Prentice Hall International
Series in Computer Science

C. A. R. Hoare, Series Editor

BACKHOUSE, R. C., *Program Construction and Verification*
BACKHOUSE, R. C., *Syntax of Programming Languages: Theory and practice*
DE BAKKER, J. W., *Mathematical Theory of Program Correctness*
BIRD, R., and WADLER, P., *Introduction to Functional Programming*
BJÖRNER, D., and JONES, C. B., *Formal Specification and Software Development*
BORNAT, R., *Programming from First Principles*
BUSTARD, D., ELDER, J., and WELSH, J., *Concurrent Program Structures*
CLARK, K. L., and MCCABE, F. G., *micro-Prolog: Programming in logic*
CROOKES, D., *Introduction to Programming in Prolog*
DROMEY, R. G., *How to Solve it by Computer*
DUNCAN, F., *Microprocessor Programming and Software Development*
ELDER, J., *Construction of Data Processing Software*
GOLDSCHLAGER, L., and LISTER, A., *Computer Science: A modern introduction (2nd edn)*
HAYES, I. (ED.), *Specification Case Studies*
HEHNER, E. C. R., *The Logic of Programming*
HENDERSON, P., *Functional Programming: Application and implementation*
HOARE, C. A. R., *Communicating Sequential Processes*
HOARE, C.A.R., and SHEPHERDSON, J. C. (EDS), *Mathematical Logic and Programming Languages*
INMOS LTD, *occam Programming Manual*
INMOS LTD, *occam 2 Reference Manual*
JACKSON, M. A., *System Development*
JOHNSTON, H., *Learning to Program*
JONES, C. B., *Systematic Software Development using VDM*
JONES, G., *Programming in occam*
JONES, G., *Programming in occam 2*
JOSEPH, M., PRASAD, V. R., and NATARAJAN, N., *A Multiprocessor Operating System*
LEW, A., *Computer Science: A mathematical introduction*
MACCALLUM, I., *Pascal for the Apple*
MACCALLUM, I., *UCSD Pascal for the IBM PC*
MEYER, B., *Object-oriented Software Construction*
PEYTON JONES, S. L., *The Implementation of Functional Programming Languages*
POMBERGER, G., *Software Engineering and Modula-2*
REYNOLDS, J. C., *The Craft of Programming*
SLOMAN, M., and KRAMER, J., *Distributed Systems and Computer Networks*
TENNENT, R. D., *Principles of Programming Languages*
WATT, D. A., WICHMANN, B. A., and FINDLAY, W., *ADA: Language and methodology*
WELSH, J., and ELDER, J., *Introduction to Modula-2*
WELSH, J., and ELDER, J., *Introduction to Pascal (3rd edn)*
WELSH, J., ELDER, J., and BUSTARD, D., *Sequential Program Structures*
WELSH, J., and HAY, A., *A Model Implementation of Standard Pascal*
WELSH, J., and MCKEAG, M., *Structured System Programming*
WIKSTRÖM, Å., *Functional Programming using Standard ML*

INTRODUCTION TO PROGRAMMING IN PROLOG

Danny Crookes
The Queen's University of Belfast,
Northern Ireland

PRENTICE HALL
NEW YORK LONDON TORONTO SYDNEY TOKYO

2

Representing facts in Prolog

A Prolog statement *describes* something, or a state of affairs, rather than giving a command to do something. It is a statement of a fact which is timeless. In the simple form of Prolog statement which we introduce in this chapter, a statement is unconditional: it states that something is true, all the time, and under all circumstances. Let us start with some simple examples.

2.1 Stating simple facts

Suppose we want to describe relationships between people in a group, in which one of the people, Fred, has set his heart on another person in the group, Elizabeth. The fact that Fred likes Elizabeth would be written in Prolog as:

```
likes (fred, elizabeth).
```

In general, something which we would say in English as:

subject verb object

is written in Prolog as:

verb (subject, object)

or, more generally, as:

relationship (object1, object2)

In the one Prolog fact which we have so far, `likes` is the relationship, and `fred` and `elizabeth` are both objects. A relationship is sometimes called a *predicate* in Prolog, and the objects which are related are called *arguments* or *parameters*. A statement of a fact in Prolog is called a *clause*.

As another example, suppose we wish to state which make of car various people drive. In a statement such as:

Michael drives a Jaguar

the verb or relationship is *drives*. In Prolog terminology, the predicate name is `drives`, and the arguments are the objects `michael` and `jaguar`. This gives the following Prolog clause which expresses the above fact:

```
drives (michael, jaguar).
```

Note that there can be several clauses for the same predicate. Thus, for instance, we could define the following clauses, which also state who drives what:

```
drives (elizabeth, aston_martin).
drives (fred, massey_ferguson).
```

These state that Elizabeth drives an Aston Martin, while Fred has a more agricultural means of transport.

Some details about defining facts

When actually typing Prolog facts into the computer, it is important to note the following details:

- The names of predicates and objects must begin with a lower-case letter rather than with a capital letter. Thus you should type `fred` and not `Fred`.
- A fact must be terminated with a full stop.
- Note that the underscore character '_' can be used in the middle of a name to make the name more readable.

