: Who is a grandparent of Jim? As our program does not directly know the **grandparent** relation this query has to be broken down into two steps, as illustrated by Figure 1.2.

(1) Who is a parent of Jim? Assume that this is some Y.
(2) Who is a parent of Y? Assume that this is some X.

Such a composed query is written in Prolog as a sequence of two simple ones:

```
?- parent( Y, jim), parent( X, Y).
```

The answer will be:

```
X = bob
Y = pat
```

Our composed query can be read: Find such X and Y that satisfy the following two requirements:

parent(Y, jim) and **parent(X, Y)**

If we change the order of the two requirements the logical meaning remains the same:

parent(X, Y) and **parent(Y, jim)**

We can indeed do this in our Prolog program and the query

```
?- parent( X, Y), parent( Y, jim).
```

will produce the same result.

In a similar way we can ask: Who are Tom's grandchildren?

```
?- parent( tom, X), parent( X, Y).
```

Prolog's answers are:

```
X = bob
Y = ann;

X = bob
Y = pat
```

Yet another question could be: Do Ann and Pat have a common parent? This can be expressed again in two steps:

(1) Who is a parent, X, of Ann?
(2) Is (this same) X a parent of Pat?

The corresponding question to Prolog is then:

```
?- parent( X, ann), parent( X, pat).
```

The answer is:

```
X = bob
```

Our example program has helped to illustrate some important points:

- It is easy in Prolog to define a relation, such as the **parent** relation, by stating the n-tuples of objects that satisfy the relation.
- The user can easily query the Prolog system about relations defined in the program.
- A Prolog program consists of *clauses*. Each clause terminates with a full stop.
- The arguments of relations can (among other things) be: concrete objects, or constants (such as **tom** and **ann**), or general objects such as X and Y. Objects of the first kind in our program are called *atoms*. Objects of the second kind are called *variables*.
- Questions to the system consist of one or more *goals*. A sequence of goals, such as

  parent(X, ann), parent(X, pat)

  means the conjunction of the goals:

  X is a parent of Ann, *and*
  X is a parent of Pat.

  The word 'goals' is used because Prolog accepts questions as goals that are to be satisfied.
- An answer to a question can be either positive or negative, depending on

whether the corresponding goal can be satisfied or not. In the case of a positive answer we say that the corresponding goal was *satisfiable* and that the goal *succeeded*. Otherwise the goal was *unsatisfiable* and it *failed*.

- If several answers satisfy the question then Prolog will find as many of them as desired by the user.

Exercises

1.1 Assuming the **parent** relation as defined in this section (see Figure 1.1), what will be Prolog's answers to the following questions?

(a) ?- **parent(jim, X).**

(b) ?- **parent(X, jim).**

(c) ?- **parent(pam, X), parent(X, pat).**

(d) ?- **parent(pam, X), parent(X, Y), parent(Y, jim).**

1.2 Formulate in Prolog the following questions about the **parent** relation:

(a) Who is Pat's parent?

(b) Does Liz have a child?

(c) Who is Pat's grandparent?

1.2 Extending the example program by rules

Our example program can be easily extended in many interesting ways. Let us first add the information on the sex of the people that occur in the **parent** relation. This can be done by simply adding the following facts to our program:

```
female( pam).
male( tom).
male( bob).
female( liz).
female( pat).
female( ann).
male( jim).
```

The relations introduced here are **male** and **female**. These relations are unary (or one-place) relations. A binary relation like **parent** defines a relation between *pairs* of objects; on the other hand, unary relations can be used to declare simple yes/no properties of objects. The first unary clause above can be read: Pam is a female. We could convey the same information declared in the two unary relations with one binary relation, **sex**, instead. An alternative piece

of program would then be:

```
sex( pam, feminine).
sex( tom, masculine).
sex( bob, masculine).
...
```

As our next extension to the program let us introduce the **offspring** relation as the inverse of the **parent** relation. We could define **offspring** in a similar way as the **parent** relation; that is, by simply providing a list of simple facts about the **offspring** relation, each fact mentioning one pair of people such that one is an offspring of the other. For example:

```
offspring( liz, tom).
```

However, the **offspring** relation can be defined much more elegantly by making use of the fact that it is the inverse of **parent**, and that **parent** has already been defined. This alternative way can be based on the following logical statement:

For all X and Y,
  Y is an offspring of X if
  X is a parent of Y.

This formulation is already close to the formalism of Prolog. The corresponding Prolog clause which has the same meaning is:

```
offspring( Y, X)  :-  parent( X, Y).
```

This clause can also be read as:

For all X and Y,
  if X is a parent of Y then
  Y is an offspring of X.

Prolog clauses such as

```
offspring( Y, X)  :-  parent( X, Y).
```

are called *rules*. There is an important difference between facts and rules. A fact like

```
parent( tom, liz).
```

is something that is always, unconditionally, true. On the other hand, rules specify things that may be true if some condition is satisfied. Therefore we say that rules have:

- a condition part (the right-hand side of the rule) and

- a conclusion part (the left-hand side of the rule).

The conclusion part is also called the *head* of a clause and the condition part the *body* of a clause. For example:

```
offspring( Y, X)  :-  parent( X, Y).
      head                body
```

If the condition **parent(X, Y)** is true then a logical consequence of this is **offspring(Y, X)**.

How rules are actually used by Prolog is illustrated by the following example. Let us ask our program whether Liz is an offspring of Tom:

```
?- offspring( liz, tom).
```

There is no fact about offsprings in the program, therefore the only way to consider this question is to apply the rule about offsprings. The rule is general in the sense that it is applicable to any objects X and Y; therefore it can also be applied to such particular objects as **liz** and **tom**. To apply the rule to **liz** and **tom**, Y has to be substituted with **liz**, and X with **tom**. We say that the variables X and Y become instantiated to:

```
X = tom    and    Y = liz
```

After the instantiation we have obtained a special case of our general rule. The special case is:

```
offspring( liz, tom)  :-  parent( tom, liz).
```

The condition part has become

```
parent( tom, liz)
```

Now Prolog tries to find out whether the condition part is true. So the initial goal

```
offspring( liz, tom)
```

has been replaced with the subgoal

```
parent( tom, liz)
```

This (new) goal happens to be trivial as it can be found as a fact in our program. This means that the conclusion part of the rule is also true, and Prolog will answer the question with **yes**.

Let us now add more family relations to our example program. The

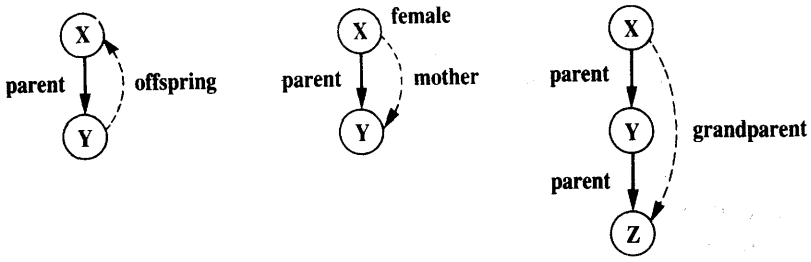


Figure 1.3 Definition graphs for the relations **offspring**, **mother** and **grandparent** in terms of other relations.

specification of the **mother** relation can be based on the following logical statement:

For all X and Y,
X is the mother of Y if
X is a parent of Y and
X is a female.

This is translated into Prolog as the following rule:

```
mother( X, Y)  :-  parent( X, Y), female( X).
```

A comma between two conditions indicates the conjunction of the conditions, meaning that *both* conditions have to be true.

Relations such as **parent, offspring** and **mother** can be illustrated by diagrams such as those in Figure 1.3. These diagrams conform to the following conventions. Nodes in the graphs correspond to objects – that is, arguments of relations. Arcs between nodes correspond to binary (or two-place) relations. The arcs are oriented so as to point from the first argument of the relation to the second argument. Unary relations are indicated in the diagrams by simply marking the corresponding objects with the name of the relation. The relations that are being defined are represented by dashed arcs. So each diagram should be understood as follows: if relations shown by solid arcs hold, then the relation shown by a dashed arc also holds. The **grandparent** relation can be, according to Figure 1.3, immediately written in Prolog as:

```
grandparent( X, Z)  :-  parent( X, Y), parent( Y, Z).
```

At this point it will be useful to make a comment on the layout of our programs. Prolog gives us almost full freedom in choosing the layout of the program. So we can insert spaces and new lines as it best suits our taste. In general we want to make our programs look nice and tidy, and, above all, easy to read. To this end we will often choose to write the head of a clause and each

goal of the body on a separate line. When doing this, we will indent goals in order to make the difference between the head and the goals more visible. For example, the **grandparent** rule would be, according to this convention, written as follows:

```
grandparent( X, Z) :-
  parent( X, Y),
  parent( Y, Z).
```

Figure 1.4 illustrates the **sister** relation:

For any X and Y,
X is a sister of Y if
(1) both X and Y have the same parent, and
(2) X is a female.

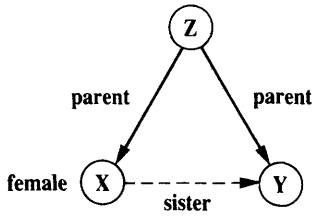


Figure 1.4 Defining the **sister** relation.

The graph in Figure 1.4 can be translated into Prolog as:

```
sister( X, Y) :-
  parent( Z, X),
  parent( Z, Y),
  female( X).
```

Notice the way in which the requirement 'both X and Y have the same parent' has been expressed. The following logical formulation was used: some Z must be a parent of X, and this *same* Z must be a parent of Y. An alternative, but less elegant way would be to say: Z1 is a parent of X, and Z2 is a parent of Y, and Z1 is equal to Z2.

We can now ask:

```
?- sister( ann, pat).
```

The answer will be 'yes', as expected (see Figure 1.1). Therefore we might

conclude that the **sister** relation, as defined, works correctly. There is, however, a rather subtle flaw in our program which is revealed if we ask the question Who is Pat's sister?:

```
?- sister( X, pat).
```

Prolog will find two answers, one of which may come as a surprise:

```
X = ann;
X = pat
```

So, Pat is a sister to herself?! This is probably not what we had in mind when defining the **sister** relation. However, according to our rule about sisters Prolog's answer is perfectly logical. Our rule about sisters does not mention that X and Y must not be the same if X is to be a sister of Y. As this is not required Prolog (rightfully) assumes that X and Y can be the same, and will as a consequence find that any female who has a parent is a sister of herself.

To correct our rule about sisters we have to add that X and Y must be different. We will see in later chapters how this can be done in several ways, but for the moment we will assume that a relation **different** is already known to Prolog, and that

```
different( X, Y)
```

is satisfied if and only if X and Y are not equal. An improved rule for the **sister** relation can then be:

```
sister( X, Y) :-
  parent( Z, X),
  parent( Z, Y),
  female( X),
  different( X, Y).
```

Some important points of this section are:

- Prolog programs can be extended by simply adding new clauses.
- Prolog clauses are of three types: *facts*, *rules* and *questions*.
- *Facts* declare things that are always, unconditionally true.
- *Rules* declare things that are true depending on a given condition.
- By means of *questions* the user can ask the program what things are true.
- Prolog clauses consist of the *head* and the *body*. The body is a list of *goals* separated by commas. Commas are understood as conjunctions.
- Facts are clauses that have the empty body. Questions only have the body. Rules have the head and the (non-empty) body.

- In the course of computation, a variable can be substituted by another object. We say that a variable becomes *instantiated*.
- Variables are assumed to be universally quantified and are read as 'for all'. Alternative readings are, however, possible for variables that appear only in the body. For example

  hasachild(X) :- parent(X, Y).

  can be read in two ways:

  (a) *For all* X and Y,
  if X is a parent of Y then
  X has a child.

  (b) *For all* X,
  X has a child if
  there is *some* Y such that X is a parent of Y.

Exercises

1.3 Translate the following statements into Prolog rules:

(a) Everybody who has a child is happy (introduce a one-argument relation **happy**).

(b) For all X, if X has a child who has a sister then X has two children (introduce new relation **hastwochildren**).

1.4 Define the relation **grandchild** using the **parent** relation. Hint: It will be similar to the **grandparent** relation (see Figure 1.3).

1.5 Define the relation **aunt(X, Y)** in terms of the relations **parent** and **sister**. As an aid you can first draw a diagram in the style of Figure 1.3 for the **aunt** relation.

1.3 A recursive rule definition

Let us add one more relation to our family program, the **predecessor** relation. This relation will be defined in terms of the **parent** relation. The whole definition can be expressed with two rules. The first rule will define the direct (immediate) predecessors and the second rule the indirect predecessors. We say that some X is an indirect predecessor of some Z if there is a parentship chain of people between X and Z, as illustrated in Figure 1.5. In our example of Figure 1.1, Tom is a direct predecessor of Liz and an indirect predecessor of Pat.

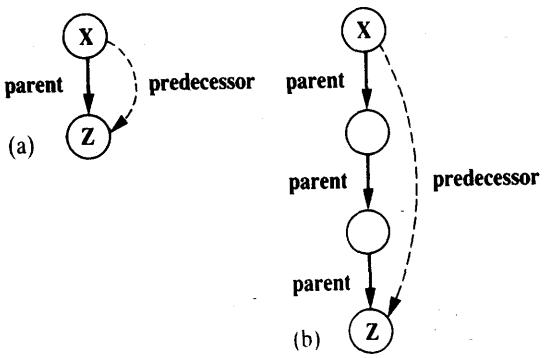


Figure 1.5 Examples of the **predecessor** relation: (a) **X** is a *direct* predecessor of **Z**; (b) **X** is an indirect predecessor of **Z**.

The first rule is simple and can be formulated as:

For all X and Z,
X is a predecessor of Z if
X is a parent of Z.

This is straightforwardly translated into Prolog as:

```
predecessor( X, Z) :-
  parent( X, Z).
```

The second rule, on the other hand, is more complicated because the chain of parents may present some problems. One attempt to define indirect predecessors could be as shown in Figure 1.6. According to this, the **predecessor**

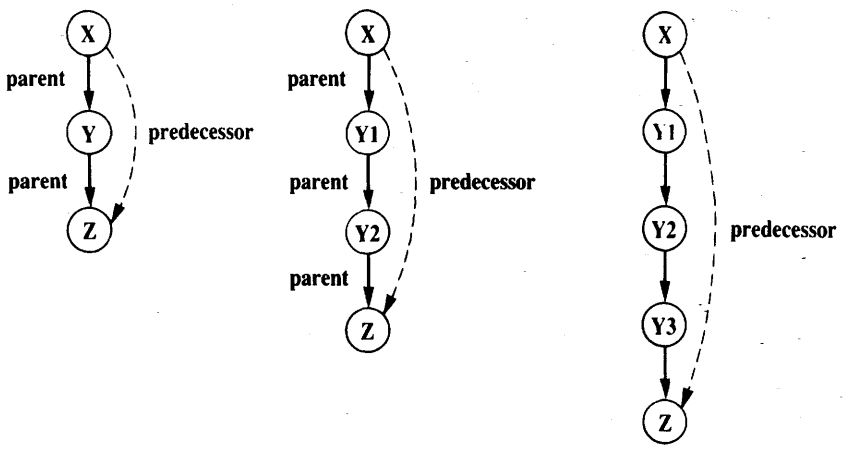


Figure 1.6 Predecessor-successor pairs at various distances.

relation would be defined by a set of clauses as follows:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  parent( Y, Z).

predecessor( X, Z) :-
  parent( X, Y1),
  parent( Y1, Y2),
  parent( Y2, Z).

predecessor( X, Z) :-
  parent( X, Y1),
  parent( Y1, Y2),
  parent( Y2, Y3),
  parent( Y3, Z).

...
```

This program is lengthy and, more importantly, it only works to some extent. It would only discover predecessors to a certain depth in a family tree because the length of the chain of people between the predecessor and the successor would be limited according to the length of our predecessor clauses.

There is, however, an elegant and correct formulation of the **predecessor** relation: it will be correct in the sense that it will work for predecessors at any depth. The key idea is to define the **predecessor** relation in terms of itself. Figure 1.7 illustrates the idea:

For all X and Z,
  X is a predecessor of Z if
  there is a Y such that
  (1) X is a parent of Y and
  (2) Y is a predecessor of Z.

A Prolog clause with the above meaning is:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

We have thus constructed a complete program for the **predecessor** relation, which consists of two rules: one for direct predecessors and one for indirect predecessors. Both rules are rewritten together here:

```
predecessor( X, Z) :-
  parent( X, Z).
```

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

The key to this formulation was the use of **predecessor** itself in its definition. Such a definition may look surprising in view of the question: When defining something, can we use this same thing that has not yet been completely defined? Such definitions are, in general, called *recursive* definitions. Logically, they are perfectly correct and understandable, which is also intuitively obvious if we look at Figure 1.7. But will the Prolog system be able to use recursive rules? It turns out that Prolog can indeed very easily use recursive definitions. Recursive programming is, in fact, one of the fundamental principles of programming in Prolog. It is not possible to solve tasks of any significant complexity in Prolog without the use of recursion.

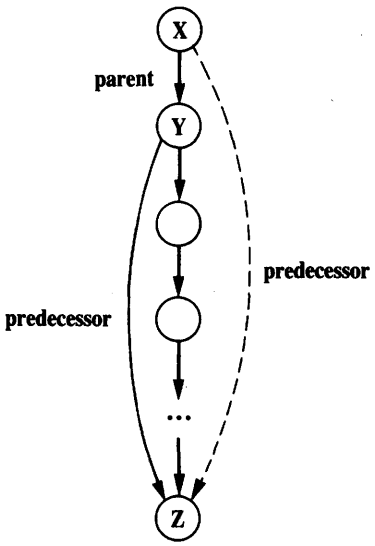


Figure 1.7 Recursive formulation of the **predecessor** relation.

Going back to our program, we can ask Prolog: Who are Pam's successors? That is: Who is a person that Pam is his or her predecessor?

```
?- predecessor( pam, X).

X = bob;

X = ann;

X = pat;

X = jim
```

Prolog's answers are of course correct and they logically follow from our definition of the **predecessor** and the **parent** relation. There is, however, a rather important question: *How* did Prolog actually use the program to find these answers?

An informal explanation of how Prolog does this is given in the next section. But first let us put together all the pieces of our family program, which

```
parent( pam, bob).                  % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).                       % Pam is female
male( tom).                         % Tom is male
male( bob).
female( liz).
female( ann).
female( pat).
male( jim).

offspring( Y, X) :-                 % Y is an offspring of X if
  parent( X, Y).                    % X is a parent of Y

mother( X, Y) :-                    % X is the mother of Y if
  parent( X, Y),                    % X is a parent of Y and
  female( X).                       % X is female

grandparent( X, Z) :-               % X is a grandparent of Z if
  parent( X, Y),                    % X is a parent of Y and
  parent( Y, Z).                    % Y is a parent of Z

sister( X, Y) :-                    % X is a sister of Y if
  parent( Z, X),
  parent( Z, Y),                    % X and Y have the same parent and
  female( X),                       % X is female and
  different( X, Y).                 % X and Y are different

predecessor( X, Z) :-               % Rule pr1: X is a predecessor of Z
  parent( X, Z).

predecessor( X, Z) :-               % Rule pr2: X is a predecessor of Z
  parent( X, Y),
  predecessor( Y, Z).
```

Figure 1.8 The family program.

was extended gradually by adding new facts and rules. The final form of the program is shown in Figure 1.8. Looking at Figure 1.8, two further points are in order here: the first will introduce the term 'procedure', the second will be about comments in programs.

The program in Figure 1.8 defines several relations – **parent, male, female, predecessor**, etc. The **predecessor** relation, for example, is defined by two clauses. We say that these two clauses are *about* the **predecessor** relation. Sometimes it is convenient to consider the whole set of clauses about the same relation. Such a set of clauses is called a *procedure*.

In Figure 1.8, the two rules about the **predecessor** relation have been distinguished by the names 'pr1' and 'pr2', added as *comments* to the program. These names will be used later as references to these rules. Comments are, in general, ignored by the Prolog system. They only serve as a further clarification to the person who reads the program. Comments are distinguished in Prolog from the rest of the program by being enclosed in special brackets '/*' and '*/'. Thus comments in Prolog look like this:

```
/* This is a comment */
```

Another method, more practical for short comments, uses the percent character '%'. Everything between '%' and the end of the line is interpreted as a comment:

```
% This is also a comment
```

Exercise

1.6 Consider the following alternative definition of the **predecessor** relation:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( Y, Z),
  predecessor( X, Y).
```

Does this also seem to be a proper definition of predecessors? Can you modify the diagram of Figure 1.7 so that it would correspond to this new definition?

1.4 How Prolog answers questions

This section gives an informal explanation of *how* Prolog answers questions.

A question to Prolog is always a sequence of one or more goals. To answer a question, Prolog tries to satisfy all the goals. What does it mean to *satisfy* a goal? To satisfy a goal means to demonstrate that the goal is true,

assuming that the relations in the program are true. In other words, to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, Prolog also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If Prolog cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then Prolog's answer to the question will be 'no'.

An appropriate view of the interpretation of a Prolog program in mathematical terms is then as follows: Prolog accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem – that is, to demonstrate that it can be logically derived from the axioms.

We will illustrate this view by a classical example. Let the axioms be:

All men are fallible.
Socrates is a man.

A theorem that logically follows from these two axioms is:

Socrates is fallible.

The first axiom above can be rewritten as:

For all X, if X is a man then X is fallible.

