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-

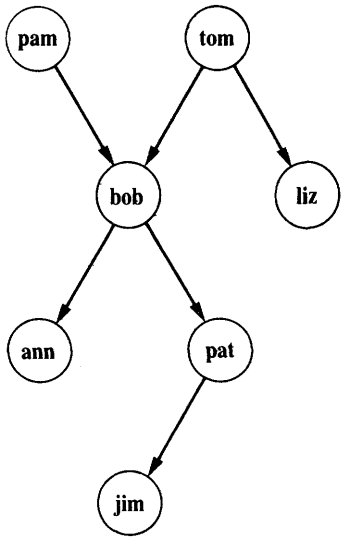

Figure 1.1 A family tree.

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).

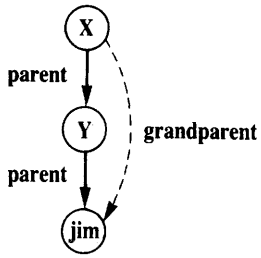

Figure 1.2 The **grandparent** relation expressed as a composition of two **parent** relations.

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

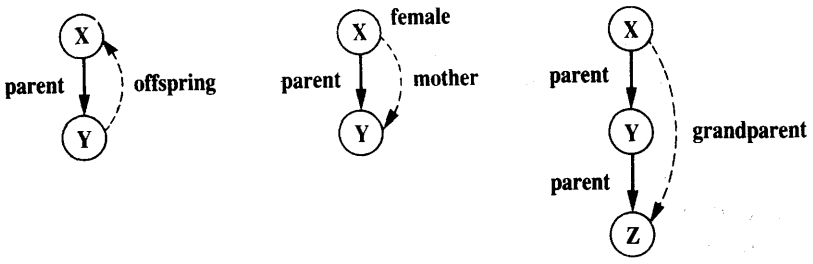

Figure 1.3 Definition graphs for the relations **offspring**, **mother** and **grandparent** in terms of other relations.

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.

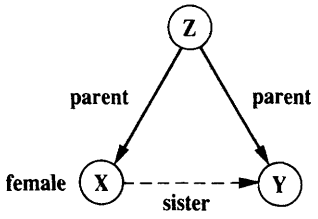

Figure 1.4 Defining the **sister** relation.

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.

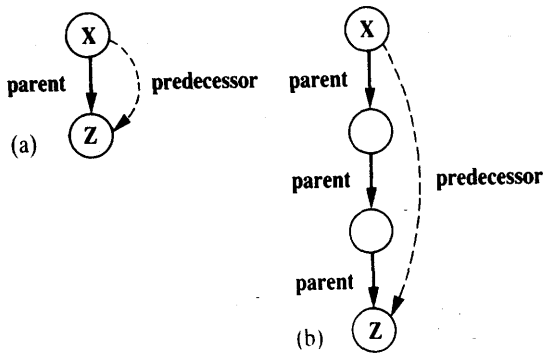

Figure 1.5 Examples of the **predecessor** relation: (a) **X** is a *direct* predecessor of **Z**; (b) **X** is an indirect predecessor of **Z**.

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**

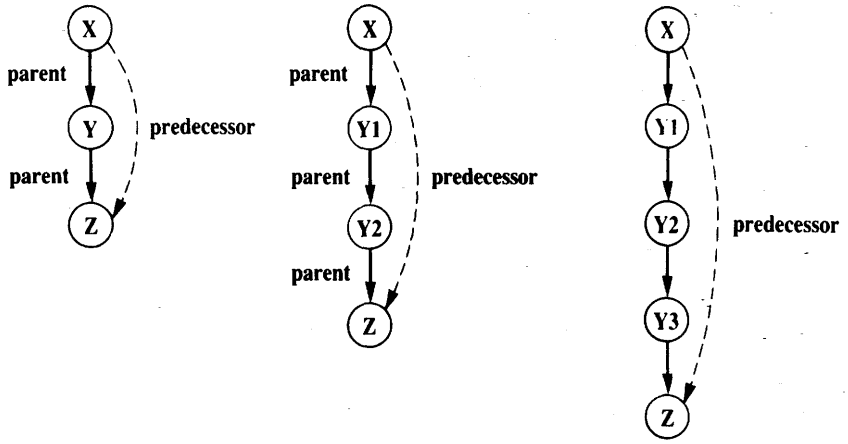

Figure 1.6 Predecessor-successor pairs at various distances.

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.

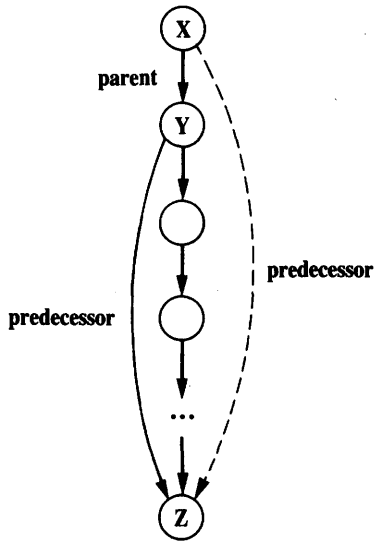

Figure 1.7 Recursive formulation of the **predecessor** relation.

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).
```

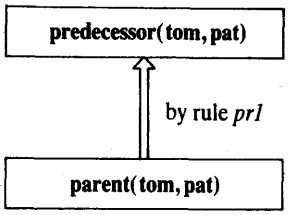

Figure 1.9 The first step of the execution. The top goal is true if the bottom goal is true.

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

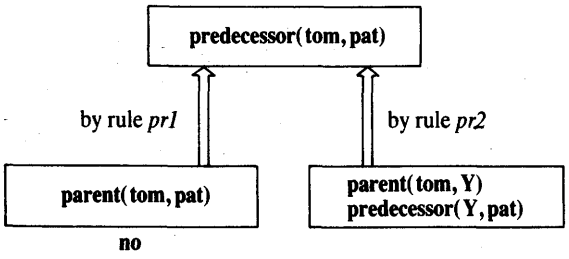

Figure 1.10 Execution trace continued from Figure 1.9.

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)

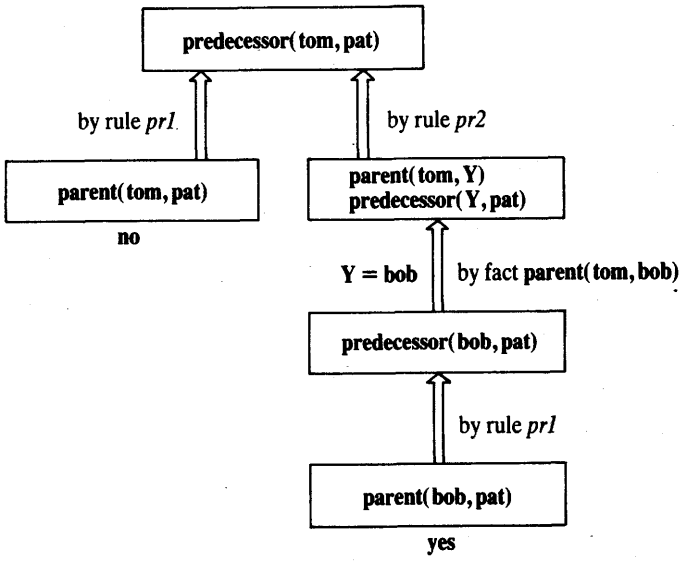

Figure 1.11 The complete execution trace to satisfy the goal **predecessor(tom, pat)**. The right-hand branch proves the goal is satisfiable.

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).