Defining properties

Not all facts in English are relationships between two objects. Sometimes we wish to make statements such as:

Elizabeth is rich

or

Joe is intelligent

which do not have the standard '*subject–verb–object*' structure. But this is not a problem in Prolog, because a predicate can have any number of arguments, even just one. Thus these facts can be expressed as:

```
rich (elizabeth).
intelligent (joe).
```

These are examples of facts which state that a particular object has a certain *property*. The property acts as the relationship. The general structure of statements like this is:

property (*object*)

Numbers as objects

An object can be a *name* which we are free to make up ourselves. Examples which we have already met are `jaguar` or `elizabeth`. Alternatively, objects can also be *numbers* (whole numbers, called integers). Relationships including numbers are defined like any other relationship. For instance, the age of a person can be defined as a simple fact; if Fred is 37 years old, while Mary happens to be a youthful 19, then this information can be defined by two Prolog clauses:

```
age (fred, 37).
age (mary, 19).
```

Or, the price of items in a store could be defined in a similar way. If a radio costs $19 and a TV costs $350, this would be stated in Prolog as:

```
price (radio, 19).
price (tv, 350).
```

Symmetric relationships

In our very first example, note that the fact that Fred likes Elizabeth in no way implies that Elizabeth likes Fred. This lack of symmetry is a feature of Prolog as well as of real life. Thus in general, the Prolog statement:

relationship (*object1*, *object2*)

is quite different from, and does not imply:

relationship (*object2*, *object1*)

If the relationship we are defining is in fact symmetrical, in Prolog we have to define both aspects, as two separate facts. For instance, to state that Jeff and Pam are married to each other would require two clauses in Prolog:

```
married_to (jeff, pam).
married_to (pam, jeff).
```

We know that one must imply the other, but Prolog does not, since it has no idea of what the relationship `married_to` means. In fact, Prolog does not understand the meaning of *any* predicate names or objects which the programmer makes up. So it would not object to the clause:

```
fly (pigs).
```

or even to meaningless clauses like:

```
nargle (bargle, fargle).
```

A Prolog program for an employment agency

Imagine an employment agency which holds on the one hand a collection of job vacancies, and on the other hand a set of trained people looking for employment. The fact that an employer has a vacancy for someone with a particular skill or training can be defined in Prolog by a predicate:

```
has_vacancy (employer, skill)
```

For instance, the fact that NASA has a vacancy for a programmer is defined in Prolog as:

```
has_vacancy (nasa, programmer).
```

Also, the fact that a person looking for a job has been trained in a particular skill can be defined by a Prolog predicate:

```
trained_as (person, skill)
```

So the fact that Elizabeth has been trained as a secretary is defined in Prolog as:

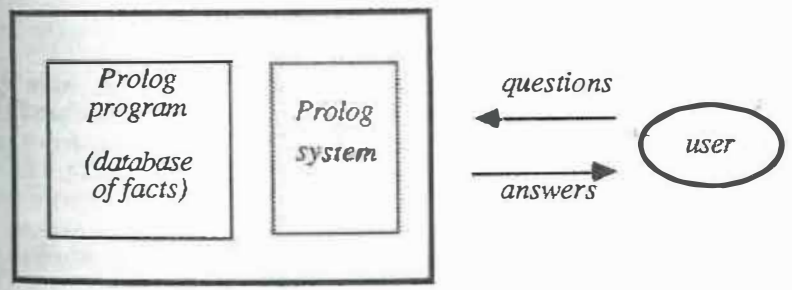
```
trained_as (elizabeth, secretary).
```

Now suppose that our employment agency has a set of vacancies and potential trained employees; these can now be defined by the following set of Prolog facts:

```
has_vacancy (harvard, secretary).
has_vacancy (prentice_hall, author).
has_vacancy (ibm, salesman).
has_vacancy (hertz, driver).
has_vacancy (nasa, programmer).
has_vacancy (prentice_hall, secretary).

trained_as (elizabeth, secretary).
trained_as (michael, salesman).
trained_as (michael, programmer).
trained_as (joe, driver).
trained_as (mary, secretary).
trained_as (fred, taxidermist).
```


This is in fact a Prolog program, and we will be referring to it throughout this chapter. It is just a small database. On its own, it is not a command to *do* anything: it merely states facts and describes things. It has to be typed into the computer so that it can be used later on. The manual of the particular Prolog system you are using will explain how to do this. The exact details of this tend to vary from one Prolog system to another and are not, strictly speaking, part of the language itself.

We do not usually speak of *running* a program like this, as might be the case with other programming languages. Instead, we ask questions, called *queries*. When the Prolog system receives a query, it consults the facts in the program, using logical reasoning to answer the query, and displays its answers on the screen. The Prolog system corresponds to the simple logical reasoning mechanism of an intelligent database, which was mentioned in Chapter 1. It is sometimes referred to as the *system*, or even just *Prolog*.



There are different types of query, depending on what sort of answer we require.

2.2 Questions with a yes/no answer

When we have a question which can be phrased in the terms:

Is it true that ... ?

we really have a *theory* which we would like either verified or disproved by the computer, on the basis of the facts in the program. For instance, in English we might wish to ask:

Is it true that Elizabeth is trained as a secretary?