Accordingly, the example can be translated into Prolog as follows:

```
fallible( X) :- man( X).        % All men are fallible
man( socrates).                 % Socrates is a man
?- fallible( socrates).         % Socrates is fallible?
yes
```

A more complicated example from the family program of Figure 1.8 is:

```
?- predecessor( tom, pat).
```

We know that **parent(bob, pat)** is a fact. Using this fact and rule *pr1* we can conclude **predecessor(bob, pat)**. This is a *derived* fact: it cannot be found explicitly in our program, but it can be derived from facts and rules in the program. An inference step, such as this, can be written in a more compact form as:

```
parent( bob, pat)  ==>  predecessor( bob, pat)
```

This can be read: from **parent(bob, pat)** it follows **predecessor(bob, pat)**, by

rule *pr1*. Further, we know that **parent(tom, bob)** is a fact. Using this fact and the derived fact **predecessor(bob, pat)** we can conclude **predecessor(tom, pat)**, by rule *pr2*. We have thus shown that our goal statement **predecessor(tom, pat)** is true. This whole inference process of two steps can be written as:

parent(bob, pat) ==> **predecessor(bob, pat)**

parent(tom, bob) *and* **predecessor(bob, pat)** ==> **predecessor(tom, pat)**

We have thus shown *what* can be a sequence of steps that satisfy a goal – that is, make it clear that the goal is true. Let us call this a proof sequence. We have not, however, shown *how* the Prolog system actually finds such a proof sequence.

Prolog finds the proof sequence in the inverse order to that which we have just used. Instead of starting with simple facts given in the program, Prolog starts with the goals and, using rules, substitutes the current goals with new goals, until new goals happen to be simple facts. Given the question

```
?- predecessor( tom, pat).
```

Prolog will try to satisfy this goal. In order to do so it will try to find a clause in the program from which the above goal could immediately follow. Obviously, the only clauses relevant to this end are *pr1* and *pr2*. These are the rules about the **predecessor** relation. We say that the heads of these rules *match* the goal.

The two clauses, *pr1* and *pr2*, represent two alternative ways for Prolog to proceed. Prolog first tries that clause which appears first in the program:

```
predecessor( X, Z) :- parent( X, Z).
```

Since the goal is **predecessor(tom, pat)**, the variables in the rule must be instantiated as follows:

X = tom, Z = pat

The original goal **predecessor(tom, pat)** is then replaced by a new goal:

parent(tom, pat)

This step of using a rule to transform a goal into another goal, as above, is graphically illustrated in Figure 1.9. There is no clause in the program whose head matches the goal **parent(tom, pat)**, therefore this goal fails. Now Prolog *backtracks* to the original goal in order to try an alternative way to derive the top goal **predecessor(tom, pat)**. The rule *pr2* is thus tried:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

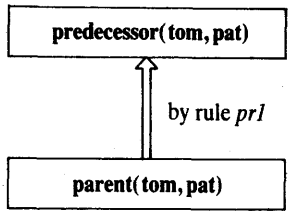


Figure 1.9 The first step of the execution. The top goal is true if the bottom goal is true.

As before, the variables X and Z become instantiated as:

X = tom, Z = pat

But Y is not instantiated yet. The top goal **predecessor(tom, pat)** is replaced by two goals:

parent(tom, Y),
predecessor(Y, pat)

This executional step is shown in Figure 1.10, which is an extension to the situation we had in Figure 1.9.

Being now faced with *two* goals, Prolog tries to satisfy them in the order that they are written. The first one is easy as it matches one of the facts in the program. The matching forces Y to become instantiated to **bob**. Thus the first goal has been satisfied, and the remaining goal has become:

predecessor(bob, pat)

To satisfy this goal the rule *pr1* is used again. Note that this (second) application of the same rule has nothing to do with its previous application. Therefore, Prolog uses a new set of variables in the rule each time the rule is applied. To

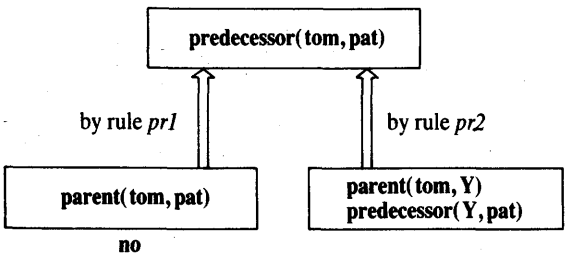


Figure 1.10 Execution trace continued from Figure 1.9.

indicate this we shall rename the variables in rule *pr1* for this application as follows:

```
predecessor( X', Z') :-
  parent( X', Z').
```

The head has to match our current goal **predecessor(bob, pat)**. Therefore

X' = bob, Z' = pat

The current goal is replaced by

parent(bob, pat)

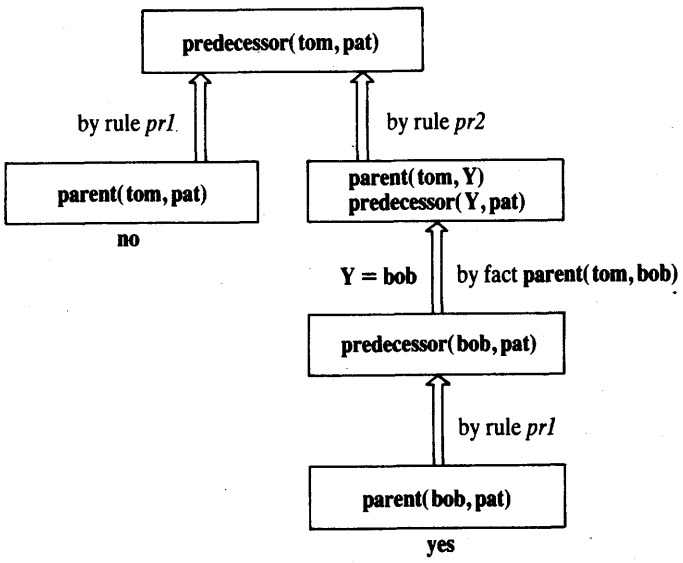


Figure 1.11 The complete execution trace to satisfy the goal **predecessor(tom, pat)**. The right-hand branch proves the goal is satisfiable.

This goal is immediately satisfied because it appears in the program as a fact. This completes the execution trace which is graphically shown in Figure 1.11.

The graphical illustration of the execution trace in Figure 1.11 has the form of a tree. The nodes of the tree correspond to goals, or to lists of goals that are to be satisfied. The arcs between the nodes correspond to the application of (alternative) program clauses that transform the goals at one node into the goals at another node. The top goal is satisfied when a path is found from the root node (top goal) to a leaf node labelled 'yes'. A leaf is labelled 'yes' if it is a simple fact. The execution of Prolog programs is the searching for such paths.

During the search Prolog may enter an unsuccessful branch. When Prolog discovers that a branch fails it automatically *backtracks* to the previous node and tries to apply an alternative clause at that node.

Exercise

1.7 Try to understand how Prolog derives answers to the following questions, using the program of Figure 1.8. Try to draw the corresponding derivation diagrams in the style of Figures 1.9 to 1.11. Will any backtracking occur at particular questions?

(a) ?- **parent(pam, bob).**

(b) ?- **mother(pam, bob).**

(c) ?- **grandparent(pam, ann).**

(d) ?- **grandparent(bob, jim).**

1.5 Declarative and procedural meaning of programs

In our examples so far it has always been possible to understand the results of the program without exactly knowing *how* the system actually found the results. It therefore makes sense to distinguish between two levels of meaning of Prolog programs; namely,

- the *declarative meaning* and
- the *procedural meaning.*

The declarative meaning is concerned only with the *relations* defined by the program. The declarative meaning thus determines *what* will be the output of the program. On the other hand, the procedural meaning also determines *how* this output is obtained; that is, how are the relations actually evaluated by the Prolog system.

The ability of Prolog to work out many procedural details on its own is considered to be one of its specific advantages. It encourages the programmer to consider the declarative meaning of programs relatively independently of their procedural meaning. Since the results of the program are, in principle, determined by its declarative meaning, this should be (in principle) sufficient for writing programs. This is of practical importance because the declarative aspects of programs are usually easier to understand than the procedural details. To take full advantage of this, the programmer should concentrate mainly on the declarative meaning and, whenever possible, avoid being distracted by the executional details. These should be left to the greatest possible extent to the Prolog system itself.

This declarative approach indeed often makes programming in Prolog easier than in typical procedurally oriented programming languages such as Pascal. Unfortunately, however, the declarative approach is not always sufficient. It will later become clear that, especially in large programs, the procedural aspects cannot be completely ignored by the programmer for practical reasons of executional efficiency. Nevertheless, the declarative style of thinking about Prolog programs should be encouraged and the procedural aspects ignored to the extent that is permitted by practical constraints.

Summary

- Prolog programming consists of defining relations and querying about relations.
- A program consists of *clauses*. These are of three types: *facts, rules* and *questions*.
- A relation can be specified by *facts*, simply stating the n-tuples of objects that satisfy the relation, or by stating *rules* about the relation.
- A *procedure* is a set of clauses about the same relation.
- Querying about relations, by means of *questions*, resembles querying a database. Prolog's answer to a question consists of a set of objects that satisfy the question.
- In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly *backtracking*. All this is done automatically by the Prolog system and is, in principle, hidden from the user.
- Two types of meaning of Prolog programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.
- The following concepts have been introduced in this chapter:

  clause, fact, rule, question
  the head of a clause, the body of a clause
  recursive rule, recursive definition
  procedure
  atom, variable
  instantiation of a variable
  goal
  goal is satisfiable, goal succeeds
  goal is unsatisfiable, goal fails
  backtracking
  declarative meaning, procedural meaning

References

Various implementations of Prolog use different syntactic conventions. In this book we use the so-called Edinburgh syntax (also called DEC-10 syntax, established by the influential implementation of Prolog for the DEC-10 computer; Pereira *et al.* 1978) which has been adopted by many popular Prologs such as Quintus Prolog, CProlog, Poplog, etc.

Bowen, D. L. (1981) *DECsystem-10 Prolog User's Manual.* University of Edinburgh: Department of Artificial Intelligence.

Mellish, C. and Hardy, S. (1984) *Integrating Prolog in the POPLOG environment. Implementations of Prolog* (J. A. Campbell, ed.). Ellis Horwood.

Pereira, F. (1982) *C-Prolog User's Manual.* University of Edinburgh: Department of Computer-Aided Architectural Design.

Pereira, L. M., Pereira, F. and Warren, D. H. D. (1978) *User's Guide to DECsystem-10 Prolog.* University of Edinburgh: Department of Artificial Intelligence.

Quintus Prolog User's Guide and Reference Manual. Palo Alto: Quintus Computer Systems Inc. (1985).

2 Syntax and Meaning of Prolog Programs

This chapter gives a systematic treatment of the syntax and semantics of basic concepts of Prolog, and introduces structured data objects. The topics included are:

- simple data objects (atoms, numbers, variables)
- structured objects
- matching as the fundamental operation on objects
- declarative (or non-procedural) meaning of a program
- procedural meaning of a program
- relation between the declarative and procedural meanings of a program
- altering the procedural meaning by reordering clauses and goals

Most of these topics have already been reviewed in Chapter 1. Here the treatment will become more formal and detailed.

2.1 Data objects

Figure 2.1 shows a classification of data objects in Prolog. The Prolog system recognizes the type of an object in the program by its syntactic form. This is possible because the syntax of Prolog specifies different forms for each type of

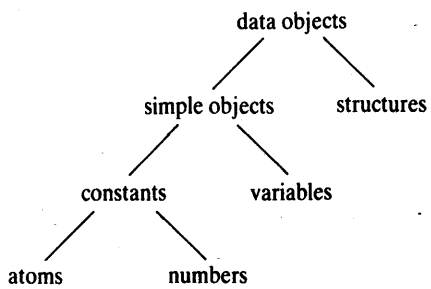


Figure 2.1 Data objects in Prolog.

data objects. We have already seen a method for distinguishing between atoms and variables in Chapter 1: variables start with upper-case letters whereas atoms start with lower-case letters. No additional information (such as data-type declaration) has to be communicated to Prolog in order to recognize the type of an object.

2.1.1 Atoms and numbers

In Chapter 1 we have seen some simple examples of atoms and variables. In general, however, they can take more complicated forms – that is, strings of the following characters:

- upper-case letters A, B, ..., Z
- lower-case letters a, b, ..., z
- digits 0, 1, 2, ..., 9
- special characters such as + − * / < > = : . & _ ~

Atoms can be constructed in three ways:

(1) Strings of letters, digits and the underscore character, '_', starting with a lower-case letter:

anna
nil
x25
x_25
x_25AB
x_
x___y
alpha_beta_procedure
miss_Jones
sarah_jones

(2) Strings of special characters:

<--->
======>
...
.:.
::=

When using atoms of this form, some care is necessary because some strings of special characters already have a predefined meaning; an example is ':-'.

(3) Strings of characters enclosed in single quotes. This is useful if we want, for example, to have an atom that starts with a capital letter. By enclosing

it in quotes we make it distinguishable from variables:

'Tom'
'South_America'
'Sarah Jones'

Numbers used in Prolog include integer numbers and real numbers. The syntax of integers is simple, as illustrated by the following examples:

1 1313 0 −97

Not all integer numbers can be represented in a computer, therefore the range of integers is limited to an interval between some smallest and some largest number permitted by a particular Prolog implementation. Normally the range allowed by an implementation is at least between −16383 and 16383, and often it is considerably wider.

The treatment of real numbers depends on the implementation of Prolog. We will assume the simple syntax of numbers, as shown by the following examples:

3.14 −0.0035 100.2

Real numbers are not used very much in typical Prolog programming. The reason for this is that Prolog is primarily a language for symbolic, non-numeric computation, as opposed to number crunching oriented languages such as Fortran. In symbolic computation, integers are often used, for example, to count the number of items in a list; but there is little need for real numbers.

Apart from this lack of necessity to use real numbers in typical Prolog applications, there is another reason for avoiding real numbers. In general, we want to keep the meaning of programs as neat as possible. The introduction of real numbers somewhat impairs this neatness because of numerical errors that arise due to rounding when doing arithmetic. For example, the evaluation of the expression

10000 + 0.0001 − 10000

may result in 0 instead of the correct result 0.0001.

2.1.2 Variables

Variables are strings of letters, digits and underscore characters. They start with an upper-case letter or an underscore character:

X
Result
Object2
Participant_list

```
ShoppingList
_x23
_23
```

When a variable appears in a clause once only, we do not have to invent a name for it. We can use the so-called 'anonymous' variable, which is written as a single underscore character. For example, let us consider the following rule:

```
hasachild( X) :- parent( X, Y).
```

This rule says: for all X, X has a child if X is a parent of some Y. We are defining the property **hasachild** which, as it is meant here, does not depend on the name of the child. Thus, this is a proper place in which to use an anonymous variable. The clause above can thus be rewritten:

```
hasachild( X) :- parent( X, _).
```

Each time a single underscore character occurs in a clause it represents a new anonymous variable. For example, we can say that there is somebody who has a child if there are two objects such that one is a parent of the other:

```
somebody_has_child :- parent( _, _).
```

This is equivalent to:

```
somebody_has_child :- parent( X, Y).
```

But this is, of course, quite different from:

```
somebody_has_child :- parent( X, X).
```

If the anonymous variable appears in a question clause then its value is not output when Prolog answers the question. If we are interested in people who have children, but not in the names of the children, then we can simply ask:

```
?- parent( X, _).
```

The *lexical scope* of variable names is one clause. This means that, for example, if the name X15 occurs in two clauses, then it signifies two different variables. But each occurrence of X15 within the same clause means the same variable. The situation is different for constants: the same atom always means the same object in any clause – that is, throughout the whole program.

2.1.3 Structures

Structured objects (or simply *structures*) are objects that have several components. The components themselves can, in turn, be structures. For example,

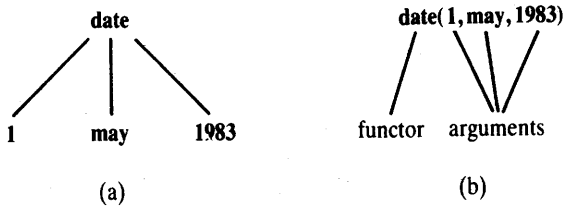


Figure 2.2 Date is an example of a structured object: (a) as it is represented as a tree; (b) as it is written in Prolog.

the date can be viewed as a structure with three components: day, month, year. Although composed of several components, structures are treated in the program as single objects. In order to combine the components into a single object we have to choose a *functor*. A suitable functor for our example is **date**. Then the date 1st May 1983 can be written as:

date(1, may, 1983)

(see Figure 2.2).

All the components in this example are constants (two integers and one atom). Components can also be variables or other structures. Any day in May can be represented by the structure:

date(Day, may, 1983)

Note that **Day** is a variable and can be instantiated to any object at some later point in the execution.

This method for data structuring is simple and powerful. It is one of the reasons why Prolog is so naturally applied to problems that involve symbolic manipulation.

Syntactically, all data objects in Prolog are *terms*. For example,

may

and

date(1, may, 1983)

are terms.

All structured objects can be pictured as trees (see Figure 2.2 for an example). The root of the tree is the functor, and the offsprings of the root are the components. If a component is also a structure then it is a subtree of the tree that corresponds to the whole structured object.

Our next example will show how structures can be used to represent some simple geometric objects (see Figure 2.3). A point in two-dimensional space is

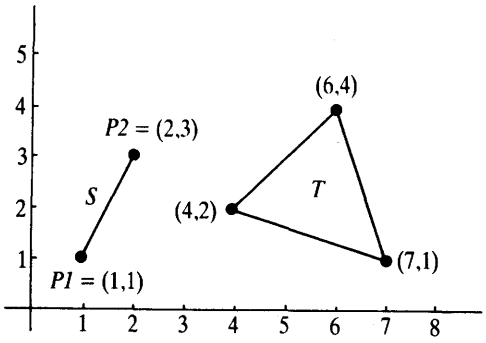


Figure 2.3 Some simple geometric objects.

defined by its two coordinates; a line segment is defined by two points; and a triangle can be defined by three points. Let us choose the following functors:

point	for points,
seg	for line segments, and
triangle	for triangles.

Then the objects in Figure 2.3 can be represented by the following Prolog terms:

P1 = point(1,1)
P2 = point(2,3)
S = seg(P1, P2) = seg(point(1,1), point(2,3))
T = triangle(point(4,2), point(6,4), point(7,1))

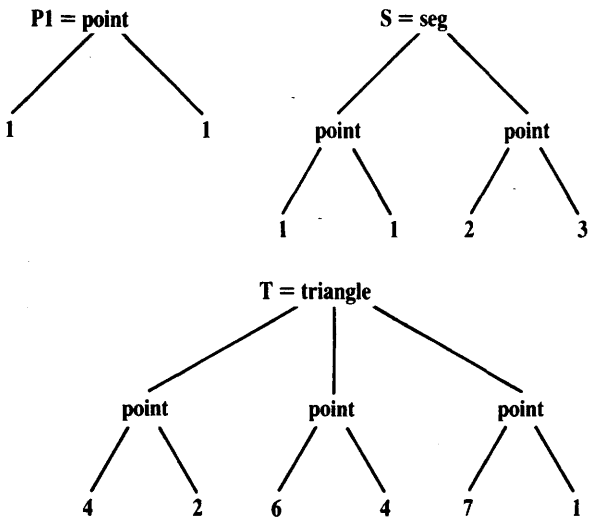


Figure 2.4 Tree representation of the objects in Figure 2.3.

The corresponding tree representation of these objects is shown in Figure 2.4. In general, the functor at the root of the tree is called the *principal functor* of the term.

If in the same program we also had points in three-dimensional space then we could use another functor, **point3**, say, for their representation:

point3(X, Y, Z)

We can, however, use the same name, **point**, for points in both two and three dimensions, and write for example:

point(X1, Y1) and **point(X, Y, Z)**

If the same name appears in the program in two different roles, as is the case for **point** above, the Prolog system will recognize the difference by the number of arguments, and will interpret this name as two functors: one of them with two arguments and the other one with three arguments. This is so because each functor is defined by two things:

(1) the name, whose syntax is that of atoms;
(2) the *arity* – that is, the number of arguments.

As already explained, all structured objects in Prolog are trees, represented in the program by terms. We will study two more examples to illustrate how naturally complicated data objects can be represented by Prolog terms. Figure 2.5 shows the tree structure that corresponds to the arithmetic expression

$(a + b) * (c - 5)$

According to the syntax of terms introduced so far this can be written, using the symbols '*', '+' and '−' as functors, as follows:

***(+(a, b), −(c, 5))**

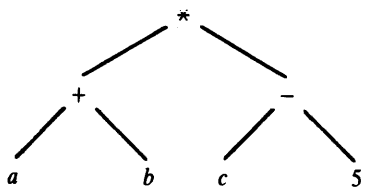


Figure 2.5 A tree structure that corresponds to the arithmetic expression $(a + b)*(c - 5)$.

This is of course a legal Prolog term; but this is not the form that we would normally like to have. We would normally prefer the usual, infix notation as used in mathematics. In fact, Prolog also allows us to use the infix notation so that the symbols '*', '+' and '−' are written as infix operators. Details of how the programmer can define his or her own operators will be discussed in Chapter 3.

