2

Clauses and Predicates

Chapter Aims

After reading this chapter you should be able to:

- Identify the components of rules and facts
- Explain the meaning of the term predicate
- Make correct use of variables in goals and clauses.

2.1 Clauses

Apart from comments and blank lines, which are ignored, a Prolog program consists of a succession of *clauses*. A clause can run over more than one line or there may be several on the same line. A clause is terminated by a dot character, followed by at least one 'white space' character, e.g. a space or a carriage return.

There are two types of clause: *facts* and *rules*. Facts are of the form

head.

head is called the *head of the clause*. It takes the same form as a goal entered by the user at the prompt, i.e. it must be an atom or a compound term. Atoms and compound terms are known collectively as *call terms*. The significance of call terms will be explained in Chapter 3.

Some examples of facts are:

```
christmas.
likes(john,mary).
likes(X,prolog).
dog(fido).
```

Rules are of the form:

head:-t_1,t_2, … , t_k. (k>=1)

head is called the *head of the clause* (or the *head of the rule*) and, as for facts, must be a call term, i.e. an atom or a compound term.

:- is called the neck of the clause (or the *'neck operator*'). It is read as 'if'.

t_1,t_2, … , t_k is called the *body of the clause* (or the *body of the rule*). It specifies the conditions that must be met in order for the conclusion, represented by the head, to be satisfied. The body consists of one or more components, separated by commas. The components are *goals* and the commas are read as 'and'.

Each goal must be a call term, i.e. an atom or a compound term. A rule can be read as '*head* is true if t_1, t_2, …, t_k are all true'.

The head of a rule can also be viewed as a *goal* with the components of its body viewed as subgoals. Thus another reading of a rule is 'to achieve goal *head*, it is necessary to achieve subgoals t_1, t_2,…, t_k in turn'.

Some examples of rules are:

```
large_animal(X):-animal(X),large(X).
grandparent(X,Y):-father(X,Z),parent(Z,Y).
go:-write('hello world'),nl.
```

Here is another version of the animals program, which includes both facts and rules.

```
/* Animals Program 2  */
dog(fido). large(fido).
cat(mary). large(mary).
dog(rover). dog(jane).
dog(tom). large(tom). cat(harry).
```

```
dog(fred). dog(henry).
cat(bill). cat(steve).
small(henry). large(fred).
large(steve). large(jim).
large(mike).
large_animal(X):- dog(X),large(X).
large_animal(Z):- cat(Z),large(Z).
```

fido, *mary*, *jane* etc. are atoms, i.e. constants, indicated by their initial lower case letters. *X* and *Y* are variables, indicated by their initial capital letters.

The first 18 clauses are facts. The final two clauses are rules.

2.2 Predicates

The following simple program has five clauses. For each of the first three clauses, the head is a compound term with *functor* **parent** and *arity* 2 (i.e. two arguments).

```
parent(victoria,albert).
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,henry).
mother(jane,henry).
```

It is possible (although likely to cause confusion) for the program also to include clauses for which the head has functor **parent**, but a different arity, for example

```
parent(john).
parent(X):-son(X,Y).
/* X is a parent if X has a son Y */
```

It is also possible for **parent** to be used as an atom in the same program, for example in the fact

```
animal(parent).
```

but this too is likely to cause confusion.

All the clauses (facts and rules) for which the head has a given combination of functor and arity comprise a definition of a *predicate*. The clauses do not have to appear as consecutive lines of a program but it makes programs easier to read if they do.

The clauses given above define two predicates with the name **parent**, one with arity two and the other with arity one. These can be written (in textbooks, reference manuals etc., not in programs) as **parent/2** and **parent/1**, to distinguish between them. When there is no risk of ambiguity, it is customary to refer to a predicate as just **dog**, **large_animal** etc.

An atom appearing as a fact or as the head of a rule, e.g.

```
christmas.

go:-parent(john,B),
   write('john has a child named '),
   write(B),nl.
```

can be regarded as a predicate with no arguments, e.g. **go/0**.