Expressed in Prolog, our theory is that:

```
trained_as (elizabeth, secretary).
```

To put this theory, called a *query*, to the Prolog system, we merely enter it whenever the system is ready to receive the next query. The system indicates this by displaying the system 'prompt'. Again, the exact form of this prompt will vary depending on the version of Prolog in use; we will use:

```
?-
```

Whenever this prompt appears, it means that the system is ready and waiting to receive our next query. So our query about Elizabeth would appear as:

```
?-trained_as (elizabeth, secretary).
```

The general form of this type of query is:

```
?- fact.
```

The Prolog system takes this query and looks up its database (the program), to try to see if *fact* is true. In effect, Prolog takes the fact in the query as a theory which it has to prove, using the program as evidence, and applying logical reasoning. For our example question, Prolog will go through all the definitions of `trained_as`, in the order in which they are held, looking for the first one which matches the one in the question. In our example, the first definition of `trained_as` matches straightaway, so the system will display the answer:

```
yes
```

Thus a user who was unable to examine the actual program would now know that Elizabeth is trained as a secretary. If the user now wanted to know if Fred is trained as a secretary, then the question the user should ask is:

```
?-trained_as (fred, secretary).
```

Given the facts about `trained_as` in the program above, the system would not be able to prove this to be the case, and would display:


```
no
```

This does not mean that the program contained positive evidence that Fred has had *no* secretarial training. A `no` answer merely indicates that the theory could not be proved from the given facts in the program. There were no facts in the program which matched the one supplied in the query. Anything which the program does not state to be true is assumed to be false.

Here are some more examples to give practice in formulating simple queries. We assume that all the example facts given to date have been input and are part of the program.

(i) *Does Michael drive a Jaguar?* In Prolog:

```
?-drives (michael, jaguar).
```

(ii) *Does IBM have a vacancy for a programmer?* In Prolog:

```
?-has_vacancy (ibm, programmer).
```

(iii) *Is Mary rich?* In Prolog:

```
?-rich (mary).
```

(iv) *Is Fred really 37?* In Prolog:

```
?-age (fred, 37).
```

Compound queries

The form of the query can be more complex than just seeking verification of a simple fact. We can ask compound queries of the form:

Is it true that ... and ... and ... ?

In Prolog, this query is phrased using a comma for 'and'; sometimes the 'and' operation is referred to as *conjunction*:

```
?- fact1, fact2, ...
```

For instance, suppose our employment agent wishes to know if Mary has any chance of getting a job with IBM as a secretary. The query the agent should ask is:

```
?-trained_as (mary, secretary),
  has_vacancy (ibm, secretary).
```

This query consists of two parts, sometimes called *subgoals*, and the overall theory to be proved is then called the *goal*. For the goal to be true, or to *succeed*, both subgoals must succeed. The system would attempt to prove this goal in two stages. The first subgoal:

```
trained_as (mary, secretary)
```

would be considered first. Once this is proved (as it will be), the second subgoal is attempted. Given the facts in our program, this second subgoal will fail, since IBM are not recorded as having any vacancies for secretaries. Thus, the overall attempt to prove this compound query will fail, and the system will display the answer:

```
no
```

If the question had been asked the other way round:

```
?-has_vacancy (ibm, secretary),
  trained_as (mary, secretary).
```

then the answer will obviously be the same; however, the train of reasoning will be slightly different. The system will again begin by taking the first subgoal, which in this case is:

```
has_vacancy (ibm, secretary)
```

and will find that this fails. There is now no need to consider the second subgoal at all, since it cannot alter the overall result. So Prolog will return the answer `no` immediately upon failing at the first stage.

We can have as many conditions as we like in a query. For instance, the question:

Is Fred a rich programmer who drives an Aston Martin?

could be reformulated as:

Is Fred rich, and
is Fred a programmer, and
does Fred drive an Aston Martin?

This would then be expressed in Prolog as the query:

```
?-rich (fred),
  programmer (fred),
  drives (fred, aston_martin).
```


More general queries

All the queries so far have been very specific, involving facts about named objects. In answering a query, the Prolog system has looked for an exact match of the supplied goal. Often, though, we need a bit more flexibility. Suppose, for instance, that our employment agent is still trying to get Mary a job as a secretary. Having not had any luck with IBM, the agent would now like to ask:

Does any employer have a vacancy for a secretary?

If a query can include only specific, named employers, then this question has to be presented as a series of queries:

Does Harvard have a vacancy for a secretary?
Does Prentice Hall have a vacancy for a secretary?
...

for every employer in the database. There are two reasons why this is not a very good idea:

(i) For a large database, the whole process becomes very tedious.
(ii) What if the agent does not actually *know* the names of all the employers in the database?

The agent's question could be expressed in Prolog if we could present a query such as:

```
?-has_vacancy  (does not matter who occupies this position,
                secretary).
```

To indicate that the first position in the relationship `has_vacancy` does not matter to us, Prolog allows us to use a special object:

```
?-has_vacancy (_, secretary).
```

The special token '_' stands for 'anything', and is called in Prolog an *anonymous variable*. It is a *variable* because it is free to take on any value which would lead to successful matching of the query with the facts in the program. It will match with *any* object. It is *anonymous* because it does not have a name. We will see soon that Prolog variables usually do have names.

