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

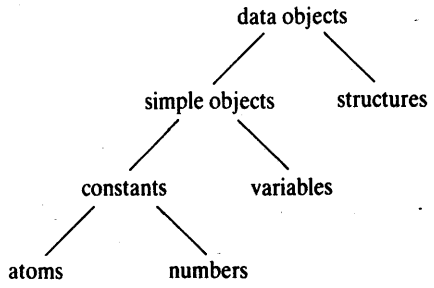

Figure 2.1 Data objects in Prolog.

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,

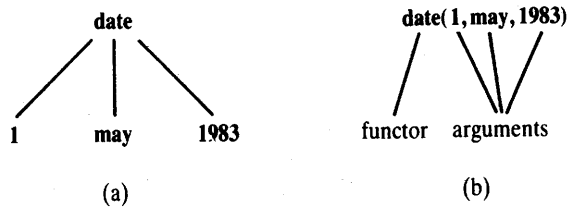

Figure 2.2 Date is an example of a structured object: (a) as it is represented as a tree; (b) as it is written in Prolog.

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

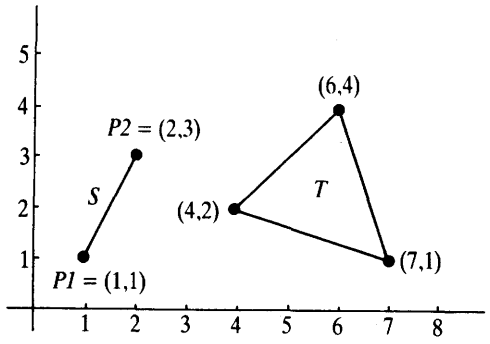

Figure 2.3 Some simple geometric objects.