There are five predicates defined in Animals Program 2: **dog/1**, **cat/1**, **large/1**, **small/1** and **large_animal/1**. The first 18 clauses are facts defining the predicates **dog/1**, **cat/1**, **large/1** and **small/1** (6, 4, 7 and 1 clauses, respectively). The final two clauses are rules, which together define the predicate **large_animal/1**.

Declarative and Procedural Interpretations of Rules

Rules have both a *declarative* and a *procedural* interpretation. For example, the declarative interpretation of the rule

```
chases(X,Y):-dog(X),cat(Y),write(X),
   write(' chases '),write(Y),nl.
```

is: '**chases(X,Y)** is true if **dog(X)** is true and **cat(Y)** is true and **write(X)** is true, etc.'

The procedural interpretation is 'To satisfy **chases(X,Y)**, first satisfy **dog(X)**, then satisfy **cat(Y)**, then satisfy **write(X)**, etc.'

Facts are generally interpreted *declaratively*, e.g.

```
dog(fido).
```

is read as 'fido is a dog'.

The order of the clauses defining a predicate and the order of the goals in the body of each rule are irrelevant to the declarative interpretation but of vital importance to the procedural interpretation and thus to determining whether or not the sequence of goals entered by the user at the system prompt is satisfied. When

evaluating a goal, the clauses in the database are examined from top to bottom. Where necessary, the goals in the body of a rule are examined from left to right. This topic will be discussed in detail in Chapter 3.

A user's program comprises facts and rules that define new predicates. These are called *user-defined predicates*. In addition there are standard predicates pre-defined by the Prolog system. These are known as *built-in predicates* (BIPs) and may not be redefined by a user program. Some examples are: **write/1**, **nl/0**, **repeat/0**, **member/2**, **append/3**, **consult/1**, **halt/0**. Some BIPs are common to all versions of Prolog. Others are version-dependent.

Two of the most commonly used built-in predicates are **write/1** and **nl/0**.

The **write/1** predicate takes a term as its argument, e.g.

write(hello)

write(X)

write('hello world')

Providing its argument is a valid term, the **write** predicate always succeeds and as a side effect writes the value of the term to the user's screen. To be more precise it is output to the *current output stream*, which by default will be assumed to be the user's screen. Information about output to other devices is given in Chapter 5. If the argument is a quoted atom, e.g. 'hello world', the quotes are not output.

The **nl/0** predicate is an atom, i.e. a predicate that takes no arguments. The predicate always succeeds and as a side effect starts a new line on the user's screen.

The name of a user-defined predicate (the functor) can be any atom, with a few exceptions, except that you may not redefine any of the Prolog system's built-in predicates. You are most unlikely to want to redefine the **write/1** predicate by putting a clause such as

```
write(27).
```

or

```
write(X):-dog(X).
```

in your programs, but if you do the system will give an error message such as 'illegal attempt to redefine a built-in predicate'.

The most important built-in predicates are described in Appendix 1. Each version of Prolog is likely to have others – sometimes many others – and if you accidentally use one of the same name and arity for one of your own predicates you will get the 'illegal attempt to redefine a built-in predicate' error message, which can be very puzzling.

It would be permitted to define a predicate with the same functor and a different arity, e.g. **write/3** but this is definitely best avoided.

Simplifying Entry of Goals

In developing or testing programs it can be tedious to enter repeatedly at the system prompt a lengthy sequence of goals such as

?-dog(X),large(X),write(X),write(' is a large dog'),nl.

A commonly used programming technique is to define a predicate such as **go/0** or **start/0**, with the above sequence of goals as the right-hand side of a rule, e.g.

```
go:-dog(X),large(X),write(X),
   write(' is a large dog'),nl.
```

This enables goals entered at the prompt to be kept brief, e.g.

?-go.

Recursion

An important technique for defining predicates, which will be used frequently later in this book, is to define them in terms of themselves. This is known as a *recursive definition*. There are two forms of recursion.

(a) Direct recursion. Predicate **pred1** is defined in terms of itself.