In answering this query, the Prolog system will attempt to match the given 'fact':

```
has_vacancy (_, secretary).
```

with the facts in the program in turn. Any fact which does not have `secretary` as the second object will not match. However, the

anonymous variable will match any object. If a match is found, then the system will display the usual answer:

```
yes
```

If no match is found, the answer `no` will be displayed. Thus, asking a query which contains the anonymous variable is really asking:

Can a value be given to the anonymous variable such that ... ?

Given our earlier question:

```
?-has_vacancy (_, secretary).
```

the system will go through each fact in the program in turn, starting from the top, until a matching fact is encountered. A matching fact is one which has anything in the first position, and `secretary` in the second. In this case, the query is satisfied by the fact:

```
has_vacancy (harvard, secretary).
```

so the answer `yes` is returned. If no matching fact had been found (because no definition of `has_vacancy` has `secretary` in the second position), then the system would have returned the answer `no`.

A few more example queries might help to consolidate the ground covered so far. We give first the English formulation of the query, and then how it would be expressed in Prolog:

(i) *Does NASA have any vacancies?* In Prolog, this would be:

```
?-has_vacancy (nasa, _).
```

(ii) *Does anyone have a vacancy for a driver?* In Prolog:

```
?-has_vacancy (_, driver).
```

(iii) *Do Fred and Joe both drive a Porsche?* In Prolog:

```
?-drives (fred, porsche),
  drives (joe, porsche).
```

(iv) *Are Elizabeth and Mary both trained as secretaries?* In Prolog:

```
?-trained_as (elizabeth, secretary),
  trained_as (mary, secretary).
```


2.3 Getting more than a yes/no answer

Usually when we ask a question, we would like more than a straight 'Yes' or 'No' answer. Imagine stopping someone and asking them '*Do you have the time?*'. They pause, look at their watch, answer '*Yes*', and walk on. That is where we end up by using only the facilities we have at present. For instance, if our agent asked the question in Prolog:

```
?-has_vacancy (_, secretary).
```

and received a positive answer, then the agent would quite like to know who it is who has the vacancy. But because the variable supplied in this query is anonymous, we cannot get at this information. What the agent would really like to ask is:

Is there an employer who has a vacancy for a secretary, and if so, who is it?

This query can be presented using a new feature of Prolog, as:

```
?-has_vacancy (Employer, secretary).
```

Note: In this query, `Employer` is actually another *variable*, except this time it has a name, unlike the anonymous variable. The Prolog system knows that `Employer` is a variable rather than an object because its name begins with a capital letter.

Initially, the variable `Employer` is free to take on any value which would bring about a successful match, just like the anonymous variable. However, once it has been matched with something in the first position of a matching `has_vacancy` fact, it is from that moment set, or *bound*, to that value; it will not match any other thereafter. Think of the matching process as taking a goal and a program statement, and *trying* to make them equal. To achieve this, the Prolog system can give whatever value is necessary to free variables, but cannot change the value of set, or *bound*, variables. In Prolog terminology, this matching process is called *unification*.

When the Prolog system finds a solution to a query, it will print out the values given to variables used in the query. This is how we get specific answers back. Only if the query contains no variables does the system respond just with the simple answer `yes`. Thus, given our earlier set of facts defining the relationship `has_vacancy`, the Prolog system would respond to the above query with the answer:

```
Employer = harvard
```

It worked this out by going through the facts for `has_vacancy` in the program, trying to match each in turn with the query. Remember that a variable which has not yet been bound to a specific object will match with

any object. The first fact succeeds, binding `Employer` to `harvard`, and displaying the value given to the variable `Employer`.

At this point, the system will pause, and wait for further instructions from the user. The reason it waits is that the user may either be satisfied with this *single* answer or may, on the other hand, want to know if there are any more answers to the query (i.e. if there are any more employers who have a vacancy for a secretary). If no more answers are required, the user at this point merely hits the *Return* key on the keyboard. However, if the user is interested in more answers, the user indicates this by typing a semicolon before hitting the *Return* key. This will cause the system to carry on *from where it left off*, continuing the matching process.

So if we wished to discover all those who have vacancies for a secretary, we would keep asking for more answers. This would be done by typing semicolon after each new set of answers, until the answer `no` finally came up. This would result in the following interaction:

```
?-has_vacancy (Employer, secretary).

Employer = harvard;
Employer = prentice_hall;
no
```

Note the need to be careful about beginning object names with a lower-case letter, and variables with an upper-case letter. This is a common source of error when defining Prolog facts or submitting queries. For instance, if by mistake we entered the query:

```
?-trained_as (fred, Secretary).
```

the system would respond with the rather confusing answer:

```
Secretary = taxidermist
```

because it has interpreted `Secretary` as a variable and not as an object.

Using variables to look up information stored in the program is fairly straightforward, as the following examples show. The answers given in each case assume that the definitions of the predicates in question which have been presented earlier in this chapter have been added to the program.