As the last example we consider some simple electric circuits shown in Figure 2.6. The right-hand side of the figure shows the tree representation of these circuits. The atoms **r1, r2, r3** and **r4** are the names of the resistors. The

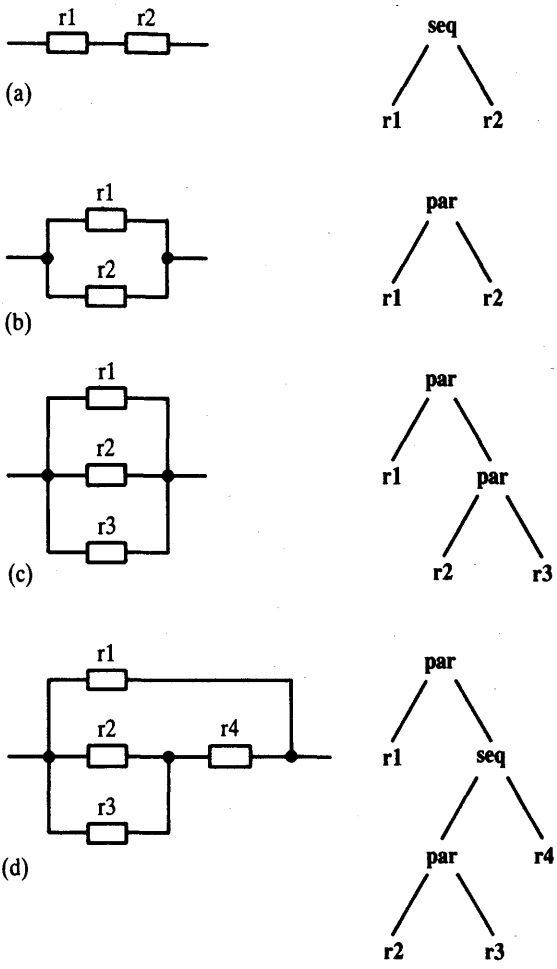


Figure 2.6 Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.

functors **par** and **seq** denote the parallel and the sequential compositions of resistors respectively. The corresponding Prolog terms are:

```
seq( r1, r2)
par( r1, r2)
par( r1, par( r2, r3) )
par( r1, seq( par( r2, r3), r4) )
```

Exercises

2.1 Which of the following are syntactically correct Prolog objects? What kinds of object are they (atom, number, variable, structure)?

(a) **Diana**
(b) **diana**
(c) **'Diana'**
(d) **_diana**
(e) **'Diana goes south'**
(f) **goes(diana, south)**
(g) **45**
(h) **5(X, Y)**
(i) **+(north, west)**
(j) **three(Black(Cats))**

2.2 Suggest a representation for rectangles, squares and circles as structured Prolog objects. Use an approach similar to that in Figure 2.4. For example, a rectangle can be represented by four points (or maybe three points only). Write some example terms that represent some concrete objects of these types using the suggested representation.

2.2 Matching

In the previous section we have seen how terms can be used to represent complex data objects. The most important operation on terms is *matching*. Matching alone can produce some interesting computation.

Given two terms, we say that they *match* if:

(1) they are identical, or
(2) the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

For example, the terms **date(D, M, 1983)** and **date(D1, may, Y1)** match. One instantiation that makes both terms identical is:

- **D** is instantiated to **D1**
- **M** is instantiated to **may**
- **Y1** is instantiated to **1983**

This instantiation is more compactly written in the familiar form in which Prolog outputs results:

```
D = D1
M = may
Y1 = 1983
```

On the other hand, the terms **date(D, M, 1983)** and **date(D1, M1, 1444)** do not match, nor do the terms **date(X, Y, Z)** and **point(X, Y, Z)**.

Matching is a process that takes as input two terms and checks whether they match. If the terms do not match we say that this process *fails*. If they do match then the process *succeeds* and it also instantiates the variables in both terms to such values that the terms become identical.

Let us consider again the matching of the two dates. The request for this operation can be communicated to the Prolog system by the following question, using the operator '=':

```
?- date( D, M, 1983) = date( D1, may, Y1).
```

We have already mentioned the instantiation **D = D1, M = may, Y1 = 1983**, which achieves the match. There are, however, other instantiations that also make both terms identical. Two of them are as follows:

```
D = 1
D1 = 1
M = may
Y1 = 1983

D = third
D1 = third
M = may
Y1 = 1983
```

These two instantiations are said to be *less general* than the first one because they constrain the values of the variables D and D1 stronger than necessary. For making both terms in our example identical, it is only important that D and D1 have the same value, although this value can be anything. Matching in Prolog always results in the *most general* instantiation. This is the instantiation that commits the variables to the least possible extent, thus leaving the greatest

possible freedom for further instantiations if further matching is required. As an example consider the following question:

```
?- date( D, M, 1983)  =  date( D1, may, Y1),
   date( D, M, 1983)  =  date( 15, M, Y).
```

To satisfy the first goal, Prolog instantiates the variables as follows:

```
D = D1
M = may
Y1 = 1983
```

After having satisfied the second goal, the instantiation becomes more specific as follows:

```
D = 15
D1 = 15
M = may
Y1 = 1983
Y = 1983
```

This example also illustrates that variables, during the execution of consecutive goals, typically become instantiated to increasingly more specific values.

The general rules to decide whether two terms, S and T, match are as follows:

(1) If S and T are constants then S and T match only if they are the same object.

(2) If S is a variable and T is anything, then they match, and S is instantiated to T. Conversely, if T is a variable then T is instantiated to S.

(3) If S and T are structures then they match only if
   (a) S and T have the same principal functor, and
   (b) all their corresponding components match.

   The resulting instantiation is determined by the matching of the components.

The last of these rules can be visualized by considering the tree representation of terms, as in the example of Figure 2.7. The matching process starts at the root (the principal functors). As both functors match, the process proceeds to the arguments where matching of the pairs of corresponding arguments occurs. So the whole matching process can be thought of as consisting of the

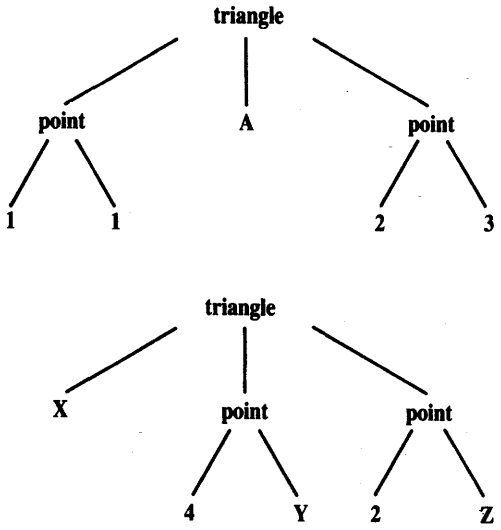


Figure 2.7 Matching **triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y), point(2,Z)).**

following sequence of (simpler) matching operations:

triangle = triangle,
point(1,1) = X,
A = point(4,Y),
point(2,3) = point(2,Z).

The whole matching process succeeds because all the matchings in the sequence succeed. The resulting instantiation is:

X = point(1,1)
A = point(4,Y)
Z = 3

The following example will illustrate how matching alone can be used for interesting computation. Let us return to the simple geometric objects of Figure 2.4, and define a piece of program for recognizing horizontal and vertical line segments. 'Vertical' is a property of segments, so it can be formalized in Prolog as a unary relation. Figure 2.8 helps to formulate this relation. A segment is vertical if the x-coordinates of its end-points are equal, otherwise there is no other restriction on the segment. The property 'horizontal' is similarly formulated, with only x and y interchanged. The following program, consisting of two facts, does the job:

```
vertical( seg( point(X,Y), point(X,Y1) ).

horizontal( seg( point(X,Y), point(X1,Y) ).
```

The following conversation is possible with this program:

```
?- vertical( seg( point(1,1), point(1,2) ) ).
yes
?- vertical( seg( point(1,1), point(2,Y) ) ).
no
?- horizontal( seg( point(1,1), point(2,Y) ) ).
Y = 1
```

The first question was answered 'yes' because the goal in the question matched one of the facts in the program. For the second question no match was possible. In the third question, Y was forced to become 1 by matching the fact about horizontal segments.

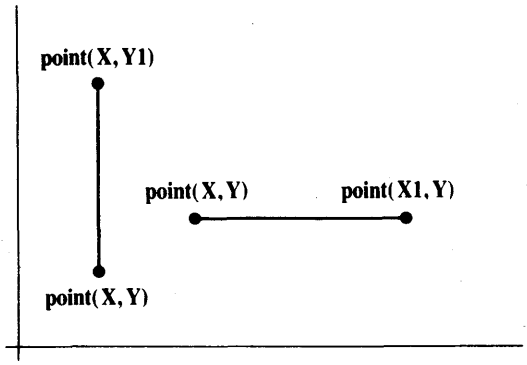


Figure 2.8 Illustration of vertical and horizontal line segments.

A more general question to the program is: Are there any vertical segments that start at the point (2,3)?

```
?- vertical( seg( point(2,3), P) ).
P = point(2,Y)
```

This answer means: Yes, any segment that ends at any point (2,Y), which means anywhere on the vertical line $x = 2$. It should be noted that Prolog's actual answer would probably not look as neat as above, but (depending on the Prolog implementation used) something like this:

```
P = point(2,_136)
```

This is, however, only a cosmetic difference. Here _136 is a variable that has

not been instantiated. _136 is, of course, a legal variable name that the system has constructed during the execution. The system has to generate new names in order to rename the user's variables in the program. This is necessary for two reasons: first, because the same name in different clauses signifies different variables, and second, in successive applications of the same clause, its 'copy' with a new set of variables is used each time.

Another interesting question to our program is: Is there a segment that is both vertical and horizontal?

```
?- vertical( S), horizontal( S).

S = seg( point(X,Y), point(X,Y) )
```

This answer by Prolog says: Yes, any segment that is degenerated to a point has the property of being vertical and horizontal at the same time. The answer was, again, derived simply by matching. As before, some internally generated names may appear in the answer, instead of the variable names X and Y.

Exercises

2.3 Will the following matching operations succeed or fail? If they succeed, what are the resulting instantiations of variables?

(a) **point(A, B) = point(1, 2)**

(b) **point(A, B) = point(X, Y, Z)**

(c) **plus(2, 2) = 4**

(d) **+(2, D) = +(E, 2)**

(e) **triangle(point(-1,0), P2, P3) = triangle(P1, point(1,0), point(0,Y))**

The resulting instantiation defines a family of triangles. How would you describe this family?

2.4 Using the representation for line segments as described in this section, write a term that represents any vertical line segment at $x = 5$.

2.5 Assume that a rectangle is represented by the term **rectangle(P1, P2, P3, P4)** where the P's are the vertices of the rectangle positively ordered. Define the relation

regular(R)

which is true if R is a rectangle whose sides are vertical and horizontal.

2.3 Declarative meaning of Prolog programs

We have already seen in Chapter 1 that Prolog programs can be understood in two ways: declaratively and procedurally. In this and the next section we will

consider a more formal definition of the declarative and procedural meanings of programs in basic Prolog. But first let us look at the difference between these two meanings again.

Consider a clause

```
P :- Q, R.
```

where P, Q and R have the syntax of terms. Some alternative declarative readings of this clause are:

P is true if Q and R are true.

From Q and R follows P.

Two alternative procedural readings of this clause are:

To solve problem P, *first* solve the subproblem Q and *then* the subproblem R.

To satisfy P, *first* satisfy Q and *then* R.

Thus the difference between the declarative readings and the procedural ones is that the latter do not only define the logical relations between the head of the clause and the goals in the body, but also the *order* in which the goals are processed.

Let us now formalize the declarative meaning.

The declarative meaning of programs determines whether a given goal is true, and if so, for what values of variables it is true. To precisely define the declarative meaning we need to introduce the concept of *instance* of a clause. An instance of a clause C is the clause C with each of its variables substituted by some term. A *variant* of a clause C is such an instance of the clause C where each variable is substituted by another variable. For example, consider the clause:

```
hasachild( X) :- parent( X, Y).
```

Two variants of this clause are:

```
hasachild( A) :- parent( A, B).
hasachild( X1) :- parent( X1, X2).
```

Instances of this clause are:

```
hasachild( peter) :- parent( peter, Z).
hasachild( barry) :- parent( barry, small(caroline) ).
```

Given a program and a goal G, the declarative meaning says:

A goal G is true (that is, satisfiable, or logically follows from the program) if and only if

(1) there is a clause C in the program such that
(2) there is a clause instance I of C such that
(a) the head of I is identical to G, and
(b) all the goals in the body of I are true.

This definition extends to Prolog questions as follows. In general, a question to the Prolog system is a *list* of goals separated by commas. A list of goals is true if *all* the goals in the list are true for the *same* instantiation of variables. The values of the variables result from the most general instantiation.

A comma between goals thus denotes the *conjunction* of goals: they *all* have to be true. But Prolog also accepts the *disjunction* of goals: *any one* of the goals in a disjunction has to be true. Disjunction is indicated by a semicolon. For example,

```
P :- Q; R.
```

is read: P is true if Q is true *or* R is true. The meaning of this clause is thus the same as the meaning of the following two clauses together:

```
P :- Q.
P :- R.
```

The comma binds stronger than the semicolon. So the clause

```
P :- Q, R; S, T, U.
```

is understood as

```
P :- ( Q, R); ( S, T, U).
```

and means the same as the clauses:

```
P :- Q, R.
P :- S, T, U.
```

Exercises

2.6 Consider the following program:

```
f( 1, one).
f( s(1), two).
```

```
f( s(s(1)), three).
f( s(s(s(X))), N) :-
  f( X, N).
```

How will Prolog answer the following questions? Whenever several answers are possible, give at least two.

(a) ?- f(s(1), A).

(b) ?- f(s(s(1)), two).

(c) ?- f(s(s(s(s(s(s(1)))))), C).

(d) ?- f(D, three).

2.7 The following program says that two people are relatives if
(a) one is a predecessor of the other, or
(b) they have a common predecessor, or
(c) they have a common successor:

```
relatives( X, Y) :-
  predecessor( X, Y).

relatives( X, Y) :-
  predecessor( Y, X).

relatives( X, Y) :-          % X and Y have a common predecessor
  predecessor( Z, X),
  predecessor( Z, Y).

relatives( X, Y) :-          % X and Y have a common successor
  predecessor( X, Z),
  predecessor( Y, Z).
```

Can you shorten this program by using the semicolon notation?

2.8 Rewrite the following program without using the semicolon notation.

```
translate( Number, Word) :-
  Number = 1, Word = one;
  Number = 2, Word = two;
  Number = 3, Word = three.
```

2.4 Procedural meaning

The procedural meaning specifies *how* Prolog answers questions. To answer a question means to try to satisfy a list of goals. They can be satisfied if the variables that occur in the goals can be instantiated in such a way that the goals logically follow from the program. Thus the procedural meaning of Prolog is a procedure for executing a list of goals with respect to a given program. To 'execute goals' means: try to satisfy them.

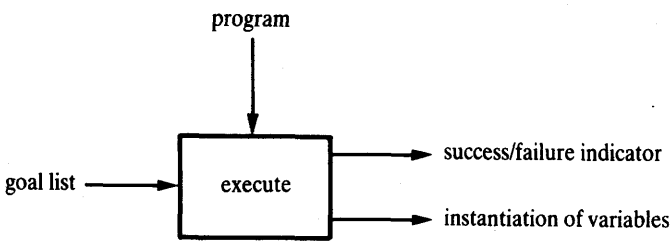


Figure 2.9 Input/output view of the procedure that executes a list of goals.

Let us call this procedure **execute**. As shown in Figure 2.9, the inputs to and the outputs from this procedure are:

input: a program and a goal list
output: a success/failure indicator and an instantiation of variables

The meaning of the two output results is as follows:

(1) The success/failure indicator is 'yes' if the goals are satisfiable and 'no' otherwise. We say that 'yes' signals a *successful* termination and 'no' a *failure*.

(2) An instantiation of variables is only produced in the case of a successful termination; in the case of failure there is no instantiation.

In Chapter 1, we have in effect already discussed informally what procedure **execute** does, under the heading 'How Prolog answers questions?'. What follows in the rest of this section is just a more formal and systematic description of this process, and can be skipped without seriously affecting the understanding of the rest of the book.

Particular operations in the goal execution process are illustrated by the example in Figure 2.10. It may be helpful to study Figure 2.10 before reading the following general description.

PROGRAM

```
big( bear).                    % Clause 1
big( elephant).                % Clause 2
small( cat).                   % Clause 3

brown( bear).                  % Clause 4
black( cat).                   % Clause 5
gray( elephant).               % Clause 6

dark( Z) :-                    % Clause 7: Anything black is dark
  black( Z).

dark( Z) :-                    % Clause 8: Anything brown is dark
  brown( Z).
```

QUESTION

```
?- dark( X), big( X).        % Who is dark and big?
```

EXECUTION TRACE

(1) Initial goal list: **dark(X), big(X).**

(2) Scan the program from top to bottom looking for a clause whose head matches the first goal **dark(X)**. Clause 7 found:

```
dark( Z) :- black( Z).
```

Replace the first goal by the instantiated body of clause 7, giving a new goal list.

```
black( X), big( X)
```

(3) Scan the program to find a match with **black(X)**. Clause 5 found: **black(cat)**. This clause has no body, so the goal list, properly instantiated, shrinks to:

```
big( cat)
```

(4) Scan the program for the goal **big(cat)**. No clause found. Therefore backtrack to step (3) and undo the instantiation **X = cat**. Now the goal list is again:

```
black( X), big( X)
```

Continue scanning the program below clause 5. No clause found. Therefore backtrack to step (2) and continue scanning below clause 7. Clause 8 is found:

```
dark( Z) :- brown( Z).
```

Replace the first goal in the goal list by **brown(X)**, giving:

```
brown( X), big( X)
```

(5) Scan the program to match **brown(X)**, finding **brown(bear)**. This clause has no body, so the goal list shrinks to:

```
big( bear)
```

(6) Scan the program and find clause **big(bear)**. It has no body so the goal list shrinks to empty. This indicates successful termination, and the corresponding variable instantiation is:

```
X = bear
```

Figure 2.10 An example to illustrate the procedural meaning of Prolog: a sample trace of the procedure **execute**.

To execute a list of goals

G1, G2, ..., Gm

the procedure **execute** does the following:

- If the goal list is empty then terminate with *success*.
- If the goal list is not empty then continue with (the following) operation called 'SCANNING'.
- *SCANNING*: Scan through the clauses in the program from top to bottom until the first clause, C, is found such that the head of C matches the first goal G1. If there is no such clause then terminate with *failure*.

  If there is such a clause C of the form

  H :- B1, ..., Bn.

  then rename the variables in C to obtain a variant C′ of C, such that C′ and the list G1, ..., Gm have no common variables. Let C′ be

  H′ :- B1′, ..., Bn′.

  Match G1 and H′; let the resulting instantiation of variables be S.

  In the goal list G1, G2, ..., Gm, replace G1 with the list B1′, ..., Bn′, obtaining a new goal list

  B1′, ..., Bn′, G2, ..., Gm

  (Note that if C is a fact then $n = 0$ and the new goal list is shorter than the original one; such shrinking of the goal list may eventually lead to the empty list and thereby a successful termination.)

  Substitute the variables in this new goal list with new values as specified in the instantiation S, obtaining another goal list

  B1″, ..., Bn″, G2′, ..., Gm′

- Execute (recursively with this same procedure) this new goal list. If the execution of this new goal list terminates with success then terminate the execution of the original goal list also with success. If the execution of the new goal list is not successful then abandon this new goal list and go back to SCANNING through the program. Continue the scanning with the clause that immediately follows the clause C (C is the clause that was last used) and try to find a successful termination using some other clause.

This procedure is more compactly written in a Pascal-like notation in Figure 2.11.

Several additional remarks are in order here regarding the procedure **execute** as presented. First, it was not explicitly described how the final resulting instantiation of variables is produced. It is the instantiation S which led to a successful termination, and was possibly further refined by additional instantiations that were done in the nested recursive calls to **execute**.

Whenever a recursive call to **execute** fails, the execution returns to SCANNING, continuing at the program clause C that had been last used before. As the application of the clause C did not lead to a successful termination Prolog has to try an alternative clause to proceed. What effectively happens is that Prolog abandons this whole part of the unsuccessful execution and backtracks to the point (clause C) where this failed branch of the execution was started. When the procedure backtracks to a certain point, all the variable instantiations that were done after that point are undone. This ensures that Prolog systematically examines all the possible alternative paths of execution until one is found that eventually succeeds, or until all of them have been shown to fail.

We have already seen that even after a successful termination the user can force the system to backtrack to search for more solutions. In our description of **execute** this detail was left out.

Of course, in actual implementations of Prolog, several other refinements have to be added to **execute**. One of them is to reduce the amount of

---

procedure *execute* (*Program*, *GoalList*, *Success*);

Input arguments:
  Program: list of clauses
  GoalList: list of goals
Output argument:
  Success: truth value; *Success* will become true if
    GoalList is true with respect to *Program*
Local variables:
  Goal: goal
  OtherGoals: list of goals
  Satisfied: truth value
  MatchOK: truth value
  Instant: instantiation of variables
  H, *H'*, *B1*, *B1'*, ..., *Bn*, *Bn'*: goals
Auxiliary functions:
  empty(L): returns true if L is the empty list
  head(L): returns the first element of list L
  tail(L): returns the rest of L
  append(L1,L2): appends list L2 at the end of list L1
  match(T1,T2,MatchOK,Instant): tries to match terms T1 and T2; if
    succeeds then *MatchOK* is true and *Instant* is the corresponding
    instantiation of variables
  substitute(Instant,Goals): substitutes variables in *Goals* according
    to instantiation *Instant*

```
begin
  if empty(GoalList) then Success := true
  else
    begin
      Goal := head(GoalList);
      OtherGoals := tail(GoalList);
      Satisfied := false;
      while not Satisfied and "more clauses in program" do
        begin
          Let next clause in Program be
            H :- B1, ..., Bn.
          Construct a variant of this clause
            H' :- B1', ..., Bn'.
          match(Goal,H',MatchOK,Instant);
          if MatchOK then
            begin
              NewGoals := append([B1',...,Bn'],OtherGoals);
              NewGoals := substitute(Instant,NewGoals);
              execute(Program,NewGoals,Satisfied)
            end
        end;
      Success := Satisfied
    end
end;
```

Figure 2.11 Executing Prolog goals.

scanning through the program clauses to improve efficiency. So a practical Prolog implementation will not scan through all the clauses of the program, but will only consider the clauses about the relation in the current goal.

Exercise

2.9 Consider the program in Figure 2.10 and simulate, in the style of Figure 2.10, Prolog's execution of the question:

```
?- big( X), dark( X).
```

Compare your execution trace with that of Figure 2.10 when the question was essentially the same, but with the goals in the order:

```
?- dark( X), big( X).
```

In which of the two cases does Prolog have to do more work before the answer is found?