(b) Indirect recursion. Predicate **pred1** is defined using **pred2**, which is defined using **pred3**, …, which is defined using **pred1**.

The first form is more common. An example of it is

```
likes(john,X):-likes(X,Y),dog(Y).
```

which can be interpreted as 'john likes anyone who likes at least one dog'.

Predicates and Functions

The use of the term 'predicate' in Prolog is closely related to its use in mathematics. Without going into technical details (this is not a book on mathematics) a predicate can be thought of as a relationship between a number of values (its arguments) such as *likes(henry,mary)* or *X=Y*, which can be either true or false.

This contrasts with a *function*, such as 6+4, *the square root of* 64 or *the first three characters of* 'hello world', which can evaluate to a number, a string of characters or some other value as well as true and false. Prolog does not make use of functions except in arithmetic expressions (see Chapter 4).

2.3 Loading Clauses

There are two built-in predicates that can be used to load clauses into the Prolog database: **consult/1** and **reconsult/1**. Both will cause the clauses contained in a text file to be loaded into the database as a side effect. However, there is a crucial difference between them, which is illustrated by the following example. Supposing file file1.pl contains

```
dog(fido).
dog(rover).
dog(jane).
dog(tom).
dog(fred).
cat(mary).
cat(harry).
small(henry).
large(fido).
large(mary).
large(tom).
large(fred).
large(steve).
large(jim).
```

and file file2.pl contains

```
dog(henry).
dog(fido).
cat(bill).
cat(steve).
large(mike).
large_animal(X):- dog(X),large(X).
large_animal(Z):- cat(Z),large(Z).
```

then entering the two goals

?-consult('file1.pl').

?-consult('file2.pl').

in succession at the prompt will put these clauses in the database.

```
dog(fido).
dog(rover).
dog(jane).
dog(tom).
dog(fred).
dog(henry).
dog(fido).
cat(mary).
cat(harry).
cat(bill).
cat(steve).
small(henry).
large(fido).
large(mary).
large(tom).
large(fred).
large(steve).
large(jim).
large(mike).
large_animal(X):- dog(X),large(X).
large_animal(Z):- cat(Z),large(Z).
```

Effectively, the clauses loaded from the second file are added to those already loaded from the first file, predicate by predicate, after those already there. Note that **dog(fido)** now appears in the database twice. There is nothing in the Prolog system to prevent this.

By contrast, entering the two goals

?-consult('file1.pl').
?-reconsult('file2.pl').

in succession at the prompt will put these clauses in the database.

```
dog(henry).
dog(fido).
cat(bill).
cat(steve).
small(henry).
large(mike).
large_animal(X):- dog(X),large(X).
large_animal(Z):- cat(Z),large(Z).
```

This is most unlikely to be what is intended. The predicate definitions in *file2.pl* completely replace any previous clauses for the same predicates in the database.

New predicates are loaded in the usual way. In the above example:

- the definitions of **dog/1**, **cat/1** and **large/1** replace those already in the database
- the definition of **small/1** in *file1.pl* remains in the database
- the definition of **large_animal/1** in *file2.pl* is placed in the database.

Although this example shows **reconsult** at its most unhelpful, in normal program development **reconsult** is routinely used. Some program developers may choose to load a large program in several parts (taking care that they have no predicates in common) using several **consult** goals, but a far more common method of program development is to load an entire program (set of clauses) as a single file, test it, then make changes, save the changes in a new version of the file with the same name and reload the clauses from the file. For this to work properly it is imperative to ensure that the old versions of the clauses are deleted each time. This can be achieved by using **consult** the first time and then **reconsult** each subsequent time.

The predicates **consult** and **reconsult** are used so frequently that in many versions of Prolog a simplified notation is available, with **['file1.pl']** standing for **consult('file1.pl')** and **[-'file1.pl']** standing for **reconsult('file1.pl').**

2.4 Variables

Variables can be used in the head or body of a clause and in goals entered at the system prompt. However, their interpretation depends on where they are used.