(i) *What age is Mary?* In Prolog:

```
?-age (mary, A).
```

This gives the answer:

```
A = 19
```

(ii) *What price is a radio?* In Prolog:


```
?-price (radio, P).

P = 19
```

(iii) *Who drives which sort of car?* Here, both arguments are variables:

```
?-drives (Driver, Make).
```

Based on the three earlier facts about `drives`, the answer would be:

```
Driver = michael
Make = jaguar;
Driver = elizabeth
Make = aston_martin;
Driver = fred
Make = massey_ferguson;
no
```

Using variables in compound queries

The use of variables in a query can be particularly helpful when the query is compound. A compound query is one with several parts or subgoals, all of which must be true for the query to succeed. For instance, if our agent wishes to know if there is an employer who has a vacancy for someone with Elizabeth's skills, then this query would be phrased as:

```
?-has_vacancy (Employer, Skill),
  trained_as (elizabeth, Skill).
```

This is read as: is there an employer (called `Employer`) who has a vacancy for someone with a particular skill (called `Skill`), and has Elizabeth been trained in this skill? This would produce the response:

```
Employer = harvard
Skill = secretary;
Employer = prentice_hall
Skill = secretary
```

Note: The use of the same variable name (`Skill`) in both subgoals means that the skill in each case must be the same. This is a general rule in Prolog: the use of the same variable name at two different places within a query means that the objects which eventually occupy those positions must be the same object.

As another example, suppose we wanted to know if Joe and Michael drive the same sort of car. This would be expressed in Prolog as:

```
?-drives (joe, Car), drives (michael, Car).
```

As it happens, if the program contains only the definitions of `drives` given at the start of this chapter, Michael and Joe do not drive the same sort of car; so the system would respond just with a `no` answer.

Let us take a few more example queries. We assume that in each case all possible answers to the query are wanted; so the system will return as many different substitutions for the variables as it can find. These examples are also used to raise some other issues, so they will not be completely straightforward.

(i) *Are there any matches of vacancies with potential employees?* In Prolog, this would be phrased as:

```
?-has_vacancy (Employer, Skill),
  trained_as (Person, Skill).
```

and would produce the answers beginning with:

```
Employer = harvard
Skill = secretary
Person = elizabeth;
Employer = harvard
Skill = secretary
Person = mary;
Employer = ibm
Skill = salesman
Person = michael;
...
```

(ii) *What competition does Mary have in looking for a job?* We might define a competitor of Mary as someone who has been trained in a skill in which Mary has also been trained. Phrasing this in Prolog gives:

```
?-trained_as (mary, Skill),
  trained_as (Competitor, Skill).
```

Given our existing program defining how each person has been trained, this would produce the answer:

```
Skill = secretary
Competitor = elizabeth;
Skill = secretary
Competitor = mary;
no
```

The strange thing is that Mary comes out as one of her own competitors! How has this come about? Substituting `mary` for

`Competitor` in the query certainly causes the query to be satisfied. The fact is, we have not been as precise in our definition of a competitor as we should have been. Strictly speaking, a competitor of Mary is someone *else* who has been trained in the same skill as Mary, where by *else* we mean someone who is not Mary. Thus we need a third condition, stating that `Competitor` is not `mary`. Saying that something is *not* the case in Prolog can be a thorny philosophical problem, since Prolog facts state what is true rather than what is false. Leaving this issue aside, however, in Prolog we state that something is not the case by writing the condition:

```
not (fact)
```

This succeeds if *fact* fails, and fails if *fact* succeeds. So to express the third condition above, that `Competitor` is not `mary`, we would write:

```
not (Competitor = mary)
```

The entire query then becomes:

```
?-trained_as (mary, Skill),
  trained_as (Competitor, Skill),
  not (Competitor = mary).
```

A word of warning about using '='. Testing if two objects are equal involves testing if they are the *same object*. This is not as straightforward as it seems; for instance, what is the effect of testing if an object is equal to a variable which is still free, and has not yet got a value? Thus, '=' in Prolog is defined rather differently, in such a way that:

`mary = mary`	is *true*;
`mary = elizabeth`	is *false*;
`mary = X`	is *true* if `X` is `mary`; *false* if `X` has any other value; *true* if `X` is free (and `X` will be set to `mary` in the process)

Think of '=' as trying to make its two sides equal. It is actually the same as the usual Prolog matching process of unification.

Finding answers in order

When all possible answers to a query are asked for, the answers can sometimes be a little surprising. Perhaps some of the answers above

surprised you at first sight. Understanding why the system responds the way it does requires some knowledge of the internal operation of the Prolog system and its logical reasoning process; this will be dealt with in more detail in Chapter 3, but some information is given here to explain a few of the effects which can arise.

Finding all possible answers is a bit like finding all combinations. For instance, suppose you were asked to list all the combinations of the two letters A and B, allowing duplications. There are four in all:

```
A A
A B
B A
B B
```

When these are listed methodically, the left-most letter is the slowest-changing one. It is chosen first, and held constant until all possible ways of filling in the remainder of the answer are found. Then we go back to the start, try a different choice for the first position, and hold this constant while the different ways of completing the remainder of the answer are found. This continues until the list is exhausted.

This process is not unlike the way Prolog tries to find all possible answers to a compound query. If a query has two subgoals, the first one is attempted first. Once a solution to the first part is found (perhaps setting variables in the process), these variables are held constant until all possible solutions to the second part are found. Then the system retraces its steps, goes back to the first subgoal again, and tries for a different solution to this first part. This second attempt at the first part could result in new values being given to the variables. Then, with these new variable values held constant, all possible solutions to the second part are found. This process continues until no new solutions to the first part can be found, at which point the system answers `no`. The process by which the system retraces its steps and goes back to a previous subgoal looking for a possible new solution to it, is called *backtracking*.