2.5 Example: monkey and banana

The monkey and banana problem is often used as a simple example of problem solving. Our Prolog program for this problem will show how the mechanisms of matching and backtracking can be used in such exercises. We will develop the program in the non-procedural way, and then study its procedural behaviour in detail. The program will be compact and illustrative.

We will use the following variation of the problem. There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana, but he cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use. The monkey can perform the following actions: walk on the floor, climb the box, push the box around (if it is already at the box) and grasp the banana if standing on the box directly under the banana. Can the monkey get the banana?

One important task in programming is that of finding a representation of the problem in terms of concepts of the programming language used. In our case we can think of the 'monkey world' as always being in some *state* that can change in time. The current state is determined by the positions of the objects. For example, the initial state of the world is determined by:

(1) Monkey is at door.
(2) Monkey is on floor.
(3) Box is at window.
(4) Monkey does not have banana.

It is convenient to combine all of these four pieces of information into one structured object. Let us choose the word 'state' as the functor to hold the four components together. Figure 2.12 shows the initial state represented as a structured object.

Our problem can be viewed as a one-person game. Let us now formalize the rules of the game. First, the goal of the game is a situation in which the monkey has the banana; that is, any state in which the last component is 'has':

```
state( _, _, _, has)
```

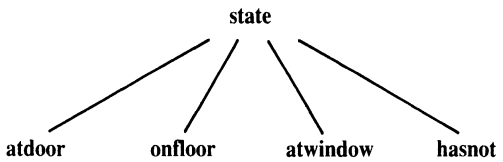


Figure 2.12 The initial state of the monkey world represented as a structured object. The four components are: horizontal position of monkey, vertical position of monkey, position of box, monkey has or has not the banana.

Second, what are the allowed moves that change the world from one state to another? There are four types of moves:

(1) grasp banana,
(2) climb box,
(3) push box,
(4) walk around.

Not all moves are possible in every possible state of the world. For example, the move 'grasp' is only possible if the monkey is standing on the box directly under the banana (which is in the middle of the room) and does not have the banana yet. Such rules can be formalized in Prolog as a three-place relation named **move**:

```
move( State1, M, State2)
```

The three arguments of the relation specify a move thus:

```
State1 --------> State2
          M
```

State1 is the state before the move, M is the move executed and **State2** is the state after the move.

The move 'grasp', with its necessary precondition on the state before the move, can be defined by the clause:

```
move( state( middle, onbox, middle, hasnot),     % Before move
      grasp,                                     % Move
      state( middle, onbox, middle, has) ).      % After move
```

This fact says that after the move the monkey has the banana, and he has remained on the box in the middle of the room.

In a similar way we can express the fact that the monkey on the floor can walk from any horizontal position P1 to any position P2. The monkey can do this regardless of the position of the box and whether it has the banana or not. All this can be defined by the following Prolog fact:

```
move( state( P1, onfloor, B, H),
      walk( P1, P2),                             % Walk from P1 to P2
      state( P2, onfloor, B, H) ).
```

Note that this clause says many things, including, for example:

- the move executed was 'walk from some position P1 to some position P2';
- the monkey is on the floor before and after the move;

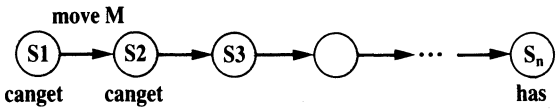


Figure 2.13 Recursive formulation of **canget**.

- the box is at some point B which remained the same after the move;
- the 'has banana' status remains the same after the move.

The clause actually specifies a whole set of possible moves because it is applicable to any situation that matches the specified state before the move. Such a specification is therefore sometimes also called a move *schema*. Due to the concept of Prolog variables such schemas can be easily programmed in Prolog.

The other two types of moves, 'push' and 'climb', can be similarly specified.

The main kind of question that our program will have to answer is: Can the monkey in some initial state S get the banana? This can be formulated as a predicate

canget(S)

where the argument S is a state of the monkey world. The program for **canget** can be based on two observations:

(1) For any state S in which the monkey already has the banana, the predicate **canget** must certainly be true; no move is needed in this case. This corresponds to the Prolog fact:

```
canget( state( _, _, _, has) ).
```

(2) In other cases one or more moves are necessary. The monkey can get the banana in any state S1 if there is some move M from state S1 to some state S2, such that the monkey can then get the banana in state S2 (in zero or more moves). This principle is illustrated in Figure 2.13. A Prolog clause that corresponds to this rule is:

```
canget( S1) :-
  move( S1, M, S2),
  canget( S2).
```

This completes our program which is shown in Figure 2.14.

The formulation of **canget** is recursive and is similar to that of the **predecessor** relation of Chapter 1 (compare Figures 2.13 and 1.7). This principle is used in Prolog again and again.

```
% Legal moves

move( state( middle, onbox, middle, hasnot),
      grasp,                                    % Grasp banana
      state( middle, onbox, middle, has) ).

move( state( P, onfloor, P, H),
      climb,                                    % Climb box
      state( P, onbox, P, H) ).

move( state( P1, onfloor, P1, H),
      push( P1, P2),                            % Push box from P1 to P2
      state( P2, onfloor, P2, H) ).

move( state( P1, onfloor, B, H),
      walk( P1, P2),                            % Walk from P1 to P2
      state( P2, onfloor, B, H) ).

% canget( State): monkey can get banana in State

canget( state( _, _, _, has) ).                 % can 1: Monkey already has it

canget( State1) :-                              % can 2: Do some work to get it
  move( State1, Move, State2),                  % Do something
  canget( State2).                              % Get it now
```

Figure 2.14 A program for the monkey and banana problem.

We have developed our monkey and banana program in the non-procedural way. Let us now study its *procedural* behaviour by considering the following question to the program:

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

Prolog's answer is 'yes'. The process carried out by Prolog to reach this answer proceeds, according to the procedural semantics of Prolog, through a sequence of goal lists. It involves some search for right moves among the possible alternative moves. At some point this search will take a wrong move leading to a dead branch. At this stage, backtracking will help it to recover. Figure 2.15 illustrates this search process.

To answer the question Prolog had to backtrack once only. A right sequence of moves was found almost straight away. The reason for this efficiency of the program was the order in which the clauses about the **move** relation occurred in the program. The order in our case (luckily) turned out to be quite suitable. However, less lucky orderings are possible. According to the rules of the game, the monkey could just as easily try to walk here or there

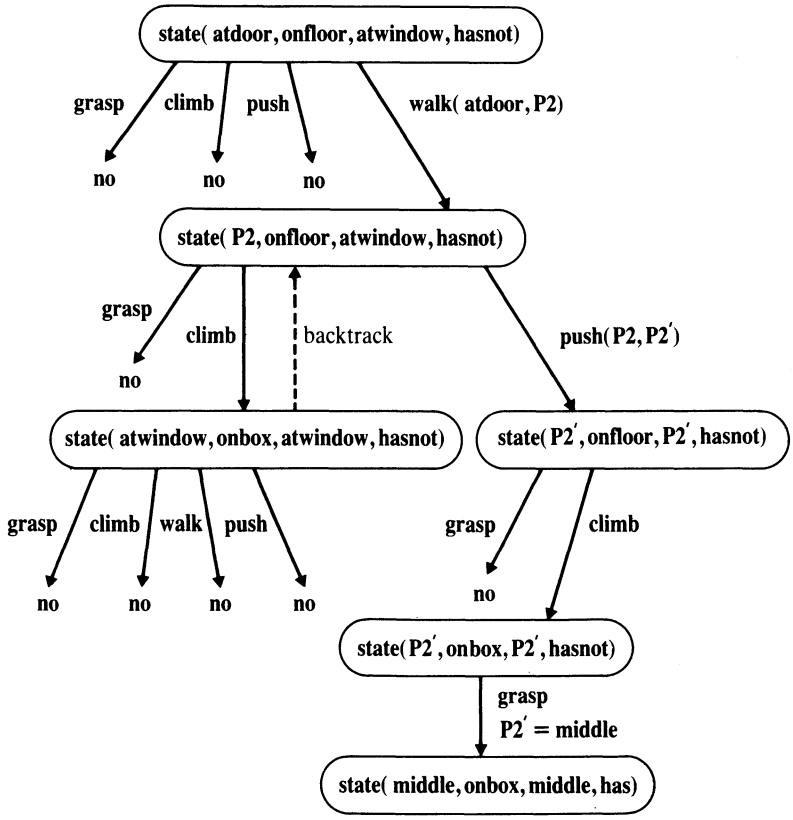


Figure 2.15 The monkey's search for the banana. The search starts at the top node and proceeds downwards, as indicated. Alternative moves are tried in the left-to-right order. Backtracking occurred once only.

without ever touching the box, or aimlessly push the box around. A more thorough investigation will reveal, as shown in the following section, that the ordering of clauses is, in the case of our program, in fact critical.

2.6 Order of clauses and goals

2.6.1 Danger of indefinite looping

Consider the following clause:

```
p :- p.
```

This says that 'p is true if p is true'. This is declaratively perfectly correct, but

procedurally is quite useless. In fact, such a clause can cause problems to Prolog. Consider the question:

```
?- p.
```

Using the clause above, the goal p is replaced by the same goal p; this will be in turn replaced by p, etc. In such a case Prolog will enter an infinite loop not noticing that no progress is being made.

This example is a simple way of getting Prolog to loop indefinitely. However, similar looping could have occurred in some of our previous example programs if we changed the order of clauses, or the order of goals in the clauses. It will be instructive to consider some examples.

In the monkey and banana program, the clauses about the **move** relation were ordered thus: grasp, climb, push, walk (perhaps 'unclimb' should be added for completeness). These clauses say that grasping is possible, climbing is possible, etc. According to the procedural semantics of Prolog, the order of clauses indicates that the monkey prefers grasping to climbing, climbing to pushing, etc. This order of preferences in fact helps the monkey to solve the problem. But what could happen if the order was different? Let us assume that the 'walk' clause appears first. The execution of our original goal of the previous section

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

would this time produce the following trace. The first four goal lists (with variables appropriately renamed) are the same as before:

(1) **canget(state(atdoor, onfloor, atwindow, hasnot))**

The second clause of **canget** ('can2') is applied, producing:

(2) **move(state(atdoor, onfloor, atwindow, hasnot), M′, S2′),**
canget(S2′)

By the move **walk(atdoor, P2′)** we get:

(3) **canget(state(P2′, onfloor, atwindow, hasnot))**

Using the clause 'can2' again the goal list becomes:

(4) **move(state(P2′, onfloor, atwindow, hasnot), M″, S2″),**
canget(S2″)

Now the difference occurs. The first clause whose head matches the first goal above is now 'walk' (and not 'climb' as before). The instantiation is

S2″ = state(P2″, onfloor, atwindow, hasnot). Therefore the goal list becomes:

(5) **canget(state(P2″, onfloor, atwindow, hasnot))**

Applying the clause 'can2' we obtain:

(6) **move(state(P2″, onfloor, atwindow, hasnot), M‴, S2‴),**
canget(S2‴)

Again, 'walk' is now tried first, producing:

(7) **canget(state(P2‴, onfloor, atwindow, hasnot))**

Let us now compare the goals (3), (5) and (7). They are the same apart from one variable; this variable is, in turn, P′, P″ and P‴. As we know, the success of a goal does not depend on particular names of variables in the goal. This means that from goal list (3) the execution trace shows no progress. We can see, in fact, that the same two clauses, 'can2' and 'walk', are used repetitively. The monkey walks around without ever trying to use the box. As there is no progress made this will (theoretically) go on for ever: Prolog will not realize that there is no point in continuing along this line.

This example shows Prolog trying to solve a problem in such a way that a solution is never reached, although a solution exists. Such situations are not unusual in Prolog programming. Infinite loops are, also, not unusual in other programming languages. What *is* unusual in comparison with other languages is that the declarative meaning of a Prolog program may be correct, but the program is at the same time procedurally incorrect in that it is not able to produce an answer to a question. In such cases Prolog may not be able to satisfy a goal because it tries to reach an answer by choosing a wrong path.

A natural question to ask at this point is: Can we not make some more substantial change to our program so as to drastically prevent any danger of looping? Or shall we always have to rely just on a suitable ordering of clauses and goals? As it turns out programs, especially large ones, would be too fragile if they just had to rely on some suitable ordering. There are several other methods that preclude infinite loops, and these are much more general and robust than the ordering method itself. These techniques will be used regularly later in the book, especially in those chapters that deal with path finding, problem solving and search.

2.6.2 Program variations through reordering of clauses and goals

Already in the example programs of Chapter 1 there was a latent danger of producing a cycling behaviour. Our program to specify the **predecessor** relation

in Chapter 1 was:

```
predecessor( X, Z) :-
  parent( X, Z).

predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

Let us analyze some variations of this program. All the variations will clearly have the same declarative meaning, but not the same procedural meaning.

```
% Four versions of the predecessor program

% The original version

pred1( X, Z) :-
  parent( X, Z).

pred1( X, Z) :-
  parent( X, Y),
  pred1( Y, Z).

% Variation a: swap clauses of the original version

pred2( X, Z) :-
  parent( X, Y),
  pred2( Y, Z).

pred2( X, Z) :-
  parent( X, Z).

% Variation b: swap goals in second clause of the original version

pred3( X, Z) :-
  parent( X, Z).

pred3( X, Z) :-
  pred3( X, Y),
  parent( Y, Z).

% Variation c: swap goals and clauses of the original version

pred4( X, Z) :-
  pred4( X, Y),
  parent( Y, Z).

pred4( X, Z) :-
  parent( X, Z).
```

Figure 2.16 Four versions of the **predecessor** program.

According to the declarative semantics of Prolog we can, without affecting the declarative meaning, change

(1) the order of clauses in the program, and
(2) the order of goals in the bodies of clauses.

The **predecessor** procedure consists of two clauses, and one of them has two goals in the body. There are, therefore, four variations of this program, all with the same declarative meaning. The four variations are obtained by

(1) swapping both clauses, and
(2) swapping the goals for each order of clauses.

The corresponding four procedures, called **pred1**, **pred2**, **pred3** and **pred4**, are shown in Figure 2.16.

There are important differences in the behaviour of these four declaratively equivalent procedures. To demonstrate these, consider the **parent** relation as shown in Figure 1.1 of Chapter 1. Now, what happens if we ask whether Tom is a predecessor of Pat using the four variations of the **predecessor** relation:

```
?- pred1( tom, pat).

yes

?- pred2( tom, pat).

yes

?- pred3( tom, pat).

yes

?- pred4( tom, pat).
```

In the last case Prolog cannot find the answer. This is manifested on the terminal by a Prolog message such as 'More core needed'.

Figure 1.11 in Chapter 1 showed the trace of **pred1** (in Chapter 1 called **predecessor**) produced for the above question. Figure 2.17 shows the corresponding traces for **pred2**, **pred3** and **pred4**. Figure 2.17(c) clearly shows that **pred4** is hopeless, and Figure 2.17(a) indicates that **pred2** is rather inefficient compared to **pred1**: **pred2** does much more searching and backtracking in the family tree.

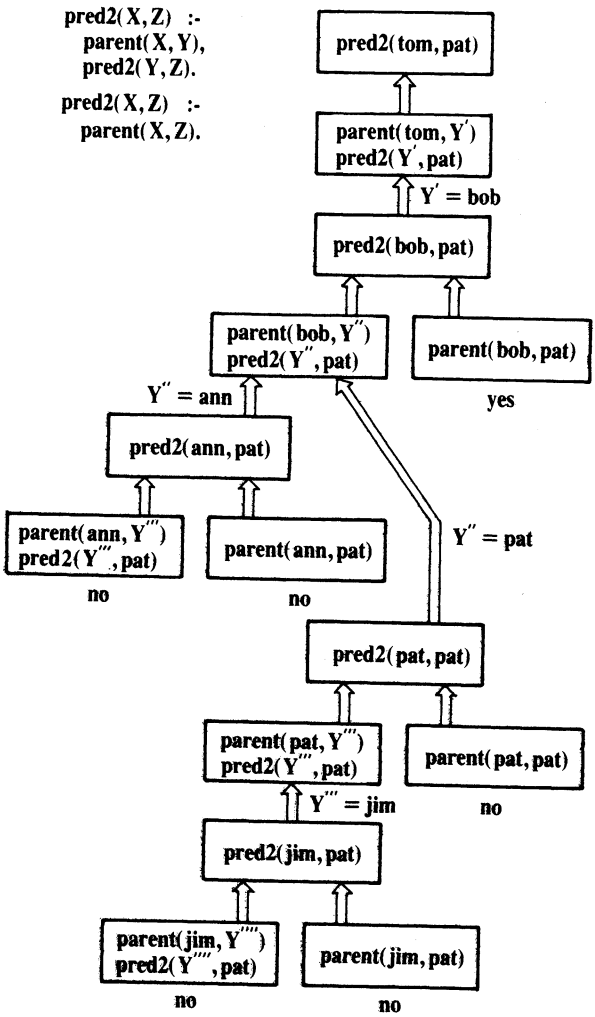
This comparison should remind us of a general practical heuristic in problem solving: it is often useful to try the simplest idea first. In our case, all the versions of the **predecessor** relation are based on two ideas:

- the simpler idea is to check whether the two arguments of the **predecessor** relation satisfy the **parent** relation;

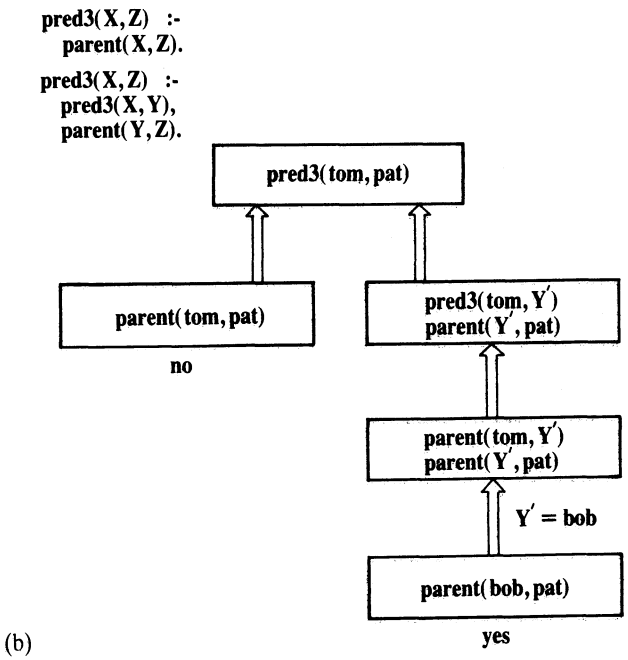
- the more complicated idea is to find somebody 'between' both people (somebody who is related to them by the **parent** and **predecessor** relations).

Of the four variations of the **predecessor** relation, **pred1** does simplest things first. On the contrary, **pred4** always tries complicated things first. **pred2** and **pred3** are in between the two extremes. Even without a detailed study of the execution traces, **pred1** should be clearly preferred merely on the grounds of the rule 'try simple things first'. This rule will be in general a useful guide in programming.

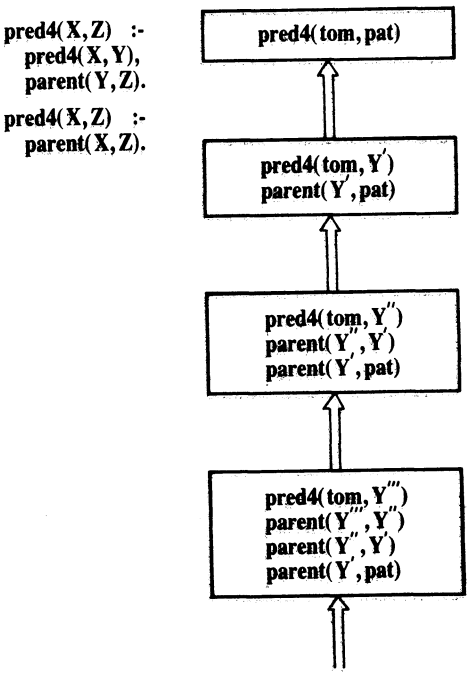
Our four variations of the **predecessor** procedure can be further compared by considering the question: What types of questions can particular variations answer, and what types can they not answer? It turns out that **pred1**



(a)



(b)



(c)

Figure 2.17 The behaviour of three formulations of the **predecessor** relation on the question: Is Tom a predecessor of Pat?

and **pred2** are both able to reach an answer for any type of question about predecessors; **pred4** can never reach an answer; and **pred3** sometimes can and sometimes cannot. One example in which **pred3** fails is:

```
?- pred3( liz, jim).
```

This question again brings the system into an infinite sequence of recursive calls. Thus **pred3** also cannot be considered procedurally correct.

2.6.3 Combining declarative and procedural views

The foregoing section has shown that the order of goals and clauses does matter. Furthermore, there are programs that are declaratively correct, but do not work in practice. Such discrepancies between the declarative and procedural meaning may appear annoying. One may argue: Why not simply forget about the declarative meaning. This argument can be brought to an extreme with a clause such as