Variables in Goals

Variables in goals can be interpreted as meaning 'find values of the variables that make the goal satisfied'. For example, the goal

?-large_animal(A).

can be read as 'find a value of A such that **large_animal(A)** is satisfied'.

A third version of the Animals Program is given below (only the clauses additional to those in Animals Program 2 in Section 2.1 are shown).

```
/* Animals Program 3  */
/* As Animals Program 2 but with the additional rules
given below */
chases(X,Y):-
   dog(X),cat(Y),
   write(X),write(' chases '),write(Y),nl.
/* chases is a predicate with two arguments*/
go:-chases(A,B).
/* go is a predicate with no arguments */
```

A goal such as

?-chases(X,Y).

means find values of variables *X* and *Y* to satisfy **chases(X,Y)**.

To do this, Prolog searches through all the clauses defining the predicate **chases** (there is only one in this case) from top to bottom until a matching clause is found. It then works through the goals in the body of that clause one by one, working from left to right, attempting to satisfy each one in turn. This process is described in more detail in Chapter 3.

The output produced by loading Animals Program 3 and entering some typical goals at the prompt is as follows.

?-consult('animals3.pl').	*System prompt*
# 0.01 seconds to consult animals3.pl	*animals3.pl loaded*
?- chases(X,Y).	User backtracks to find first two solutions only.
fido chases mary	
X = fido ,	
Y = mary ;	Note use of *write* and *nl* predicates
fido chases harry	
X = fido ,	
Y = harry	

?-chases(D,henry). **no**	Nothing chases henry
?-go. **fido chases mary** **yes**	Note that no variable values are output. (All output is from the *write* and *nl* predicates.) Because of this, the user has no opportunity to backtrack.

It should be noted that there is nothing to prevent the same answer being generated more than once by backtracking. For example if the program is

```
chases(fido,mary):-fchasesm.
chases(fido,john).
chases(fido,mary):-freallychasesm.
fchasesm.
freallychasesm.
```

The query **?-chases(fido,X)** will produce two identical answers out of three by backtracking.

?- chases(fido,X).
X = mary ;
X = john ;
X = mary
?-

Binding Variables

Initially all variables used in a clause are said to be *unbound*, meaning that they do not have values. When the Prolog system evaluates a goal some variables may be given values such as *dog*, -6.4 etc. This is known as *binding* the variables. A variable that has been bound may become unbound again and possibly then bound to a different value by the process of *backtracking*, which will be described in Chapter 3.

Lexical Scope of Variables

In a clause such as

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

the variables *X* and *Y* are entirely unrelated to any other variables with the same name used elsewhere. All occurrences of variables *X* and *Y* in the clause can be

replaced consistently by any other variables, e.g. by *First_person* and *Second_person* giving

```
parent(First_person,Second_person):-
    father(First_person,Second_person).
```

This does not change the meaning of the clause (or the user's program) in any way. This is often expressed by saying that the *lexical scope* of a variable is the clause in which it appears.

Universally Quantified Variables

If a variable appears in the head of a rule or fact it is taken to indicate that the rule or fact *applies for all possible values of the variable*. For example, the rule

```
large_animal(X):-dog(X),large(X).
```

can be read as 'for all values of X, X is a large animal if X is a dog and X is large'.

Variable X is said to be *universally quantified*.

Existentially Quantified Variables

Suppose now that the database contains the following clauses:

```
person(frances,wilson,female,28,architect).
person(fred,jones,male,62,doctor).
person(paul,smith,male,45,plumber).
person(martin,williams,male,23,chemist).
person(mary,jones,female,24,programmer).
person(martin,johnson,male,47,solicitor).
man(A):-person(A,B,male,C,D).
```

The first six clauses (all facts) comprise the definition of predicate **person/5**, which has five arguments with obvious interpretations, i.e. the forename, surname, sex, age and occupation of the person represented by the corresponding fact.

The last clause is a rule, defined using the **person** predicate, which also has a natural interpretation, i.e. 'for all *A*, *A* is a man if *A* is a person whose sex is male'. As explained previously, the variable *A* in the head of the clause (representing forename in this case) stands for 'for all *A*' and is said to be *universally quantified*.