To illustrate this process, consider a query which asks for all possible pairs of employers and candidates for any vacancy as a secretary:

```
?-has_vacancy (Employer, secretary),
  trained_as (Candidate, secretary).
```

The answer to this query will be all combinations of the variables `Employer` and `Candidate` (and note that the query introduces them in that order), such that the given condition holds. The system finds these combinations by taking each solution to the first subgoal in turn; then, for each of these, it finds all solutions to the second subgoal. Since there are just two solutions to the first part (`Employer = harvard`, and `Employer = prentice_hall`), the combinations will be found in two groups: first, all those with `Employer = harvard`; then those with `Employer = prentice_hall`:


```
{ First, with Employer = harvard, find all solutions to
  trained_as (Candidate, secretary)                    }

Employer = harvard
Candidate = elizabeth;
Employer = harvard
Candidate = mary;

{ Now with Employer = prentice_hall, find all solutions to
  trained_as (Candidate, secretary)                    }

Employer = prentice_hall
Candidate = elizabeth;
Employer = prentice_hall
Candidate = mary
```

2.4 A geography database

Figure 2.1 shows a map of part of Europe, though it is considerably simplified, and even inaccurate. It contains some details of regions, seas and mountains. The information on the map is to be stored as a Prolog program, so that anyone can ask questions of the system about what is on the map. The sorts of simple question to be asked might be:

- What regions are there?
- What mountains are there, and which regions are they in?
- Which regions border another given region?
- Which seas border a given region?

The information given by the map can be stated by simple Prolog relationships. It is good practice to do this in two stages:

(i) Define the basic components on the map.
(ii) Define the relationships between all these basic components.

Firstly, the basic components. There are three different types of component — regions, seas and mountains. The fact that France, say, is a region (rather than a sea or a mountain) could be written as:

```
region (france).
```

This needs to be done for all the regions on the map, giving the following facts:

```
region (france).
region (germany).
region (switzerland).
region (austria).
```

```
region (italy).
region (sicily).
```

The seas can similarly be defined, using a predicate *sea*:

```
sea (english_channel).
sea (mediterranean).
sea (adriatic).
```

Figure 2.1 *Map of Europe*

Lastly, the mountains need to be defined:


```
mountains (alps).
mountains (pyrenees).
mountains (apennines).
```

So far, only the basic components have been defined; there is nothing to say which region borders which, or which region contains which mountains. This information is added to the program by defining the various relationships which exist between the basic components. The relationships which the map shows are:

- The fact that one region borders another.
- The fact that a sea touches the coast of a particular region.
- In which region each mountain range is contained.

The fact that, say, France borders Germany can be stated in Prolog as:

```
borders (france, germany).
```

But since Prolog does not realise that this naturally means that Germany borders France, a second fact must be added to this effect:

```
borders (germany, france).
```

This needs to be done for all eight borders, giving sixteen facts in total:

```
borders (france, germany).
borders (france, switzerland).
borders (france, italy).
borders (germany, france).
borders (germany, switzerland).
borders (germany, austria).
borders (switzerland, france).
borders (switzerland, germany).
borders (switzerland, austria).
borders (switzerland, italy).
borders (austria, germany).
borders (austria, switzerland).
borders (austria, italy).
borders (italy, france).
borders (italy, switzerland).
borders (italy, austria).
```

We also need to define which seas touch the coast of which region:

```
touches_coast_of (english_channel, france).
touches_coast_of (mediterranean, france).
touches_coast_of (mediterranean, italy).
touches_coast_of (mediterranean, sicily).
touches_coast_of (adriatic, italy).
```

Finally, we define which regions contain which mountains:

```
contains (france, pyrenees).
contains (switzerland, alps).
contains (italy, apennines).
```

This has defined all the basic information which the map gives. So it should now be possible to extract this information by asking queries. Here are some examples:

(i) *What seas are there?* In Prolog, this would be formulated as:

```
?-sea (Sea).
```

and would produce the following answers (if all answers were requested):

```
Sea = english_channel;
Sea = mediterranean;
Sea = adriatic;
no
```

(ii) *What mountains are there, and which region is each range in?* In Prolog, this is:

```
?-mountains (Mountains),
  contains (Region, Mountains).
```

Actually, the first of these subgoals might be regarded as being redundant, since mountains can only be contained in a country anyway. As it stands, the query would produce the following response (if all solutions were asked for):

```
Mountains = alps
Region = switzerland;
Mountains = pyrenees
Region = france;
Mountains = apennines
Region = italy;
no
```

Other information which is not stated explicitly can be worked out from the database of facts, by asking compound queries:

(iii) *Which regions are islands?* The program does not state this explicitly, but it can be worked out logically, if we assume that an island is a region which has no border with another region. If a region has no land borders, then:


```
?-borders (Region, _).
```

should fail. However, to get something which *succeeds* if there are no borders rather than fails, we have to make use of the built-in Prolog `not` predicate, where:

```
not (fact)
```

succeeds if *fact* fails, and vice versa. Therefore, to find all islands, we would present the query:

```
?-region (Island),
  not (borders (Region, _)).
```