```
predecessor( X, Z)  :-  predecessor( X, Z).
```

which is declaratively correct, but is completely useless as a working program.

The reason why we should not forget about the declarative meaning is that progress in programming technology is achieved by moving away from procedural details toward declarative aspects, which are normally easier to formulate and understand. The system itself, not the programmer, should carry the burden of filling in the procedural details. Prolog does help toward this end, although, as we have seen in this section, it only helps partially: it sometimes does work out the procedural details itself properly, and sometimes it does not. The philosophy adopted by many is that it is better to have at least *some* declarative meaning rather than *none* ('none' is the case in most other programming languages). The practical aspect of this view is that it is often rather easy to get a working program once we have a program that is declaratively correct. Consequently, a useful practical approach that often works is to concentrate on the declarative aspects of the problem, then test the resulting program on the computer, and if it fails procedurally try to rearrange the clauses and goals into a right order.

2.7 Remarks on the relation between Prolog and logic

Prolog is related to mathematical logic, so its syntax and meaning can be specified most concisely with references to logic. Prolog is indeed often defined that way. However, such an introduction to Prolog assumes that the reader is familiar with certain concepts of mathematical logic. These concepts are, on the other hand, certainly not necessary for understanding and using Prolog as a

programming tool, which is the aim of this book. For the reader who is especially interested in the relation between Prolog and logic, the following are some basic links to mathematical logic, together with some appropriate references.

Prolog's syntax is that of the *first-order predicate logic* formulas written in the so-called *clause form* (a form in which quantifiers are not explicitly written), and further restricted to *Horn clauses* only (clauses that have at most one positive literal). Clocksin and Mellish (1981) give a Prolog program that transforms a first-order predicate calculus formula into the clause form. The procedural meaning of Prolog is based on the *resolution principle* for mechanical theorem proving introduced by Robinson in his classical paper (1965). Prolog uses a special strategy for resolution theorem proving called SLD. An introduction to the first-order predicate calculus and resolution-based theorem proving can be found in Nilsson 1981. Mathematical questions regarding the properties of Prolog's procedural meaning with respect to logic are analyzed by Lloyd (1984).

Matching in Prolog corresponds to what is called *unification* in logic. However, we avoid the word unification because matching, for efficiency reasons in most Prolog systems, is implemented in a way that does not exactly correspond to unification (see Exercise 2.10). But from the practical point of view this approximation to unification is quite adequate.

Exercise

2.10 What happens if we ask Prolog:

```
?- X = f( X).
```

Should this request for matching succeed or fail? According to the definition of unification in logic this should fail, but what happens according to our definition of matching in Section 2.2? Try to explain why many Prolog implementations answer the question above with:

```
X = f(f(f(f(f(f(f(f(f(f(f(f(f( ...
```

Summary

So far we have covered a kind of basic Prolog, also called 'pure Prolog'. It is 'pure' because it corresponds closely to formal logic. Extensions whose aim is to tailor the language toward some practical needs will be covered later in the book (Chapters 3, 5, 6, 7). Important points of this chapter are:

- Simple objects in Prolog are *atoms*, *variables* and *numbers*. Structured objects, or *structures*, are used to represent objects that have several components.

- Structures are constructed by means of *functors*. Each functor is defined by its name and arity.
- The type of object is recognized entirely by its syntactic form.
- The *lexical scope* of variables is one clause. Thus the same variable name in two clauses means two different variables.
- Structures can be naturally pictured as trees. Prolog can be viewed as a language for processing trees.
- The *matching* operation takes two terms and tries to make them identical by instantiating the variables in both terms.
- Matching, if it succeeds, results in the *most general* instantiation of variables.
- The *declarative semantics* of Prolog defines whether a goal is true with respect to a given program, and if it is true, for what instantiation of variables it is true.
- A comma between goals means the conjunction of goals. A semicolon between goals means the disjunction of goals.
- The *procedural semantics* of Prolog is a procedure for satisfying a list of goals in the context of a given program. The procedure outputs the truth or falsity of the goal list and the corresponding instantiations of variables. The procedure automatically backtracks to examine alternatives.
- The declarative meaning of programs in 'pure Prolog' does not depend on the order of clauses and the order of goals in clauses.
- The procedural meaning does depend on the order of goals and clauses. Thus the order can affect the efficiency of the program; an unsuitable order may even lead to infinite recursive calls.
- Given a declaratively correct program, changing the order of clauses and goals can improve the program's efficiency while retaining its declarative correctness. Reordering is one method of preventing indefinite looping.
- There are other more general techniques, apart from reordering, to prevent indefinite looping and thereby make programs procedurally robust.
- Concepts discussed in this chapter are:

  data objects: atom, number, variable, structure
  term
  functor, arity of a functor
  principal functor of a term
  matching of terms
  most general instantiation
  declarative semantics
  instance of a clause, variant of a clause
  procedural semantics
  executing goals

References

Clocksin, W. F. and Mellish, C. S. (1981) *Programming in Prolog.* Springer-Verlag.

Lloyd, J. W. (1984) *Foundations of Logic Programming.* Springer-Verlag.

Nilsson, N. J. (1981) *Principles of Artificial Intelligence.* Tioga; also Springer-Verlag.

Robinson, A. J. (1965) A machine-oriented logic based on the resolution principle. *JACM* **12**: 23–41.

3 Lists, Operators, Arithmetic

In this chapter we will study a special notation for lists, one of the simplest and most useful structures, and some programs for typical operations on lists. We will also look at simple arithmetic and the operator notation which often improves the readability of programs. Basic Prolog of Chapter 2, extended with these three additions, becomes a convenient framework for writing interesting programs.

3.1 Representation of lists

The *list* is a simple data structure widely used in non-numeric programming. A list is a sequence of any number of items, such as **ann**, **tennis**, **tom**, **skiing**. Such a list can be written in Prolog as:

[ann, tennis, tom, skiing]

This is, however, only the external appearance of lists. As we have already seen in Chapter 2, all structured objects in Prolog are trees. Lists are no exception to this.

How can a list be represented as a standard Prolog object? We have to consider two cases: the list is either empty or non-empty. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list can be viewed as consisting of two things:

(1) the first item, called the *head* of the list;
(2) the remaining part of the list, called the *tail*.

For our example list

[ann, tennis, tom, skiing]

the head is **ann** and the tail is the list

[tennis, tom, skiing]

In general, the head can be anything (any Prolog object, for example, a tree or a variable); the tail has to be a list. The head and the tail are then combined into a structure by a special functor. The choice of this functor depends on the Prolog implementation; we will assume here that it is the dot:

.(Head, Tail)

Since **Tail** is in turn a list, it is either empty or it has its own head and tail. Therefore, to represent lists of any length no additional principle is needed. Our example list is then represented as the term:

.(ann, .(tennis, .(tom, .(skiing, []))))

Figure 3.1 shows the corresponding tree structure. Note that the empty list appears in the above term. This is because the one but last tail is a single item list:

[skiing]

This list has the empty list as its tail:

[skiing] = .(skiing, [])

This example shows how the general principle for structuring data objects in Prolog also applies to lists of any length. As our example also shows, the straightforward notation with dots and possibly deep nesting of subterms in the tail part can produce rather confusing expressions. This is the reason why Prolog provides the neater notation for lists, so that they can be written as sequences of items enclosed in square brackets. A programmer can use both notations, but the square bracket notation is, of course, normally preferred. We will be aware, however, that this is only a cosmetic improvement and that our lists will be internally represented as binary trees. When such terms are

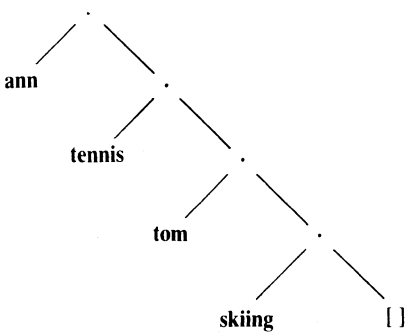


Figure 3.1 Tree representation of the list **[ann, tennis, tom, skiing]**.

output they will be automatically converted into their neater form. Thus the following conversation with Prolog is possible:

```
?- List1 = [a,b,c],
   List2 = .( a, .( b, .( c, [] ) ) ).

List1 = [a,b,c]
List2 = [a,b,c]

?- Hobbies1 = .( tennis, .( music, [] ) ),
   Hobbies2 = [ skiing, food],
   L = [ ann, Hobbies1, tom, Hobbies2].

Hobbies1 = [ tennis, music]
Hobbies2 = [ skiing, food]
L = [ ann, [tennis,music], tom, [skiing,food] ]
```

This example also reminds us that the elements of a list can be objects of any kind, in particular they can also be lists.

It is often practical to treat the whole tail as a single object. For example, let

```
L = [a,b,c]
```

Then we could write

```
Tail = [b,c]   and   L = .( a, Tail)
```

To express this in the square bracket notation for lists, Prolog provides another notational extension, the vertical bar, which separates the head and the tail:

```
L = [ a | Tail]
```

The vertical bar notation is in fact more general: we can list any number of elements followed by '|' and the list of remaining items. Thus alternative ways of writing the above list are:

```
[a,b,c] = [a | [b,c] ] = [a,b | [c] ] = [a,b,c | [] ]
```

To summarize:

- A list is a data structure that is either empty or consists of two parts: a *head* and a *tail*. The tail itself has to be a list.

- Lists are handled in Prolog as a special case of binary trees. For improved

readability Prolog provides a special notation for lists, thus accepting lists written as:

[Item1, Item2, ...]

or

[Head | Tail]

or

[Item1, Item2, ... | Others]

3.2 Some operations on lists

Lists can be used to represent sets although there is a difference: the order of elements in a set does not matter while the order of items in a list does; also, the same object can occur repeatedly in a list. Still, the most common operations on lists are similar to those on sets. Among them are:

- checking whether some object is an element of a list, which corresponds to checking for the set membership;
- concatenation of two lists, obtaining a third list, which corresponds to the union of sets;
- adding a new object to a list, or deleting some object from it.

In the remainder of this section we give programs for these and some other operations on lists.

3.2.1 Membership

Let us implement the membership relation as

member(X, L)

where X is an object and L is a list. The goal **member(X, L)** is true if X occurs in L. For example,

member(b, [a,b,c])

is true,

member(b, [a,[b,c]])

is not true, but

member([b,c], [a,[b,c]])

is true. The program for the membership relation can be based on the following observation:

X is a member of L if either
(1) X is the head of L, or
(2) X is a member of the tail of L.

This can be written in two clauses, the first is a simple fact and the second is a rule:

```
member( X, [X | Tail] ).

member( X, [Head | Tail] ) :-
  member( X, Tail).
```

3.2.2 Concatenation

For concatenating lists we will define the relation

conc(L1, L2, L3)

Here L1 and L2 are two lists, and L3 is their concatenation. For example

conc([a,b], [c,d], [a,b,c,d])

is true, but

conc([a,b], [c,d], [a,b,a,c,d])

is false. In the definition of **conc** we will have again two cases, depending on the first argument, L1:

(1) If the first argument is the empty list then the second and the third arguments must be the same list (call it L); this is expressed by the following Prolog fact:

```
conc( [], L, L).
```

(2) If the first argument of **conc** is a non-empty list then it has a head and a tail and must look like this:

[X | L1]

Figure 3.2 illustrates the concatenation of **[X | L1]** and some list L2. The result of the concatenation is the list **[X | L3]** where L3 is the concatenation of L1 and L2. In Prolog this is written as:

```
conc( [X | L1], L2, [X | L3] ) :-
  conc( L1, L2, L3).
```

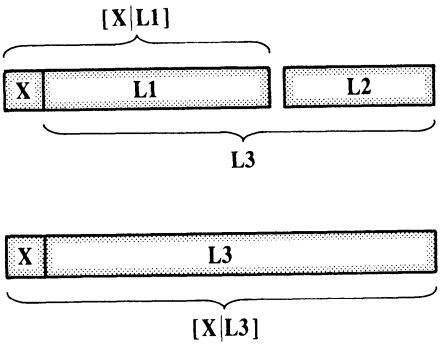


Figure 3.2 Concatenation of lists.

This program can now be used for concatenating given lists, for example:

```
?- conc( [a,b,c], [1,2,3], L).

L = [a,b,c,1,2,3]

?- conc( [a,[b,c],d], [a,[],b], L).

L = [a, [b,c], d, a, [], b]
```

Although the **conc** program looks rather simple it can be used flexibly in many other ways. For example, we can use **conc** in the inverse direction for *decomposing* a given list into two lists, as follows:

```
?- conc( L1, L2, [a,b,c] ).

L1 = []
L2 = [a,b,c];

L1 = [a]
L2 = [b,c];

L1 = [a,b]
L2 = [c];

L1 = [a,b,c]
L2 = [];

no
```

It is possible to decompose the list **[a,b,c]** in four ways, all of which were found by our program through backtracking.

We can also use our program to look for a certain pattern in a list. For

example, we can find the months that precede and the months that follow a given month, as in the following goal:

```
?- conc( Before, [may | After],
         [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec].
```

Further we can find the immediate predecessor and the immediate successor of May by asking:

```
?- conc( _, [Month1,may,Month2 | _],
         [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Month1 = apr
Month2 = jun
```

Further still, we can, for example, delete from some list, L1, everything that follows three successive occurrences of z in L1 together with the three z's. For example:

```
?- L1 = [a,b,z,z,c,z,z,z,d,e],
   conc( L2, [z,z,z | _], L1).

L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

We have already programmed the membership relation. Using **conc**, however, the membership relation could be elegantly programmed by the clause:

```
member1( X, L) :-
  conc( L1, [X | L2], L).
```

This clause says: X is a member of list L if L can be decomposed into two lists so that the second one has X as its head. Of course, **member1** defines the same relation as **member**. We have just used a different name to distinguish between the two implementations. Note that the above clause can be written using anonymous variables as:

```
member1( X, L) :-
  conc( _, [X | _], L).
```

It is interesting to compare both implementations of the membership relation, **member** and **member1**. **member** has a rather straightforward procedural meaning, which is as follows:

To check whether some X is a member of some list L:

(1) first check whether the head of L is equal to X, and then

(2) check whether X is a member of the tail of L.

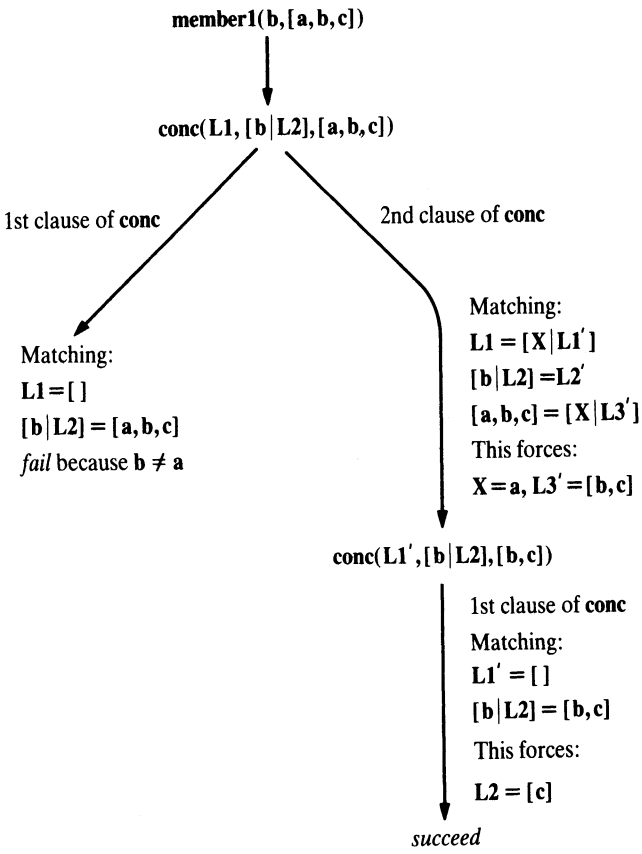


Figure 3.3 Procedure **member1** finds an item in a given list by sequentially searching the list.

On the other hand, the declarative meaning of **member1** is straightforward, but its procedural meaning is not so obvious. An interesting exercise is to find how **member1** actually computes something. An example execution trace will give some idea: let us consider the question:

```
?- member1( b, [a,b,c] ).
```

Figure 3.3 shows the execution trace. From the trace we can infer that **member1** behaves similarly to **member**. It scans the list, element by element, until the item in question is found or the list is exhausted.

Exercises

3.1 (a) Write a goal, using **conc**, to delete the last three elements from a list L producing another list L1. Hint: L is the concatenation of L1 and a three-element list.

(b) Write a sequence of goals to delete the first three elements and the last three elements from a list L producing list L2.

3.2 Define the relation

last(Item, List)

so that **Item** is the last element of a list **List**. Write two versions: (a) using the **conc** relation, (b) without **conc**.

3.2.3 Adding an item

To add an item to a list, it is easiest to put the new item in front of the list so that it becomes the new head. If X is the new item and the list to which X is added is L then the resulting list is simply

[X | L]

So we actually need no procedure for adding a new element in front of the list. Nevertheless, if we want to define such a procedure explicitly, it can be written as the fact:

```
add( X, L, [X | L] ).
```

3.2.4 Deleting an item

Deleting an item, X, from a list, L, can be programmed as a relation

del(X, L, L1)

where L1 is equal to the list L with the item X removed. The **del** relation can be defined similarly to the membership relation. We have, again, two cases:

(1) If X is the head of the list then the result after the deletion is the tail of the list.

(2) If X is in the tail then it is deleted from there.

```
del( X, [X | Tail], Tail).

del( X, [Y | Tail], [Y | Tail1] ) :-
  del( X, Tail, Tail1).
```

Like **member**, **del** is also non-deterministic in nature. If there are several occurrences of X in the list then **del** will be able to delete anyone of them by backtracking. Of course, each alternative execution will only delete one occur-

rence of X, leaving the others untouched. For example:

```
?- del( a, [a,b,a,a], L).

L = [b,a,a];

L = [a,b,a];

L = [a,b,a];

no
```

del will fail if the list does not contain the item to be deleted.

del can also be used in the inverse direction, to add an item to a list by inserting the new item anywhere in the list. For example, if we want to insert **a** at any place in the list **[1,2,3]** then we can do this by asking the question: What is L such that after deleting **a** from L we obtain **[1,2,3]**?

```
?- del( a, L, [1,2,3] ).

L = [a,1,2,3];

L = [1,a,2,3];

L = [1,2,a,3];

L = [1,2,3,a];

no
```

In general, the operation of inserting X at any place in some list **List** giving **BiggerList** can be defined by the clause:

```
insert( X, List, BiggerList) :-
  del( X, BiggerList, List).
```

In **member1** we elegantly implemented the membership relation by using **conc**. We can also use **del** to test for membership. The idea is simple: some X is a member of **List** if X can be deleted from **List**:

```
member2( X, List) :-
  del( X, List, _).
```

3.2.5 Sublist

Let us now consider the **sublist** relation. This relation has two arguments, a list L and a list S such that S occurs within L as its sublist. So

```
sublist( [c,d,e], [a,b,c,d,e,f] )
```

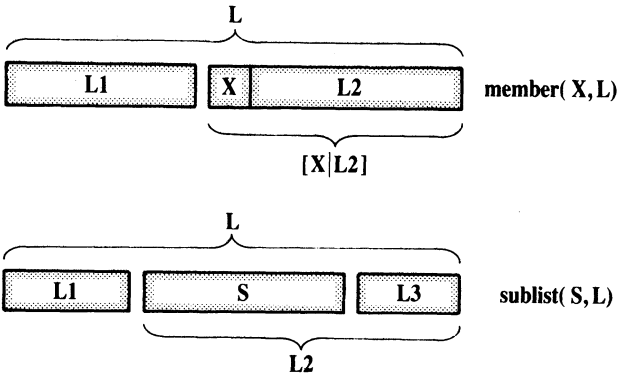


Figure 3.4 The **member** and **sublist** relations.

is true, but

```
sublist( [c,e], [a,b,c,d,e,f] )
```

is not. The Prolog program for **sublist** can be based on the same idea as **member1**, only this time the relation is more general (see Figure 3.4). Accordingly, the relation can be formulated as:

S is a sublist of L if
(1) L can be decomposed into two lists, L1 and L2, and
(2) L2 can be decomposed into two lists, S and some L3.

As we have seen before, the **conc** relation can be used for decomposing lists. So the above formulation can be expressed in Prolog as:

```
sublist( S, L) :-
  conc( L1, L2, L),
  conc( S, L3, L2).
```

Of course, the **sublist** procedure can be used flexibly in several ways. Although it was designed to check if some list occurs as a sublist within another list it can also be used, for example, to find all sublists of a given list:

```
?- sublist( S, [a,b,c] ).

S = [];

S = [a];

S = [a,b];

S = [a,b,c];

S = [b];

...
```

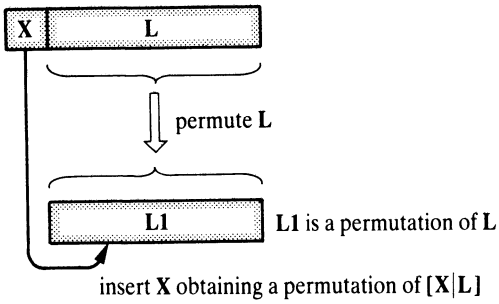


Figure 3.5 One way of constructing a permutation of the list [X | L].

3.2.6 Permutations

Sometimes it is useful to generate permutations of a given list. To this end, we will define the **permutation** relation with two arguments. The arguments are two lists such that one is a permutation of the other. The intention is to generate permutations of a list through backtracking using the **permutation** procedure, as in the following example:

```
?- permutation( [a,b,c], P).

P = [a,b,c];

P = [a,c,b];

P = [b,a,c];

...
```

The program for **permutation** can be, again, based on the consideration of two cases, depending on the first list:

(1) If the first list is empty then the second list must also be empty.

(2) If the first list is not empty then it has the form [X | L], and a permutation of such a list can be constructed as shown in Figure 3.5: first permute L obtaining L1 and then insert X at any position into L1.

Two Prolog clauses that correspond to these two cases are:

```
permutation( [], [] ).

permutation( [X | L], P) :-
  permutation( L, L1),
  insert( X, L1, P).
```

One alternative to this program would be to delete an element, X, from the first list, permute the rest of it obtaining a list P, and then add X in front of P. The corresponding program is:

```
permutation2( [], [] ).

permutation2( L, [X | P] ) :-
  del( X, L, L1),
  permutation2( L1, P).
```

It is instructive to do some experiments with our permutation programs. Its normal use would be something like this:

```
?- permutation( [red,blue,green], P).
```

This would result in all six permutations, as intended:

```
P = [ red, blue, green];

P = [ red, green, blue];

P = [ blue, red, green];

P = [ blue, green, red];

P = [ green, red, blue];

P = [ green, blue, red];

no
```

Another attempt to use **permutation** is:

```
?- permutation( L, [a,b,c] ).
```

Our first version, **permutation**, will now instantiate L successfully to all six permutations. If the user then requests more solutions, the program would never answer 'no' because it would get into an infinite loop trying to find another permutation when there is none. Our second version, **permutation2**, will in this case find only the first (identical) permutation and then immediately get into an infinite loop. Thus, some care is necessary when using these **permutation** relations.

