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 such as: Prolog clauses, facts, rules and procedures. Prolog's built-in backtracking mechanism and the distinction between declarative and procedural meanings of a program are discussed.

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 arguments. 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. For example, **parent(tom, bob)** is a particular *instance* of the **parent** relation. Such an instance is also called a *relationship*. In general, a relation is defined as the set of all its instances.

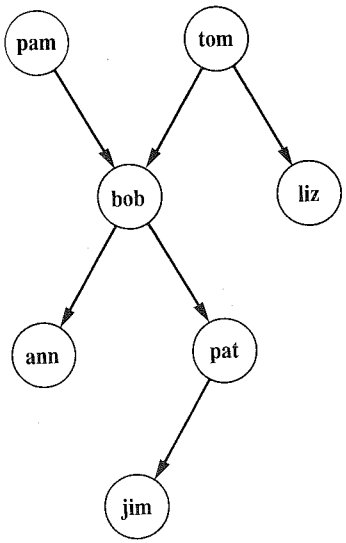

Figure 1.1 A family tree.

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 request another solution (in many Prolog implementations by typing a semicolon), and Prolog will find:

```
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).

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
```


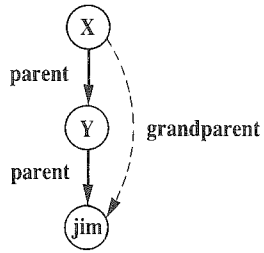

Figure 1.2 The **grandparent** relation expressed as a composition of two **parent** relations.

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 are 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 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

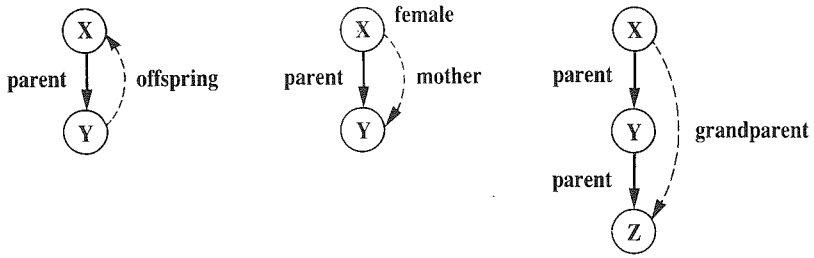

Figure 1.3 Definition graphs for the relations **offspring**, **mother** and **grandparent** in terms of other relations.

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.

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,

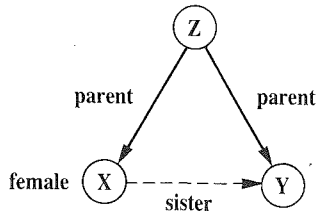

Figure 1.4 Defining the **sister** relation.

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 a head and 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.