This would produce the answer:

```
Island = sicily;
no
```

A final note on Prolog's matching of queries

The Prolog system answers queries by matching patterns between the query and the stored database (the program). This matching process is methodical and mechanical, and does not require the system to *understand* what either the program or the query means. For instance, given the fact:

```
likes (fred, elizabeth).
```

the system has no idea who or what `fred` and `elizabeth` are, and does not have the faintest notion what `likes` means. It merely notes that a relationship called `likes` exists between two objects called `fred` and `elizabeth`, in that order. Answering queries can be done without understanding, though. For instance, someone in Iceland might write some facts from a saga as the following Prolog clauses:

```
elska (helgi, unn).
elska (hrafnkel, bergthora).
elska (helgi, bergthora).
```

If the following query was presented:

```
?-elska (X, bergthora).
```

then, irrespective of what this all means, and without appreciating the blood that may have been spilt because of the situation described by these facts, it should be fairly obvious that the answer will come back:

```
X = hrafnkel;
X = helgi;
no
```

Because no attempt is made to understand what is input, Prolog is quite prepared to accept what we would regard as either a mistake, or as utter nonsense. For instance, if by mistake we entered the fact:

```
fred (likes, elizabeth).
```

then Prolog would interpret this blindly as stating in English that:

likes fred elizabeth

(whatever it means *to fred* something). Or, it would happily accept the following facts:

```
nosh (gurble, nurdle).
jikker (snuffit, yaxxen).
```

Prolog's simple built-in pattern matching reasoning process can carry on regardless in the face of information which to us would be totally confusing.

EXERCISES

1 A database of hobbies

The fact that a person has a certain hobby, or enjoys a particular activity, can be expressed in Prolog by defining a relation such as:

```
enjoys ( person, activity )
```

For instance, the fact that Fred enjoys playing baseball would be stated as:

```
enjoys (fred, baseball).
```

Take an imaginary group consisting of the following people:

Margaret, Nancy, Jane, Denis, Ronald and Robert

These individuals enjoy various activities selected from the following list:

music, soccer, baseball, reading, travel and painting

For each of these hypothetical people, draw up a list of activities which you imagine them to enjoy. Write each of these as a Prolog fact, stating that the person enjoys that particular activity or hobby. Naturally, a person can have more than one hobby. Define up to three or four activities for each person.

How would you then ask questions such as the following in Prolog:

(i) *What hobbies does Robert have?* Ask if there is an activity such that Robert enjoys it, and if so, what is it. You can get all his activities by repeatedly asking for more solutions.

(ii) *Do Ronald and Margaret have any common interests?* Ask if there is an activity which they both enjoy.

(iii) *Is there anyone who enjoys soccer and baseball, but does not enjoy music?*

2 A family tree

Relationships within a family can be defined in terms of the following simple predicates:

```
married_to ( person1, person2 )
child_of ( person1, person2 )
male ( person )
female ( person )
```

which, respectively, state that:

person1 is married to *person2*
person1 is a child of *person2*
person is male
person is female

For instance, a database of a family with parents Denis and Margaret, and son Mark, would be defined by the facts:

```
male (denis).
male (mark).
female (margaret).

married_to (denis, margaret).
married_to (margaret, denis).

child_of (mark, denis).
child_of (mark, margaret).
```

Now consider the following well-known family tree shown in Figure 2.2. Using only the above predicates, write the information in this tree as a Prolog program.

How would you then ask the following questions:

(i) *Who are the children of Anne?*

(ii) *Who are the parents of William?*
(iii) *What brothers does Edward have?* A brother of Edward is someone who has the same parents as Edward does, and who is also male.
(iv) *Who are the grandparents of Peter?* A grandparent is a parent of a parent.

This exercise will be taken up in more detail in Chapter 3.

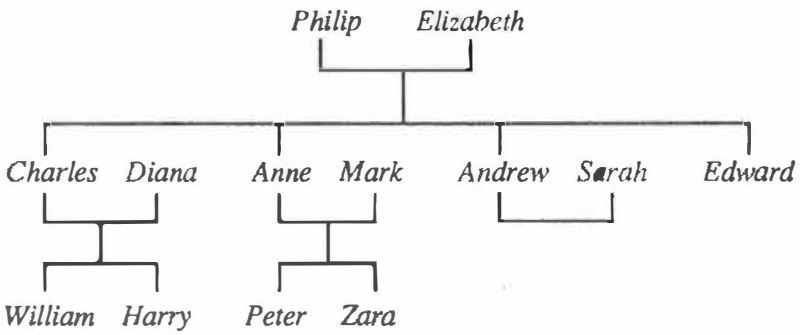


Figure 2.2 *Family tree*

3 Another geography database

We want a program which does the same job for a map of the British Isles as was done for the earlier European geography database.

The information included on the physical map of the British Isles is as follows:

- For the sake of argument, call the regions England, Scotland, Wales, Ulster, Eire, and Isle of Man.
- The seas which touch the coasts of the British Isles are the English Channel, the North Sea, the Irish Sea and the Atlantic.
- The mountains to be included are: the Pennines (in England), the Mournes (in Ulster), and the Grampians (in Scotland).