Exercises

3.3 Define two predicates

evenlength(List) and **oddlength(List)**

so that they are true if their argument is a list of even or odd length

respectively. For example, the list **[a,b,c,d]** is 'evenlength' and **[a,b,c]** is 'oddlength'.

3.4 Define the relation

reverse(List, ReversedList)

that reverses lists. For example, **reverse([a,b,c,d], [d,c,b,a]).**

3.5 Define the predicate **palindrome(List).** A list is a palindrome if it reads the same in the forward and in the backward direction. For example, **[m,a,d,a,m].**

3.6 Define the relation

shift(List1, List2)

so that **List2** is **List1** 'shifted rotationally' by one element to the left. For example,

```
?- shift( [1,2,3,4,5], L1),
   shift( L1, L2).
```

produces:

```
L1 = [2,3,4,5,1]
L2 = [3,4,5,1,2]
```

3.7 Define the relation

translate(List1, List2)

to translate a list of numbers between 0 and 9 to a list of the corresponding words. For example:

translate([3,5,1,3], [three,five,one,three])

Use the following as an auxiliary relation:

means(0, zero). means(1, one). means(2, two). ...

3.8 Define the relation

subset(Set, Subset)

where **Set** and **Subset** are two lists representing two sets. We would like to be able to use this relation not only to check for the subset relation, but also to generate all possible subsets of a given set. For example:

```
?- subset( [a,b,c], S).

S = [a,b,c];
S = [b,c];
S = [c];
```

```
S = [];
S = [a,c];
S = [a];
...
```

3.9 Define the relation

dividelist(List, List1, List2)

so that the elements of **List** are partitioned between **List1** and **List2**, and **List1** and **List2** are of approximately the same length. For example, **partition([a,b,c,d,e], [a,c,e], [b,d]).**

3.10 Rewrite the monkey and banana program of Chapter 2 as the relation

canget(State, Actions)

to answer not just 'yes' or 'no', but to produce a sequence of monkey's actions that lead to success. Let **Actions** be such a sequence represented as a list of moves:

Actions = [walk(door,window), push(window,middle), climb, grasp]

3.11 Define the relation

flatten(List, FlatList)

where **List** can be a list of lists, and **FlatList** is **List** 'flattened' so that the elements of **List**'s sublists (or sub-sublists) are reorganized as one plain list. For example:

```
?- flatten( [a,b,[c,d],[],[[[e]]],f], L).
L = [a,b,c,d,e,f]
```

3.3 Operator notation

In mathematics we are used to writing expressions like

$$2*a + b*c$$

where + and * are operators, and 2, *a*, *b*, *c* are arguments. In particular, + and * are said to be *infix* operators because they appear *between* the two arguments. Such expressions can be represented as trees, as in Figure 3.6, and can be written as Prolog terms with + and * as functors:

```
+( *(2,a), *(b,c) )
```

Since we would normally prefer to have such expressions written in the usual,

infix style with operators, Prolog caters for this notational convenience. Prolog will therefore accept our expression written simply as:

2*a + b*c

This will be, however, only the external representation of this object, which will be automatically converted into the usual form of Prolog terms. Such a term will be output for the user, again, in its external, infix form.

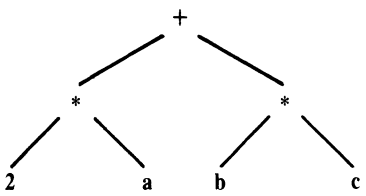


Figure 3.6 Tree representation of the expression **2*a + b*c**.

Thus expressions are dealt with in Prolog merely as a notational extension and no new principle for structuring data objects is involved. If we write **a + b**, Prolog will handle it exactly as if it had been written **+(a,b)**. In order that Prolog properly understands expressions such as **a + b*c**, Prolog has to know that * binds stronger than +. We say that + has higher precedence than *. So the precedence of operators decides what is the correct interpretation of expressions. For example, the expression **a + b*c** can be, in principle, understood either as

+(a, *(b,c))

or as

***(+(a,b), c)**

The general rule is that the operator with the highest precedence is the principal functor of the term. If expressions containing + and * are to be understood according to our normal conventions, then + has to have a higher precedence than *. Then the expression **a + b*c** means the same as **a + (b*c)**. If another interpretation is intended, then it has to be explicitly indicated by parentheses – for example, **(a + b)*c**.

A programmer can define his or her own operators. So, for example, we can define the atoms **has** and **supports** as infix operators and then write in the program facts like:

peter has information.
floor supports table.

These facts are exactly equivalent to:

```
has( peter, information).
supports( floor, table).
```

A programmer can define new operators by inserting into the program special kinds of clauses, sometimes called *directives*, which act as operator definitions. An operator definition must appear in the program before any expression containing that operator. For our example, the operator **has** can be properly defined by the directive:

```
:- op( 600, xfx, has).
```

This tells Prolog that we want to use 'has' as an operator, whose precedence is 600 and its type is 'xfx', which is a kind of infix operator. The form of the specifier 'xfx' suggests that the operator, denoted by 'f', is between the two arguments denoted by 'x'.

Notice that operator definitions do not specify any operation or action. In principle, *no operation on data is associated with an operator* (except in very special cases). Operators are normally used, as functors, only to combine objects into structures and not to invoke actions on data, although the word 'operator' appears to suggest an action.

Operator names are atoms, and their precedence must be in some range which depends on the implementation. We will assume that the range is between 1 and 1200.

There are three groups of operator types which are indicated by type specifiers such as **xfx**. The three groups are:

(1) infix operators of three types:

xfx xfy yfx

(2) prefix operators of two types:

fx fy

(3) postfix operators of two types:

xf yf

The specifiers are chosen so as to reflect the structure of the expression where 'f' represents the operator and 'x' and 'y' represent arguments. An 'f' appearing between the arguments indicates that the operator is infix. The prefix and postfix specifiers have only one argument, which follows or precedes the operator respectively.

There is a difference between 'x' and 'y'. To explain this we need to introduce the notion of the *precedence of argument*. If an argument is enclosed in parentheses or it is an unstructured object then its precedence is 0; if an argument is a structure then its precedence is equal to the precedence of its

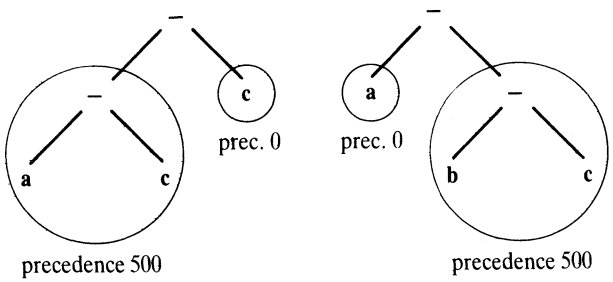


Figure 3.7 Two interpretations of the expression **a – b – c** assuming that '–' has precedence 500. If '–' is of type **yfx**, then interpretation 2 is invalid because the precedence of **b – c** is not less than the precedence of '–'.

principal functor. 'x' represents an argument whose precedence must be strictly lower than that of the operator. 'y' represents an argument whose precedence is lower or equal to that of the operator.

These rules help to disambiguate expressions with several operators of the same precedence. For example, the expression

a – b – c

is normally understood as **(a – b) – c**, and not as **a – (b – c)**. To achieve the normal interpretation the operator '–' has to be defined as **yfx**. Figure 3.7 shows why the second interpretation is then ruled out.

As another example consider the prefix operator **not**. If **not** is defined as **fy** then the expression

not not p

is legal; but if **not** is defined as **fx** then this expression is illegal because the argument to the first **not** is **not p**, which has the same precedence as **not** itself. In this case the expression has to be written with parentheses:

not (not p)

For convenience, some operators are predefined in the Prolog system so that they can be readily used and no definition is needed for them. What these operators are and what their precedences are depends on the implementation of Prolog. We will assume that this set of 'standard' operators is as if defined by the clauses in Figure 3.8. As Figure 3.8 also shows, several operators can be declared by one clause if they all have the same precedence and if they are all of the same type. In this case the operators' names are written as a list.

The use of operators can greatly improve the readability of programs. As an example let us assume that we are writing a program for manipulating

```
:- op( 1200, xfx, ':-').
:- op( 1200, fx, [ :-, ?-] ).
:- op( 1100, xfy, ';').
:- op( 1000, xfy, ',').
:- op( 700, xfx, [ =, is, < , > , =< , >= , == , =\=, \==, =:=] ).
:- op( 500, yfx, [ + , -] ).
:- op( 500, fx, [ + , -, not] ).
:- op( 400, yfx, [ *, /, div] ).
:- op( 300, xfx, mod).
```

Figure 3.8 A set of predefined operators.

Boolean expressions. In such a program we may want to state, for example, one of de Morgan's equivalence theorems, which can in mathematics be written as:

$$\sim(A \,\&\, B) \quad <===> \quad \sim A \vee \sim B$$

One way to state this in Prolog is by the clause:

```
equivalence( not( and( A, B)), or( not( A), not( B) ) ).
```

However, it is in general a good programming practice to try to retain as much resemblance as possible between the original problem notation and the notation used in the program. In our example, this can be achieved almost completely by using operators. A suitable set of operators for our purpose can be defined as:

```
:- op( 800, xfx, <===>).
:- op( 700, xfy, v).
:- op( 600, xfy, &).
:- op( 500, fy, ~).
```

Now the de Morgan's theorem can be written as the fact:

```
~(A & B) <===> ~A v ~B.
```

According to our specification of operators above, this term is understood as shown in Figure 3.9.

To summarize:

- The readability of programs can be often improved by using the operator notation. Operators can be infix, prefix or postfix.

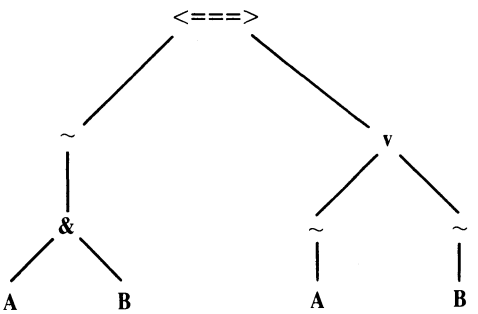


Figure 3.9 Interpretation of the term ~(A & B) <===> ~A v ~B.

- In principle, no operation on data is associated with an operator except in special cases. Operator definitions do not define any action, they only introduce new notation. Operators, as functors, only hold together components of structures.
- A programmer can define his or her own operators. Each operator is defined by its name, precedence and type.
- The precedence is an integer within some range, say between 1 and 1200. The operator with the highest precedence in the expression is the principal functor of the expression. Operators with lowest precedence bind strongest.
- The type of the operator depends on two things: (1) the position of the operator with respect to the arguments, and (2) the precedence of the arguments compared to the precedence of the operator itself. In a specifier like **xfy**, **x** indicates an argument whose precedence is strictly lower than that of the operator; **y** indicates an argument whose precedence is less than or equal to that of the operator.

Exercises

3.12 Assuming the operator definitions

```
:- op( 300, xfx, plays).
:- op( 200, xfy, and).
```

then the following two terms are syntactically legal objects:

Term1 = jimmy plays football and squash
Term2 = susan plays tennis and basketball and volleyball

How are these terms understood by Prolog? What are their principal functors and what is their structure?

3.13 Suggest an appropriate definition of operators ('was', 'of', 'the') to be able to write clauses like

```
diana was the secretary of the department.
```

and then ask Prolog:

```
?- Who was the secretary of the department.
Who = diana

?- diana was What.
What = the secretary of the department
```

3.14 Consider the program:

```
t( 0+1, 1+0).
t( X+0+1, X+1+0).
t( X+1+1, Z) :-
  t( X+1, X1),
  t( X1+1, Z).
```

How will this program answer the following questions if '+' is an infix operator of type **yfx** (as usual):

(a) `?- t( 0+1, A).`

(b) `?- t( 0+1+1, B).`

(c) `?- t( 1+0+1+1+1, C).`

(d) `?- t( D, 1+1+1+0).`

3.15 In the previous section, relations involving lists were written as:

```
member( Element, List),
conc( List1, List2, List3),
del( Element, List, NewList), ...
```

Suppose that we would prefer to write these relations as:

```
Element in List,
concatenating List1 and List2 gives List3,
deleting Element from List gives NewList, ...
```

Define 'in', 'concatenating', 'and', etc. as operators to make this possible. Also, redefine the corresponding procedures.

3.4 Arithmetic

Prolog is mainly a language for symbolic computation where the need for numerical calculation is comparatively modest. Accordingly, the means for

doing arithmetic in Prolog are also rather simple. Some of the predefined operators can be used for basic arithmetic operations. These are:

+	addition
–	subtraction
*	multiplication
/	division
mod	modulo, the remainder of integer division

Notice that this is an exceptional case in which an operator may in fact invoke an operation. But even in such cases an additional indication to perform the action will be necessary. Prolog knows how to carry out the calculation denoted by these operators, but this is not entirely sufficient for direct use. The following question is a naive attempt to request arithmetic computation:

```
?- X = 1 + 2.
```

Prolog will 'quietly' answer

```
X = 1 + 2
```

and not X = 3 as we might possibly expect. The reason is simple: the expression 1 + 2 merely denotes a Prolog term where + is the functor and 1 and 2 are its arguments. There is nothing in the above goal to force Prolog to actually activate the addition operation. A special predefined operator, **is**, is provided to circumvent this problem. The **is** operator will force evaluation. So the right way to invoke arithmetic is:

```
?- X is 1 + 2.
```

Now the answer will be:

```
X = 3
```

The addition here was carried out by a special procedure that is associated with the operator +. We call such procedures *built-in procedures*.

There is no generally agreed notational convention for arithmetic in Prolog, so different implementations of Prolog may use somewhat different notations. For example, the '/' operator may denote integer division or real division, depending on the implementation. In this book, we will assume that '/' denotes real division, and that the **div** operator denotes integer division. Accordingly, the question

```
?- X is 3/2,
   Y is 3 div 2.
```

is answered by

```
X = 1.5
Y = 1
```

The left argument of the **is** operator is a simple object. The right argument is an arithmetic expression composed of arithmetic operators, numbers and variables. Since the **is** operator will force the evaluation, all the variables in the expression must already be instantiated to numbers at the time of execution of this goal. The precedence of the predefined arithmetic operators (see Figure 3.8) is such that the associativity of arguments with operators is the same as normally in mathematics. Parentheses can be used to indicate different associations. Note that +, –, *, / and **div** are defined as **yfx**, which means that evaluation is carried out from left to right. For example,

```
X is 5 - 2 - 1
```

is interpreted as

```
X is (5 - 2) - 1
```

Arithmetic is also involved when *comparing* numerical values. We can, for example, test whether the product of 277 and 37 is greater than 10000 by the goal:

```
?- 277 * 37 > 10000.
yes
```

Note that, similarly to **is**, the '>' operator also forces the evaluation.

Suppose that we have in the program a relation **born** that relates the names of people with their birth years. Then we can retrieve the names of people born between 1950 and 1960 inclusive with the following question:

```
?- born( Name, Year),
   Year >= 1950,
   Year =< 1960.
```

The comparison operators are as follows:

X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the values of X and Y are equal
X =\= Y	the values of X and Y are not equal

Notice the difference between the matching operators '=' and '=:='; for example, in the goals X = Y and X =:= Y. The first goal will cause the matching of the objects X and Y, and will, if X and Y match, possibly instantiate some variables in X and Y. There will be no evaluation. On the other hand, X =:= Y causes the arithmetic evaluation and cannot cause any instantiation of variables. These differences are illustrated by the following examples:

```
?- 1 + 2 =:= 2 + 1.

yes

?- 1 + 2 = 2 + 1.

no

?- 1 + A = B + 2.

A = 2
B = 1
```

Let us further illustrate the use of arithmetic operations by two simple examples. The first involves computing the greatest common divisor; the second, counting the items in a list.

Given two positive integers, X and Y, their greatest common divisor, D, can be found according to three cases:

(1) If X and Y are equal then D is equal to X.

(2) If X < Y then D is equal to the greatest common divisor of X and the difference Y – X.

(3) If Y < X then do the same as in case (2) with X and Y interchanged.

It can be easily shown by an example that these three rules actually work. Choosing, for example, X = 20 and Y = 25, the above rules would give D = 5 after a sequence of subtractions.

These rules can be formulated into a Prolog program by defining a three-argument relation, say

```
gcd( X, Y, D)
```

The three rules are then expressed as three clauses, as follows:

```
gcd( X, X, X).

gcd( X, Y, D) :-
  X < Y,
  Y1 is Y - X,
  gcd( X, Y1, D).
```

```
gcd( X, Y, D) :-
  Y < X,
  gcd( Y, X, D).
```

Of course, the last goal in the third clause could be equivalently replaced by the two goals:

```
X1 is X - Y,
gcd( X1, Y, D)
```

Our next example involves counting, which usually requires some arithmetic. An example of such a task is to establish the length of a list; that is, we have to count the items in the list. Let us define the procedure

```
length( List, N)
```

which will count the elements in a list **List** and instantiate N to their number. As was the case with our previous relations involving lists, it is useful to consider two cases:

(1) If the list is empty then its length is 0.

(2) If the list is not empty then **List = [Head | Tail]**; then its length is equal to 1 plus the length of the tail **Tail**.

These two cases correspond to the following program:

```
length( [], 0).

length( [_ | Tail], N) :-
  length( Tail, N1),
  N is 1 + N1.
```

An application of **length** can be:

```
?- length( [a,b,[c,d],e], N).

N = 4
```

Note that in the second clause of **length**, the two goals of the body cannot be swapped. The reason for this is that N1 has to be instantiated before the goal

```
N is 1 + N1
```

can be processed. With the built-in procedure **is**, a relation has been introduced that is sensitive to the order of processing and therefore the procedural considerations have become vital.

It is interesting to see what happens if we try to program the **length** relation without the use of **is**. Such an attempt can be:

```
length1( [], 0).

length1( [_ | Tail], N) :-
  length1( Tail, N1),
  N = 1 + N1.
```

Now the goal

```
?- length1( [a,b,[c,d],e], N).
```

will produce the answer:

```
N = 1+(1+(1+(1+0))).
```

The addition was never explicitly forced and was therefore not carried out at all. But in **length1** we can, unlike in **length**, swap the goals in the second clause:

```
length1( [_ | Tail], N) :-
  N = 1 + N1,
  length1( Tail, N1).
```

This version of **length1** will produce the same result as the original version. It can also be written shorter, as follows,

```
length1( [_ | Tail], 1 + N) :-
  length1( Tail, N).
```

still producing the same result. We can, however, use **length1** to find the number of elements in a list as follows:

```
?- length1( [a,b,c], N), Length is N.

N = 1+(1+(1+0))
Length = 3
```

To summarize:

- Built-in procedures can be used for doing arithmetic.
- Arithmetic operations have to be explicitly requested by the built-in procedure **is**. There are built-in procedures associated with the predefined operators +, −, *, /, **div** and **mod**.
- At the time that evaluation is carried out, all arguments must be already instantiated to numbers.

- The values of arithmetic expressions can be compared by operators such as <, =<, etc. These operators force the evaluation of their arguments.

Exercises

3.16 Define the relation

max(X, Y, Max)

so that **Max** is the greater of two numbers X and Y.

3.17 Define the predicate

maxlist(List, Max)

so that **Max** is the greatest number in the list of numbers **List**.

3.18 Define the predicate

sumlist(List, Sum)

so that **Sum** is the sum of a given list of numbers **List**.

3.19 Define the predicate

ordered(List)

which is true if **List** is an ordered list of numbers. For example, **ordered([1,5,6,6,9,12])**.

3.20 Define the predicate

subsum(Set, Sum, SubSet)

so that **Set** is a list of numbers, **SubSet** is a subset of these numbers, and the sum of the numbers in **SubSet** is **Sum**. For example:

```
?- subsum( [1,2,5,3,2], 5, Sub).

Sub = [1,2,2];
Sub = [2,3];
Sub = [5];
...
```

3.21 Define the procedure

between(N1, N2, X)

which, for two given integers N1 and N2, generates through backtracking all the integers X that satisfy the constraint $N1 \leqq X \leqq N2$.

3.22 Define the operators 'if', 'then', 'else' and ':=' so that the following becomes a legal term:

```
if X > Y then Z := X else Z := Y
```

Choose the precedences so that 'if' will be the principal functor. Then define the relation 'if' as a small interpreter for a kind of 'if-then-else' statement of the form

```
if Val1 > Val2 then Var := Val3 else Var := Val4
```

where **Val1**, **Val2**, **Val3** and **Val4** are numbers (or variables instantiated to numbers) and **Var** is a variable. The meaning of the 'if' relation should be: if the value of **Val1** is greater than the value of **Val2** then **Var** is instantiated to **Val3**, otherwise to **Val4**. Here is an example of the use of this interpreter:

```
?- X = 2, Y = 3,
   Val2 is 2*X,
   Val4 is 4*X,
   if Y > Val2 then Z := Y else Z := Val4,
   if Z > 5 then W := 1 else W := 0.

X = 2
Y = 3
Z = 8
W = 1
```

Summary

- The list is a frequently used structure. It is either empty or consists of a *head* and a *tail* which is a list as well. Prolog provides a special notation for lists.
- Common operations on lists, programmed in this chapter, are: list membership, concatenation, adding an item, deleting an item, sublist.
- The *operator notation* allows the programmer to tailor the syntax of programs toward particular needs. Using operators the readability of programs can be greatly improved.
- New operators are defined by the directive **op**, stating the name of an operator, its type and precedence.
- In principle, there is no operation associated with an operator; operators are merely a syntactic device providing an alternative syntax for terms.
- Arithmetic is done by built-in procedures. Evaluation of an arithmetic expression is forced by the procedure **is** and by the comparison predicates <, =<, etc.