What about variables *B*, *C* and *D*? It would be a very bad idea for them to be taken to mean 'for all values of *B*, *C* and *D*'. In order to show that, say, *paul* is a man, there would then need to be **person** clauses with the forename *paul* for all possible surnames, ages and occupations, which is clearly not a reasonable

requirement. A far more helpful interpretation would be to take variable *B* to mean 'for at least one value of *B*' and similarly for variables *C* and *D*.

This is the convention used by the Prolog system. Thus the final clause in the database means 'for all *A*, *A* is a man if there a person with forename *A*, surname *B*, sex male, age *C* and occupation *D*, for at least one value of *B*, *C* and *D*'.

By virtue of the third **person** clause, *paul* qualifies as a man, with values *smith*, *45* and *plumber* for variables *B*, *C* and *D* respectively.

?- man(paul).
yes

The key distinction between variable *A* and variables *B*, *C* and *D* in the definition of predicate **man** is that *B*, *C* and *D* do not appear in the head of the clause.

The convention used by Prolog is that if a variable, say *Y*, appears in the body of a clause but not in its head it is taken to mean 'there is (or there exists) at least one value of Y'. Such variables are said to be *existentially quantified*. Thus the rule

```
dogowner(X):-dog(Y),owns(X,Y).
```

can be interpreted as meaning 'for all values of X, X is a dog owner if there is some Y such that Y is a dog and X owns Y'.

The Anonymous Variable

In order to find whether there is a clause corresponding to anyone called *paul* in the database, it is only necessary to enter a goal such as:

?- person(paul,Surname,Sex,Age,Occupation).

at the prompt. Prolog replies as follows:

Surname = smith ,
Sex = male ,
Age = 45 ,
Occupation = plumber

In many cases it may be that knowing the values of some or all of the last four variables is of no importance. If it is only important to establish whether there is someone with forename *paul* in the database an easier way is to use the goal:

?- person(paul,_,_,_,_).
yes

The underscore character _ denotes a special variable, called the *anonymous variable*. This is used when the user does not care about the value of the variable.

If only the surname of any people named *paul* is of interest, this can be found by making the other three variables anonymous in a goal, e.g.

?- person(paul,Surname,_,_,_).
Surname = smith

Similarly, if only the ages of all the people named *martin* in the database are of interest, it would be simplest to enter the goal:

?- person(martin,_,_,Age,_).

This will give two answers by backtracking.

Age = 23 ;
Age = 47

The three anonymous variables are not bound, i.e. given values, as would normally be expected.

Note that there is no assumption that all the anonymous variables have the same value (in the above examples they do not). Entering the alternative goal

?- person(martin,X,X,Age,X).

with variable *X* instead of underscore each time, would produce the answer

no

as there are no clauses with first argument *martin* where the second, third and fifth arguments are identical.

Chapter Summary

This chapter introduces the two types of Prolog clause, namely facts and rules and their components. It also introduces the concept of a predicate and describes different features of variables.

Practical Exercise 2

(1) Type the following program into a file and load it into Prolog.

```
/* Animals Database */
animal(mammal,tiger,carnivore,stripes).
animal(mammal,hyena,carnivore,ugly).
animal(mammal,lion,carnivore,mane).
animal(mammal,zebra,herbivore,stripes).
animal(bird,eagle,carnivore,large).
animal(bird,sparrow,scavenger,small).
animal(reptile,snake,carnivore,long).
animal(reptile,lizard,scavenger,small).
```

Devise and test goals to find (a) all the mammals, (b) all the carnivores that are mammals, (c) all the mammals with stripes, (d) whether there is a reptile that has a mane.

(2) Type the following program into a file

```
/* Dating Agency Database */
person(bill,male).
person(george,male).
person(alfred,male).
person(carol,female).
person(margaret,female).
person(jane,female).
```

Extend the program with a rule that defines a predicate **couple** with two arguments, the first being the name of a man and the second the name of a woman. Load your revised program into Prolog and test it.