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) )
```

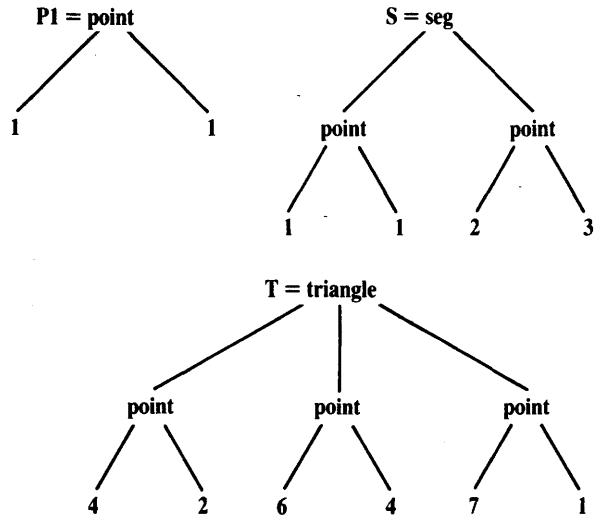

Figure 2.4 Tree representation of the objects in Figure 2.3.

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))**

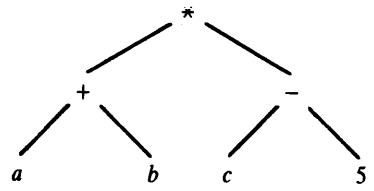

Figure 2.5 A tree structure that corresponds to the arithmetic expression $(a + b)*(c - 5)$.

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

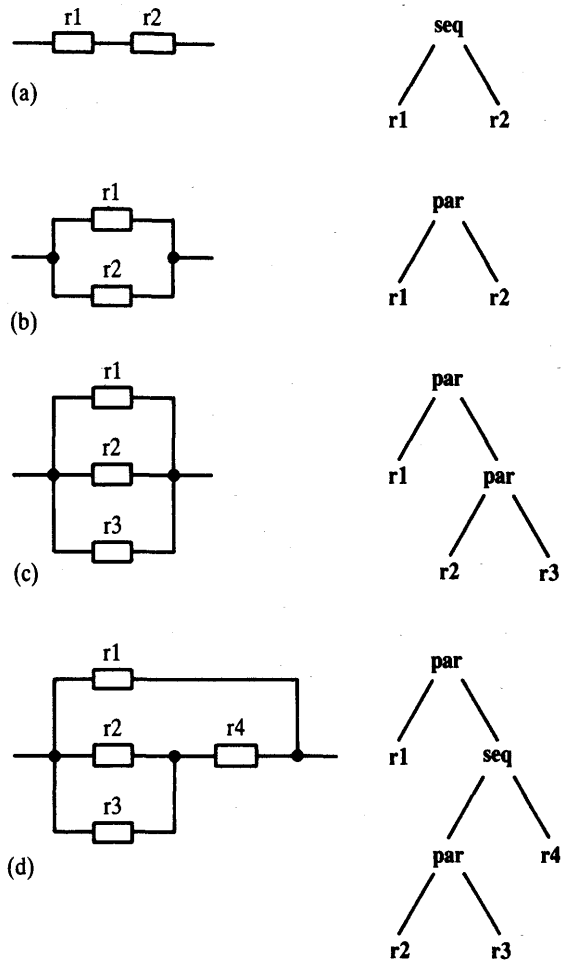

Figure 2.6 Some simple electric circuits and their tree representations: (a) sequential composition of resistors r1 and r2; (b) parallel composition of two resistors; (c) parallel composition of three resistors; (d) parallel composition of r1 and another circuit.

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

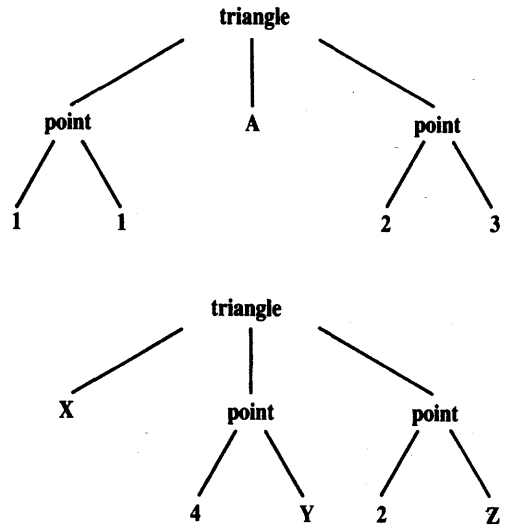

Figure 2.7 Matching **triangle(point(1,1), A, point(2,3)) = triangle(X, point(4,Y), point(2,Z)).**

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.

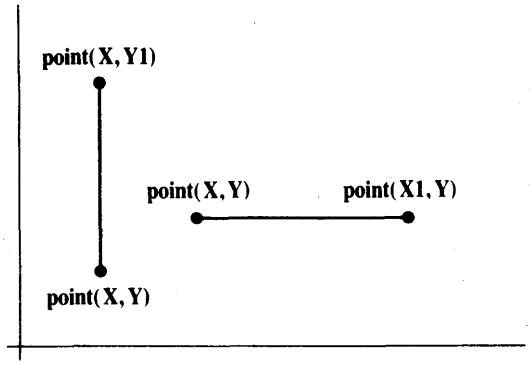

Figure 2.8 Illustration of vertical and horizontal line segments.

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(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.

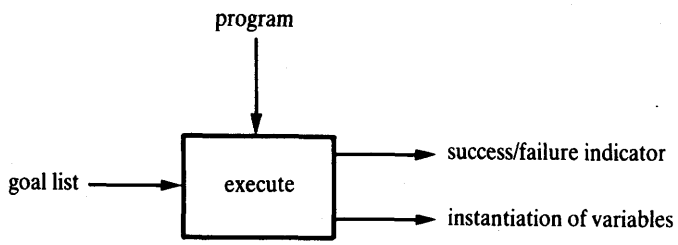

Figure 2.9 Input/output view of the procedure that executes a list of goals.

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?