- Concepts introduced in this chapter are:

  list, head of list, tail of list
  list notation
  operators, operator notation
  infix, prefix and suffix operators
  precedence of an operator
  arithmetic built-in procedures

4 Using Structures: Example Programs

Data structures, with matching, backtracking and arithmetic, are a powerful programming tool. In this chapter we will develop the skill of using this tool through programming examples: retrieving structured information from a database, simulating a non-deterministic automaton, travel planning and eight queens on the chessboard. We will also see how the principle of data abstraction can be carried out in Prolog.

4.1 Retrieving structured information from a database

This exercise develops the skill of representing and manipulating structured data objects. It also illustrates Prolog as a natural database query language.

A database can be naturally represented in Prolog as a set of facts. For example, a database about families can be represented so that each family is described by one clause. Figure 4.1 shows how the information about each family can be structured. Each family has three components: husband, wife and children. As the number of children varies from family to family the children are represented by a list that is capable of accommodating any number of items. Each person is, in turn, represented by a structure of four components: name, surname, date of birth, job. The job information is 'unemployed',

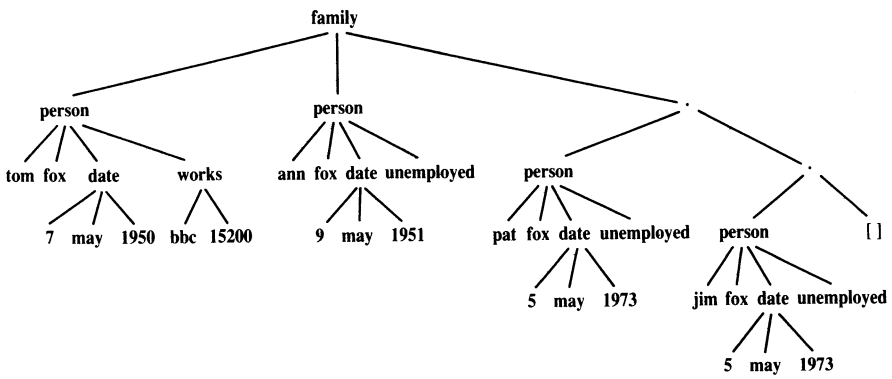


Figure 4.1 Structuring information about the family.

or it specifies the working organization and salary. The family of Figure 4.1 can be stored in the database by the clause:

```
family(
  person( tom, fox, date(7,may,1950), works(bbc,15200) ),
  person( ann, fox, date(9,may,1951), unemployed),
  [ person( pat, fox, date(5,may,1973), unemployed),
    person( jim, fox, date(5,may,1973), unemployed) ] ).
```

Our database would then be comprised of a sequence of facts like this describing all families that are of interest to our program.

Prolog is, in fact, a very suitable language for retrieving the desired information from such a database. One nice thing about Prolog is that we can refer to objects without actually specifying all the components of these objects. We can merely indicate the *structure* of objects that we are interested in, and leave the particular components in the structures unspecified or only partially specified. Figure 4.2 shows some examples. So we can refer to all Armstrong families by:

```
family( person( _, armstrong, _, _), _, _)
```

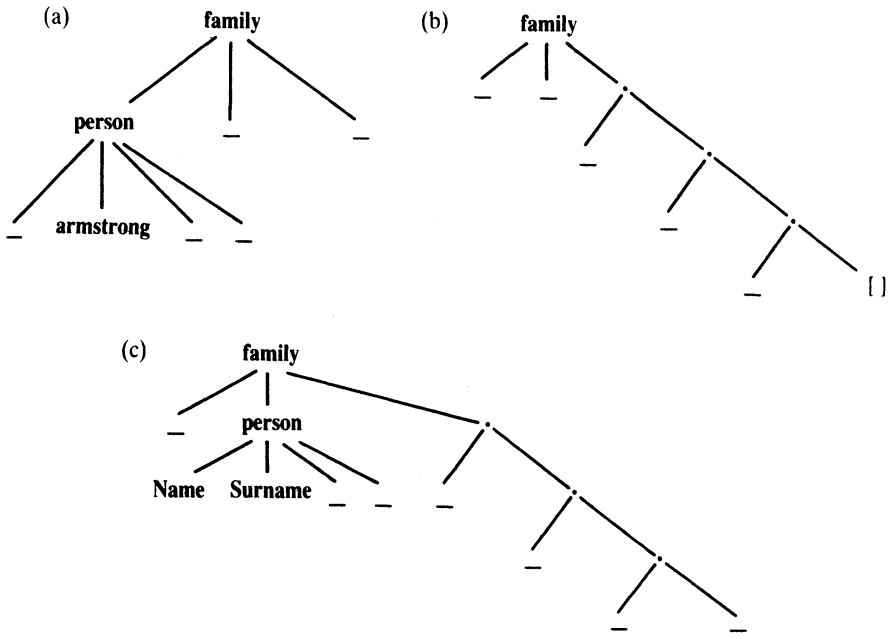


Figure 4.2 Specifying objects by their structural properties: (a) any Armstrong family; (b) any family with exactly three children; (c) any family with at least three children. Structure (c) makes provision for retrieving the wife's name through the instantiation of the variables **Name** and **Surname**.

The underscore characters denote different anonymous variables; we do not care about their values. Further, we can refer to all families with three children by the term:

```
family( _, _, [_, _, _] )
```

To find all married women that have at least three children we can pose the question:

```
?- family( _, person( Name, Surname, _, _), [_, _, _ | _] ).
```

The point of these examples is that we can specify objects of interest not by their content, but by their structure. We only indicate their structure and leave their arguments as unspecified slots.

We can provide a set of procedures that can serve as a utility to make the interaction with the database more comfortable. Such utility procedures could be part of the user interface. Some useful utility procedures for our database are:

```
husband( X) :-                          % X is a husband
  family( X, _, _).

wife( X) :-                             % X is a wife
  family( _, X, _ ).

child( X) :-                            % X is a child
  family( _, _, Children),
  member( X, Children).

member( X, [X | L] ).

member( X, [Y | L] ) :-
  member( X, L).

exists( Person) :-                      % Any person in the database
  husband( Person);
  wife( Person);
  child( Person).

dateofbirth( person( _, _, Date, _), Date).

salary( person( _, _, _, works( _, S) ), S).   % Salary of working person

salary( person( _, _, _, unemployed), 0).      % Salary of unemployed
```

We can use these utilities, for example, in the following queries to the database:

- Find the names of all the people in the database:

```
?- exists( person( Name, Surname, _, _) ).
```

- Find all children born in 1981:

```
?- child( X),
   dateofbirth( X, date( _, _, 1981) ).
```

- Find all employed wives:

```
?- wife( person( Name, Surname, _, works( _, _) ) ).
```

- Find the names of unemployed people who were born before 1963:

```
?- exists( person( Name, Surname, date( _, _, Year), unemployed) ),
   Year < 1963.
```

- Find people born before 1950 whose salary is less than 8000:

```
?- exists( Person),
   dateofbirth( Person, date( _, _, Year) ),
   Year < 1950,
   salary( Person, Salary),
   Salary < 8000.
```

- Find the names of families with at least three children:

```
?- family( person( _, Name, _, _), _, [_, _, _ | _] ).
```

To calculate the total income of a family it is useful to define the sum of salaries of a list of people as a two-argument relation:

```
total( List_of_people, Sum_of_their_salaries)
```

This relation can be programmed as:

```
total( [], 0).                          % Empty list of people

total( [Person | List], Sum) :-
  salary( Person, S),                   % S: salary of first person
  total( List, Rest),                   % Rest: sum of salaries of others
  Sum is S + Rest.
```

The total income of families can then be found by the question:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income).
```

Let the **length** relation count the number of elements of a list, as defined in

Section 3.4. Then we can specify all families that have an income per family member of less than 2000 by:

```
?- family( Husband, Wife, Children),
   total( [Husband, Wife | Children], Income),
   length( [Husband, Wife | Children], N),        % N: size of family
   Income/N < 2000.
```

Exercises

4.1 Write queries to find the following from the family database:

(a) names of families without children;

(b) all employed children;

(c) names of families with employed wives and unemployed husbands;

(d) all the children whose parents differ in age by at least 15 years.

4.2 Define the relation

twins(Child1, Child2)

to find twins in the family database.

4.2 Doing data abstraction

Data abstraction can be viewed as a process of organizing various pieces of information into natural units (possibly hierarchically), thus structuring the information into some conceptually meaningful form. Each such unit of information should be easily accessible in the program. Ideally, all the details of implementing such a structure should be invisible to the user of the structure – the programmer can then just concentrate on objects and relations between them. The point of the process is to make the use of information possible without the programmer having to think about the details of how the information is actually represented.

Let us discuss one way of carrying out this principle in Prolog. Consider our family example of the previous section again. Each family is a collection of pieces of information. These pieces are all clustered into natural units such as a person or a family, so they can be treated as single objects. Assume again that the family information is structured as in Figure 4.1. Let us now define some relations through which the user can access particular components of a family without knowing the details of Figure 4.1. Such relations can be called *selectors* as they select particular components. The name of such a selector relation will be the name of the component to be selected. The relation will have two

arguments: first, the object that contains the component, and second, the component itself:

```
selector_relation( Object, Component_selected)
```

Here are some selectors for the family structure:

```
husband( family( Husband, _, _), Husband).

wife( family( _, Wife, _), Wife).

children( family( _, _, ChildList), ChildList).
```

We can also define selectors for particular children:

```
firstchild( Family, First) :-
  children( Family, [First | _] ).

secondchild( Family, Second) :-
  children( Family, [_, Second | _] ).

...
```

We can generalize this to selecting the Nth child:

```
nthchild( N, Family, Child) :-
  children( Family, ChildList),
  nth_member( N, ChildList, Child).          % Nth element of a list
```

Another interesting object is a person. Some related selectors according to Figure 4.1 are:

```
firstname( person( Name, _, _, _), Name).

surname( person( _, Surname, _, _), Surname).

born( person( _, _, Date, _), Date).
```

How can we benefit from selector relations? Having defined them, we can now forget about the particular way that structured information is represented. To create and manipulate this information, we just have to know the names of the selector relations and use these in the rest of the program. In the case of complicated representations, this is easier than always referring to the representation explicitly. In our family example in particular, the user does not have to know that the children are represented as a list. For example, assume that we want to say that Tom Fox and Jim Fox belong to the same family and that Jim is the second child of Tom. Using the selector relations above, we can

define two persons, call them **Person1** and **Person2**, and the family. The following list of goals does this:

```
firstname( Person1, tom), surname( Person1, fox),  % Person1 is Tom Fox
firstname( Person2, jim), surname( Person2, fox),  % Person2 is Jim Fox
husband( Family, Person1),
secondchild( Family, Person2)
```

The use of selector relations also makes programs easier to modify. Imagine that we would like to improve the efficiency of a program by changing the representation of data. All we have to do is to change the definitions of the selector relations, and the rest of the program will work unchanged with the new representation.

Exercise

4.3 Complete the definition of **nthchild** by defining the relation

nth_member(N, List, X)

which is true if X is the Nth member of **List**.

4.3 Simulating a non-deterministic automaton

This exercise shows how an abstract mathematical construct can be translated into Prolog. In addition, our resulting program will be much more flexible than initially intended.

A *non-deterministic finite automaton* is an abstract machine that reads as input a string of symbols and decides whether to *accept* or to *reject* the input string. An automaton has a number of *states* and it is always in one of the states. It can change its state by moving from the current state to another state. The internal structure of the automaton can be represented by a transition graph such as that in Figure 4.3. In this example, s_1, s_2, s_3 and s_4 are the *states* of the automaton. Starting from the initial state (s_1 in our example), the automaton moves from state to state while reading the input string. Transitions depend on the current input symbol, as indicated by the arc labels in the transition graph.

A transition occurs each time an input symbol is read. Note that transitions can be non-deterministic. In Figure 4.3, if the automaton is in state s_1 and the current input symbol is *a* then it can transit into s_1 or s_2. Some arcs are labelled *null* denoting the 'null symbol'. These arcs correspond to 'silent moves' of the automaton. Such a move is said to be *silent* because it occurs without any reading of input, and the observer, viewing the automaton as a black box, will not be able to notice that any transition has occurred.

The state s_3 is double circled, which indicates that it is a *final state*. The

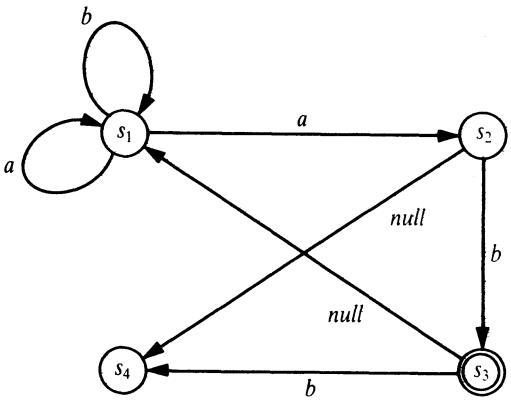


Figure 4.3 An example of a non-deterministic finite automaton.

automaton is said to *accept* the input string if there is a transition path in the graph such that

(1) it starts with the initial state,
(2) it ends with a final state, and
(3) the arc labels along the path correspond to the complete input string.

It is entirely up to the automaton to decide which of the possible moves to execute at any time. In particular, the automaton may choose to make or not to make a silent move, if it is available in the current state. But abstract non-deterministic machines of this kind have a magic property: if there is a choice then they always choose a 'right' move; that is, a move that leads to the acceptance of the input string, if such a move exists. The automaton in Figure 4.3 will, for example, accept the strings *ab* and *aabaab*, but it will reject the strings *abb* and *abba*. It is easy to see that this automaton accepts any string that terminates with *ab*, and rejects all others.

In Prolog, an automaton can be specified by three relations:

(1) A unary relation **final** which defines the final states of the automaton;

(2) A three-argument relation **trans** which defines the state transitions so that

```
trans( S1, X, S2)
```

means that a transition from a state S1 to S2 is possible when the current input symbol X is read.

(3) A binary relation

```
silent( S1, S2)
```

meaning that a silent move is possible from S1 to S2.

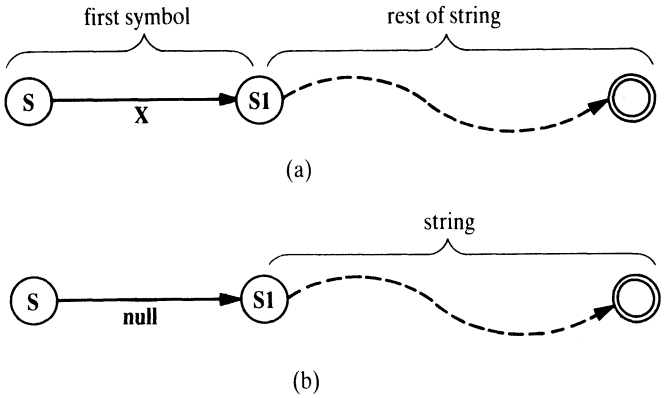


Figure 4.4 Accepting a string: (a) by reading its first symbol **X**; (b) by making a silent move.

For the automaton in Figure 4.3 these three relations are:

```
final( s3).

trans( s1, a, s1).
trans( s1, a, s2).
trans( s1, b, s1).
trans( s2, b, s3).
trans( s3, b, s4).

silent( s2, s4).
silent( s3, s1).
```

We will represent input strings as Prolog lists. So the string *aab* will be represented by **[a,a,b]**. Given the description of the automaton, the simulator will process a given input string and decide whether the string is accepted or rejected. By definition, the non-deterministic automaton accepts a given string if (starting from an initial state), after having read the whole input string, the automaton can (possibly) be in its final state. The simulator is programmed as a binary relation, **accepts**, which defines the acceptance of a string from a given state. So

```
accepts( State, String)
```

is true if the automaton, starting from the state **State** as initial state, accepts the string **String**. The **accepts** relation can be defined by three clauses. They correspond to the following three cases:

(1) The empty string, [], is accepted from a state S if S is a final state.

(2) A non-empty string is accepted from a state S if reading the first symbol in the string can bring the automaton into some state S1, and the rest of the string is accepted from S1. Figure 4.4(a) illustrates.

(3) A string is accepted from a state S if the automaton can make a silent move from S to S1 and then accept the (whole) input string from S1. Figure 4.4(b) illustrates.

These rules can be translated into Prolog as:

```
accepts( S, [] ) :-                          % Accept empty string
  final( S).

accepts( S, [X | Rest] ) :-                  % Accept by reading first symbol
  trans( S, X, S1),
  accepts( S1, Rest).

accepts( S, String) :-                       % Accept by making silent move
  silent( S, S1),
  accepts( S1, String).
```

The program can be asked, for example, about the acceptance of the string *aaab* by:

```
?- accepts( s1, [a,a,a,b] ).

yes
```

As we have already seen, Prolog programs are often able to solve more general problems than problems for which they were originally developed. In our case, we can also ask the simulator which state our automaton can be in initially so that it will accept the string *ab*:

```
?- accepts( S, [a,b] ).

S = s1;

S = s3
```

Amusingly, we can also ask: What are all the strings of length 3 that are accepted from state s_1?

```
?- accepts( s1, [X1,X2,X3] ).

X1 = a
X2 = a
X3 = b;

X1 = b
X2 = a
X3 = b;

no
```

If we prefer the acceptable input strings to be typed out as lists then we can formulate the question as:

```
?- String = [_, _, _], accepts( s1, String).

String = [a,a,b];

String = [b,a,b];

no
```

We can make further experiments asking even more general questions, such as: From what states will the automaton accept input strings of length 7?

Further experimentation could involve modifications in the structure of the automaton by changing the relations **final**, **trans** and **silent**. The automaton in Figure 4.3 does not contain any cyclic 'silent path' (a path that consists only of silent moves). If in Figure 4.3 a new transition

```
silent( s1, s3)
```

is added then a 'silent cycle' is created. But our simulator may now get into trouble. For example, the question

```
?- accepts( s1, [a] ).
```

would induce the simulator to cycle in state s_1 indefinitely, all the time hoping to find some way to the final state.

Exercises

4.4 Why could cycling not occur in the simulation of the original automaton in Figure 4.3, when there was no 'silent cycle' in the transition graph?

4.5 Cycling in the execution of **accepts** can be prevented, for example, by counting the number of moves made so far. The simulator would then be requested to search only for paths of some limited length. Modify the **accepts** relation this way. Hint: Add a third argument: the maximum number of moves allowed:

```
accepts( State, String, Max_moves)
```

4.4 Travel planning

In this section we will construct a program that gives advice on planning air travel. The program will be a rather simple advisor, yet it will be able to answer

some useful questions, such as:

- What days of the week is there a direct flight from London to Ljubljana?
- How can I get from Ljubljana to Edinburgh on Thursday?
- I have to visit Milan, Ljubljana and Zurich, starting from London on Tuesday and returning to London on Friday. In what sequence should I visit these cities so that I have no more than one flight each day of the tour?

The program will be centred around a database holding the flight information. This will be represented as a three-argument relation

timetable(Place1, Place2, List_of_flights)

where **List_of_flights** is a list of structured items of the form:

Departure_time / Arrival_time / Flight_number / List_of_days

List_of_days is either a list of weekdays or the atom 'alldays'. One clause of the **timetable** relation can be, for example:

```
timetable( london, edinburgh,
           [  9:40 / 10:50 / ba4733 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).
```

The times are represented as structured objects with two components, hours and minutes, combined by the operator ':'.

The main problem is to find exact routes between two given cities on a given day of the week. This will be programmed as a four-argument relation:

route(Place1, Place2, Day, Route)

Here **Route** is a sequence of flights that satisfies the following criteria:

(1) the start point of the route is **Place1**;
(2) the end point is **Place2**;
(3) all the flights are on the same day of the week, **Day**;
(4) all the flights in **Route** are in the **timetable** relation;
(5) there is enough time for transfer between flights.

The route is represented as a list of structured objects of the form:

From - To : Flight_number : Departure_time

We will also use the following auxiliary predicates:

(1) **flight(Place1, Place2, Day, Flight_num, Dep_time, Arr_time)**

This says that there is a flight, **Flight_num**, between **Place1** and **Place2** on the day of the week **Day** with the specified departure and arrival times.

(2) **deptime(Route, Time)**

Departure time of **Route** is **Time**.

(3) **transfer(Time1, Time2)**

There is at least 40 minutes between **Time1** and **Time2**, which should be sufficient for transfer between two flights.

The problem of finding a route is reminiscent of the simulation of the non-deterministic automaton of the previous section. The similarities of both problems are as follows:

- The states of the automaton correspond to the cities.
- A transition between two states corresponds to a flight between two cities.
- The **transition** relation of the automaton corresponds to the **timetable** relation.
- The automaton simulator finds a path in the transition graph between the initial state and a final state; the travel planner finds a route between the start city and the end city of the tour.

Not surprisingly, therefore, the **route** relation can be defined similarly to the **accepts** relation, with the exception that here we have no 'silent moves'. We have two cases:

(1) Direct flight connection: if there is a direct flight between places **Place1** and **Place2** then the route consists of this flight only:

```
route( Place1, Place2, Day, [Place1-Place2 : Fnum : Dep] ) :-
  flight( Place1, Place2, Day, Fnum, Dep, Arr).
```

(2) Indirect flight connection: the route between places P1 and P2 consists of the first flight, from P1 to some intermediate place P3, followed by a route between P3 to P2. In addition, there must be enough time between the arrival of the first flight and the departure of the second flight for transfer.

```
route( P1, P2, Day, [P1-P3 : Fnum1 : Dep1 | Route] ) :-
  route( P3, P2, Day, Route),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( Route, Dep2),
  transfer( Arr1, Dep2).
```

The auxiliary relations **flight**, **transfer** and **deptime** are easily programmed and are included in the complete travel planning program in Figure 4.5. Also included is an example timetable database.

Our route planner is extremely simple and may examine paths that obviously lead nowhere. Yet it will suffice if the flight database is not large. A really large database would require more intelligent planning to cope with the large number of potential candidate paths.

Some example questions to the program are as follows:

- What days of the week is there a direct flight from London to Ljubljana?

```
?- flight( london, ljubljana, Day, _, _, _).

Day = fr;

Day = su;

no
```

```
% A FLIGHT ROUTE PLANNER

:- op( 50, xfy, :).

flight( Place1, Place2, Day, Fnum, Deptime, Arrtime) :-
  timetable( Place1, Place2, Flightlist),
  member( Deptime / Arrtime / Fnum / Daylist , Flightlist),
  flyday( Day, Daylist).

member( X, [X | L] ).

member( X, [Y | L] ) :-
  member( X, L).

flyday( Day, Daylist) :-
  member( Day, Daylist).

flyday( Day, alldays) :-
  member( Day, [mo,tu,we,th,fr,sa,su] ).

route( P1, P2, Day [P1-P2 : Fnum : Deptime] ) :-          % Direct flight
  flight( P1, P2, Day, Fnum, Deptime, _).

route( P1, P2, Day, [P1-P3 : Fnum1 : Dep1 | Route] ) :- % Indirect connection
  route( P3, P2, Day, Route),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( Route, Dep2),
  transfer( Arr1, Dep2).

deptime( [P1-P2 : Fnum : Dep | _], Dep).

transfer( Hours1:Mins1, Hours2:Mins2) :-
  60 * (Hours2 - Hours1) + Mins2 - Mins1 >= 40.
```

```
% A FLIGHT DATABASE

timetable( edinburgh, london,
           [  9:40 / 10:50 / ba4733 / alldays,
             13:40 / 14:50 / ba4773 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).

timetable( london, edinburgh,
           [  9:40 / 10:50 / ba4732 / alldays,
             11:40 / 12:50 / ba4752 / alldays,
             18:40 / 19:50 / ba4822 / [mo,tu,we,th,fr] ] ).

timetable( london, ljubljana,
           [ 13:20 / 16:20 / ju201 / [fr],
             13:20 / 16:20 / ju213 / [su] ] ).

timetable( london, zurich,
           [  9:10 / 11:45 / ba614 / alldays,
             14:45 / 17:20 / sr805 / alldays ] ).

timetable( london, milan,
           [  8:30 / 11:20 / ba510 / alldays,
             11:00 / 13:50 / az459 / alldays ] ).

timetable( ljubljana, zurich,
           [ 11:30 / 12:40 / ju322 / [tu,th] ] ).

timetable( ljubljana, london,
           [ 11:10 / 12:20 / yu200 / [fr],
             11:25 / 12:20 / yu212 / [su] ] ).

timetable( milan, london,
           [  9:10 / 10:00 / az458 / alldays,
             12:20 / 13:10 / ba511 / alldays ] ).

timetable( milan, zurich,
           [  9:25 / 10:15 / sr621 / alldays,
             12:45 / 13:35 / sr623 / alldays ] ).

timetable( zurich, ljubljana,
           [ 13:30 / 14:40 / yu323 / [tu,th] ] ).

timetable( zurich, london,
           [  9:00 / 9:40 / ba613 / [mo,tu,we,th,fr,sa],
             16:10 / 16:55 / sr806 / [mo,tu,we,th,fr,su] ] ).

timetable( zurich, milan,
           [  7:55 / 8:45 / sr620 / alldays ] ).
```

Figure 4.5 A flight route planner and an example flight timetable.

- How can I get from Ljubljana to Edinburgh on Thursday?

```
?- route( ljubljana, edinburgh, th, R).

R = [ljubljana-zurich:yu322:11:30, zurich-london:sr806:16:10,
     london-edinburgh:ba4822:18:40]
```

- How can I visit Milan, Ljubljana and Zurich, starting from London on Tuesday and returning to London on Friday, with no more than one flight each day of the tour? This question is somewhat trickier. It can be formulated by using the **permutation** relation, programmed in Chapter 3. We are asking for a permutation of the cities Milan, Ljubljana and Zurich such that the corresponding flights are possible on successive days:

```
?- permutation( [milan, ljubljana, zurich], [City1, City2, City3] ),
   flight( london, City1, tu, FN1, Dep1, Arr1),
   flight( City1, City2, we, FN2, Dep2, Arr2),
   flight( City2, City3, th, FN3, Dep3, Arr3),
   flight( City3, london, fr, FN4, Dep4, Arr4).

City1 = milan
City2 = zurich
City3 = ljubljana
FN1 = ba510
Dep1 = 8:30
Arr1 = 11:20
FN2 = sr621
Dep2 = 9:25
Arr2 = 10:15
FN3 = yu323
Dep3 = 13:30
Arr3 = 14:40
FN4 = yu200
Dep4 = 11:10
Arr4 = 12:20
```

4.5 The eight queens problem

The problem here is to place eight queens on the empty chessboard in such a way that no queen attacks any other queen. The solution will be programmed as a unary predicate

```
solution( Pos)
```

which is true if and only if **Pos** represents a position with eight queens that do

not attack each other. It will be interesting to compare various ideas for programming this problem. Therefore we will present three programs based on somewhat different representations of the problem.

4.5.1 Program 1

First we have to choose a representation of the board position. One natural choice is to represent the position by a list of eight items, each of them corresponding to one queen. Each item in the list will specify a square of the board on which the corresponding queen is sitting. Further, each square can be specified by a pair of coordinates (X and Y) on the board, where each coordinate is an integer between 1 and 8. In the program we can write such a pair as

X/Y

where, of course, the '/' operator is not meant to indicate division, but simply combines both coordinates together into a square. Figure 4.6 shows one solution of the eight queens problem and its list representation.

Having chosen this representation, the problem is to find such a list of the form

[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]

which satisfies the no-attack requirement. Our procedure **solution** will have to search for a proper instantiation of the variables **X1, Y1, X2, Y2, ..., X8, Y8**. As we know that all the queens will have to be in different columns to prevent vertical attacks, we can immediately constrain the choice and so make the search task easier. We can thus fix the X-coordinates so that the solution list will fit the following, more specific template:

[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]

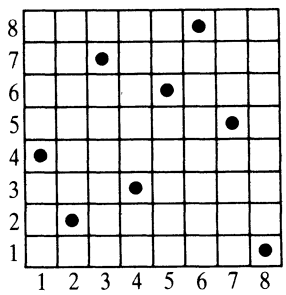


Figure 4.6 A solution to the eight queens problem. This position can be specified by the list **[1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/8]**.

We are interested in the solution on a board of size 8 by 8. However, in programming, in general, the key to the solution is often in considering a more general problem. Paradoxically, it is often the case that the solution for the more general problem is easier to formulate than that for the more specific, original problem; then the original problem is simply solved as a special case of the more general problem.

The creative part of the problem is to find the correct generalization of the original problem. In our case, a good idea is to generalize the number of queens (the number of columns in the list) from 8 to any number, including zero. The **solution** relation can then be formulated by considering two cases:

Case 1 The list of queens is empty: the empty list is certainly a solution because there is no attack.

Case 2 The list of queens is non-empty: then it looks like this:

[X/Y | Others]

In case 2, the first queen is at some square **X/Y** and the other queens are at squares specified by the list **Others**. If this is to be a solution then the following conditions must hold:

(1) There must be no attack between the queens in the list **Others**; that is, **Others** itself must also be a solution.

(2) X and Y must be integers between 1 and 8.

(3) A queen at square **X/Y** must not attack any of the queens in the list **Others**.

To program the first condition we can simply use the **solution** relation itself. The second condition can be specified as follows: Y will have to be a member of the list of integers between 1 and 8 – that is, **[1,2,3,4,5,6,7,8]**. On the other hand, we do not have to worry about X since the solution list will have to match the template in which the X-coordinates are already specified. So X will be guaranteed to have a proper value between 1 and 8. We can implement the third condition as another relation, **noattack**. All this can then be written in Prolog as follows:

```
solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).
```

It now remains to define the **noattack** relation:

```
noattack( Q, Qlist)
```

Again, this can be broken down into two cases:

(1) If the list **Qlist** is empty then the relation is certainly true because there is no queen to be attacked.

(2) If **Qlist** is not empty then it has the form **[Q1 | Qlist1]** and two conditions must be satisfied:

   (a) the queen at Q must not attack the queen at Q1, and

   (b) the queen at Q must not attack any of the queens in **Qlist1**.

To specify that a queen at some square does not attack another square is easy: the two squares must not be in the same row, the same column or the same diagonal. Our solution template guarantees that all the queens are in different columns, so it only remains to specify explicitly that:

- the Y-coordinates of the queens are different, and
- they are not in the same diagonal, either upward or downward; that is, the distance between the squares in the X-direction must not be equal to that in the Y-direction.

Figure 4.7 shows the complete program. To alleviate its use a template list has

```
solution( [] ).

solution( [X/Y | Others] ) :-          % First queen at X/Y, other queens at Others
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).              % First queen does not attack others

noattack( _, [] ).                     % Nothing to attack

noattack( X/Y, [X1/Y1 | Others] ) :-
  Y =\= Y1,                            % Different Y-coordinates
  Y1-Y =\= X1-X,                       % Different diagonals
  Y1-Y =\= X-X1,
  noattack( X/Y, Others).

member( X, [X | L] ).

member( X, [Y | L] ) :-
  member( X, L).

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Figure 4.7 Program 1 for the eight queens problem.

been added. This list can be retrieved in a question for generating solutions. So we can now ask

```
?- template( S), solution( S).
```

and the program will generate solutions as follows:

```
S = [ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];
S = [ 1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];
S = [ 1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];
...
```

Exercise

4.6 When searching for a solution, the program of Figure 4.7 explores alternative values for the Y-coordinates of the queens. At which place in the program is the order of alternatives defined? How can we easily modify the program to change the order? Experiment with different orders with the view of studying the executional efficiency of the program.

4.5.2 Program 2

In the board representation of program 1, each solution had the form

```
[ 1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]
```

because the queens were simply placed in consecutive columns. No information is lost if the X-coordinates were omitted. So a more economical representation of the board position can be used, retaining only the Y-coordinates of the queens:

```
[ Y1, Y2, Y3, ..., Y8]
```

To prevent the horizontal attacks, no two queens can be in the same row. This imposes a constraint on the Y-coordinates. The queens have to occupy all the rows 1, 2, ..., 8. The choice that remains is the *order* of these eight numbers. Each solution is therefore represented by a permutation of the list

```
[1,2,3,4,5,6,7,8]
```

Such a permutation, S, is a solution if all the queens are safe. So we can write:

```
solution( S) :-
  permutation( [1,2,3,4,5,6,7,8], S),
  safe( S).
```

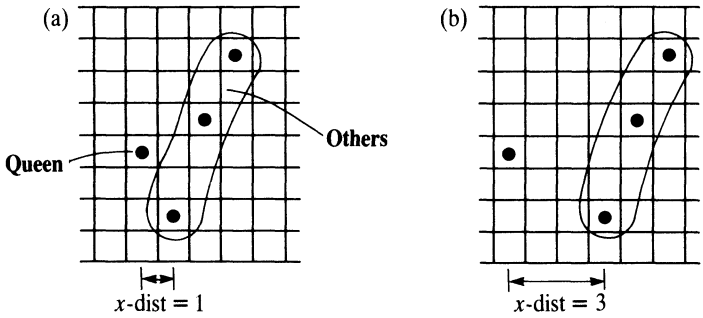


Figure 4.8 (a) X-distance between **Queen** and **Others** is 1. (b) X-distance between **Queen** and **Others** is 3.

We have already programmed the **permutation** relation in Chapter 3, but the **safe** relation remains to be specified. We can split its definition into two cases:

(1) S is the empty list: this is certainly safe as there is nothing to be attacked.

(2) S is a non-empty list of the form **[Queen | Others]**. This is safe if the list **Others** is safe, and **Queen** does not attack any queen in the list **Others**.

In Prolog, this is:

```
safe( [] ).

safe( [Queen | Others] ) :-
  safe( Others),
  noattack( Queen, Others).
```

The **noattack** relation here is slightly trickier. The difficulty is that the queens' positions are only defined by their Y-coordinates, and the X-coordinates are not explicitly present. This problem can be circumvented by a small generalization of the **noattack** relation, as illustrated in Figure 4.8. The goal

```
noattack( Queen, Others)
```

is meant to ensure that **Queen** does not attack **Others** when the X-distance between **Queen** and **Others** is equal to 1. What is needed is the generalization of the X-distance between **Queen** and **Others**. So we add this distance as the third argument of the **noattack** relation:

```
noattack( Queen, Others, Xdist)
```

Accordingly, the **noattack** goal in the **safe** relation has to be modified to

```
noattack( Queen, Others, 1)
```

```
solution( Queens) :-
  permutation( [1,2,3,4,5,6,7,8], Queens),
  safe( Queens).

permutation( [], [] ).

permutation( [Head | Tail], PermList) :-
  permutation( Tail, PermTail),
  del( Head, PermList, PermTail).          % Insert Head in permuted Tail

del( A, [A | List], List).

del( A, [B | List], [B | List1] ) :-
  del( A, List, List1).

safe( [] ).

safe( [Queen | Others] ) :-
  safe( Others),
  noattack( Queen, Others, 1).

noattack( _, [], _).

noattack( Y, [Y1 | Ylist], Xdist) :-
  Y1-Y =\= Xdist,
  Y-Y1 =\= Xdist,
  Dist1 is Xdist + 1,
  noattack( Y, Ylist, Dist1).
```

Figure 4.9 Program 2 for the eight queens problem.

The **noattack** relation can now be formulated according to two cases, depending on the list **Others**: if **Others** is empty then there is no target and certainly no attack; if **Others** is non-empty then **Queen** must not attack the first queen in **Others** (which is **Xdist** columns from **Queen**) and also the tail of **Others** at **Xdist + 1**. This leads to the program shown in Figure 4.9.

4.5.3 Program 3

Our third program for the eight queens problem will be based on the following reasoning. Each queen has to be placed on some square; that is, into some column, some row, some upward diagonal and some downward diagonal. To make sure that all the queens are safe, each queen must be placed in a different column, a different row, a different upward and a different downward diago-

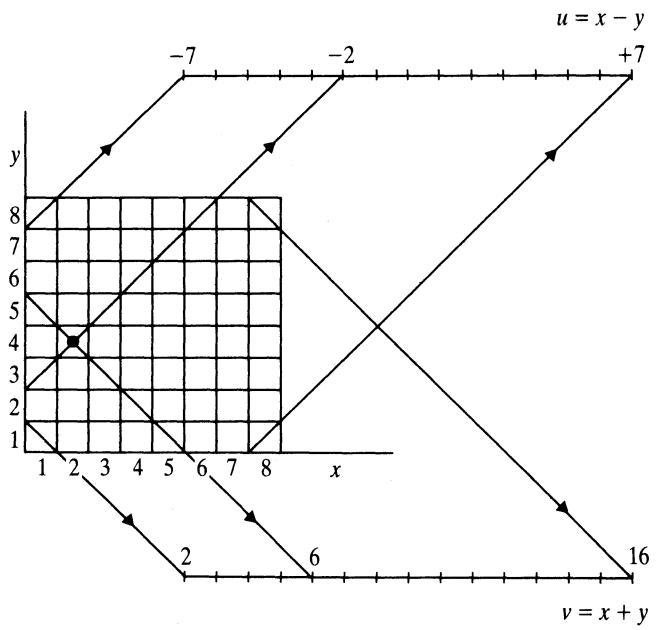


Figure 4.10 The relation between columns, rows, upward and downward diagonals. The indicated square has coordinates: $x = 2$, $y = 4$, $u = 2 - 4 = -2$, $v = 2 + 4 = 6$.

nal. It is thus natural to consider a richer representation system with four coordinates:

x	columns
y	rows
u	upward diagonals
v	downward diagonals

The coordinates are not independent: given x and y, u and v are determined (Figure 4.10 illustrates). For example, as

$$u = x - y$$

$$v = x + y$$

The domains for all four dimensions are:

Dx = [1,2,3,4,5,6,7,8]

Dy = [1,2,3,4,5,6,7,8]

Du = [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7]

Dv = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

The eight queens problem can now be stated as follows: select eight 4-tuples (X,Y,U,V) from the domains (X from Dx, Y from Dy, etc.), never using the same element twice from any of the domains. Of course, once X and Y are chosen, U and V are determined. The solution can then be, roughly speaking, as follows: given all four domains, select the position of the first queen, delete the corresponding items from the four domains, and then use the rest of the domains for placing the rest of the queens. A program based on this idea is shown in Figure 4.11. The board position is, again, represented by a list of Y-coordinates. The key relation in this program is

sol(Ylist, Dx, Dy, Du, Dv)

which instantiates the Y-coordinates (in **Ylist**) of the queens, assuming that they are placed in consecutive columns taken from Dx. All Y-coordinates and the corresponding U and V-coordinates are taken from the lists Dy, Du and Dv. The top procedure, **solution**, can be invoked by the question

?- solution(S).

This will cause the invocation of **sol** with the complete domains that correspond to the problem space of eight queens.

```
solution( Ylist) :-
  sol( Ylist,                                  % Y-coordinates of queens
       [1,2,3,4,5,6,7,8],                      % Domain for Y-coordinates
       [1,2,3,4,5,6,7,8],                      % Domain for X-coordinates
       [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7], % Upward diagonals
       [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ).  % Downward diagonals

sol( [], [], Dy, Du, Dv).

sol( [Y | Ylist], [X | Dx1], Dy, Du, Dv) :-
  del( Y, Dy, Dy1),                  % Choose a Y-coordinate
  U is X-Y,                          % Corresponding upward diagonal
  del( U, Du, Du1),                  % Remove it
  V is X+Y,                          % Corresponding downward diagonal
  del( V, Dv, Dv1),                  % Remove it
  sol( Ylist, Dx1, Dy1, Du1, Dv1).   % Use remaining values

del( A, [A | List], List).

del( A, [B | List], [B | List1] ) :-
  del( A, List, List1).
```

Figure 4.11 Program 3 for the eight queens problem.

The **sol** procedure is general in the sense that it can be used for solving the N-queens problem (on a chessboard of size N by N). It is only necessary to properly set up the domains Dx, Dy, etc.

It is practical to mechanize the generation of the domains. For that we need a procedure

```
gen( N1, N2, List)
```

which will, for two given integers N1 and N2, produce the list

```
List = [ N1, N1 + 1, N1 + 2, ..., N2 - 1, N2]
```

Such a procedure is:

```
gen( N, N, [N] ).

gen( N1, N2, [N1 | List] ) :-
  N1 < N2,
  M is N1 + 1,
  gen( M, N2, List).
```

The top level relation, **solution**, has to be accordingly generalized to

```
solution( N, S)
```

where N is the size of the board and S is a solution represented as a list of Y-coordinates of N queens. The generalized **solution** relation is:

```
solution( N, S) :-
  gen( 1, N, Dxy),
  Nu1 is 1 - N, Nu2 is N - 1,
  gen( Nu1, Nu2, Du),
  Nv2 is N + N,
  gen( 2, Nv2, Dv),
  sol( S, Dxy, Dxy, Du, Dv).
```

For example, a solution to the 12-queens problem would be generated by:

```
?- solution( 12, S).

S = [1,3,5,8,10,12,6,11,2,7,9,4]
```

4.5.4 Concluding remarks

The three solutions to the eight queens problem show how the same problem can be approached in different ways. We also varied the representation of data. Sometimes the representation was more economical, sometimes it was more

explicit and partially redundant. The drawback of the more economical representation is that some information always has to be recomputed when it is required.

At several points, the key step toward the solution was to generalize the problem. Paradoxically, by considering a more general problem, the solution became easier to formulate. This generalization principle is a kind of standard technique that can often be applied.

Of the three programs, the third one illustrates best how to approach general problems of constructing under constraints a structure from a given set of elements.

A natural question is: Which of the three programs is most efficient? In this respect, program 2 is far inferior while the other two programs are similar. The reason is that permutation-based program 2 constructs complete permutations while the other two programs are able to recognize and reject unsafe permutations when they are only partially constructed. Program 3 is the most efficient. It avoids some of the arithmetic computation that is essentially captured in the redundant board representation this program uses.

Exercise

4.7 Let the squares of the chessboard be represented by pairs of their coordinates of the form **X/Y**, where both X and Y are between 1 and 8.

(a) Define the relation **jump(Square1, Square2)** according to the knight jump on the chessboard. Assume that **Square1** is always instantiated to a square while **Square2** can be uninstantiated. For example:

```
?- jump( 1/1, S).

S = 3/2;

S = 2/3;

no
```

(b) Define the relation **knightpath(Path)** where **Path** is a list of squares that represent a legal path of a knight on the empty chessboard.

(c) Using this **knightpath** relation, write a question to find any knight's path of length 4 moves from square 2/1 to the opposite edge of the board (Y = 8) that goes through square 5/4 after the second move.

Summary

The examples of this chapter illustrate some strong points and characteristic features of Prolog programming:

- A database can be naturally represented as a set of Prolog facts.

- Prolog's mechanisms of querying and matching can be flexibly used for retrieving structured information from a database. In addition, utility procedures can be easily defined to further alleviate the interaction with a particular database.
- *Data abstraction* can be viewed as a programming technique that makes the use of complex data structures easier, and contributes to the clarity of programs. It is easy in Prolog to carry out the essential principles of data abstraction.
- Abstract mathematical constructs, such as automata, can often be readily translated into executable Prolog definitions.
- As in the case of eight queens, the same problem can be approached in different ways by varying the representation of the problem. Often, introducing redundancy into the representation saves computation. This entails trading space for time.
- Often, the key step toward a solution is to generalize the problem. Paradoxically, by considering a more general problem the solution may become easier to formulate.