Define the basic components on the map, and then the relationships between all these basic components. Once the program is complete, ask the following queries:

(i) *What regions are there?*
(ii) *Which regions are land-locked?* A land-locked region has no coast-line.
(iii) *Which regions have a coastline and mountains?*

3

General rules in Prolog

So far, all our Prolog statements have been blunt statements of fact, which state that something is true, all the time, and under all circumstances. Unfortunately, life is not quite as simple as this. Often in real life, a statement is true only under certain circumstances. To make such a statement in Prolog, we have to be able to qualify the statement by giving the conditions under which it holds. This chapter is about how to do this by writing general rules in Prolog.

3.1 Writing Prolog rules

In English, something which is conditional (as opposed to an unconditional fact) is often stated in the terms:

*such-and-such is true **if** some condition holds*

Let us say, for instance, that an employer is looking for a person with certain *qualities*, rather than with a specific training. For example, if NASA happens to be looking for someone who is clear-thinking and accurate, then this information might be written as the rule:

*NASA might employ someone **if***
that person is clear-thinking and accurate

In Prolog, a statement can be made to be conditional by qualifying it using *if*, written as ':-' in Prolog, followed by the conditions. For instance, our general rule about who NASA might employ would be written as:

```
might_employ (nasa, X) :-
     clear_thinking (X),
     accurate (X).
```

A rule of this form states that provided the condition(s) on the right-hand side hold, then the conclusion on the left is valid. A fact as described in Chapter 2 is just a special case of a Prolog rule, being one with no conditions. In general, the right-hand side can contain several conditions, all of which must be true for the conclusion to be valid:

conclusion :- *condition*$_1$, *condition*$_2$, ...

Whenever a query is asked, the system makes use of the rules, and follows them logically to work out the answer. Thus, if the question is asked:

```
?-might_employ (Employer, joe).
```

then the system selects the first rule for `might_employ`. If this happens to be the above rule about NASA, then the system interprets this rule as saying: to check if NASA might employ Joe, check first if Joe is clear-thinking; if so, check if he is accurate. Only if both conditions (or subgoals) are satisfied does the query succeed. Note that the reasoning process is just the same as when answering a compound query.

Likewise, a candidate might also wish to place certain conditions on a prospective employer. For instance, to state that Michael is prepared to work for an employer only if the employer is hi-tech and small, we could define the Prolog rule:

```
prepared_to_work_for (michael, Employer) :-
     hi_tech (Employer),
     small (Employer).
```

Rules can themselves be used to define other rules, such as in:

```
match_possible (Employer, Candidate) :-
     might_employ (Employer, Candidate),
     prepared_to_work_for (Candidate,
                           Employer).
```

Checking a goal of this kind can lead to a chain of reasoning which gets deeper each time another rule is referenced. For instance, for the goal:

```
?-match_possible (nasa, michael).
```

the first subgoal will be tackled first:

```
might_employ (nasa, michael)
```

This subgoal is checked by selecting the rule which defines the conditions under which NASA might employ someone. Thus, this subgoal will in turn be broken down into two further subgoals:


```
clear_thinking (michael)
```

and

```
accurate (michael)
```

Provided each of these succeeds, the system will then proceed to the second of the two original subgoals:

```
prepared_to_work_for (michael, nasa)
```

This is likewise broken down into its two constituent subgoals, by applying the rule which defines who Michael is prepared to work for. This chain of reasoning can become quite involved, and difficult to follow. So it will help to trace the checking of a goal by drawing it out systematically. The following is one way of doing this:

```
match_possible (nasa, michael)  ?
     might_employ (nasa, michael)  ?
          clear_thinking (michael)  ?
          - succeeds
          accurate (michael) ?
          - succeeds
     prepared_to_work_for (michael, nasa)  ?
          hi-tech (nasa) ?
          - succeeds
          small (nasa) ?
          - fails
     - fails
- fails
```

3.2 Alternative clause definitions

We have already noted that Prolog allows us to build a compound query or goal using 'and' (written using a comma). Often, though, we wish to write clauses using 'or', such as:

Joe likes Mary or Elizabeth

More generally, let us say we want to make statements of the form:

a likes b or c

One way of achieving this is to make two separate statements:

```
a_likes_b_or_c (A, B, C) :- likes (A, B).
a_likes_b_or_c (A, B, C) :- likes (A, C).
```

It should be clear that each of these statements on its own is true, although not necessarily the *whole* truth. If A likes B, then the conclusion that A likes either B or C is certainly valid, as the first clause states. If A likes C, then it likewise follows that A likes B or C (which is what the second clause says). There is no need to think that we have to state everything about a relationship in a single clause.

From the system's point of view, if it is asked:

```
?-a_likes_b_or_c (joe, mary, elizabeth).
```

then it will try to apply the first definition of the relation `a_likes_b_or_c`, and check that:

```
likes (joe, mary)
```

If Joe does not like Mary, then this will fail. But all is not lost, because the system will now look for alternative definitions of the predicate `a_likes_b_or_c`, and only admit failure when all definitions have been exhausted. Thus, the next attempt will be to try the second definition of `a_likes_b_or_c`, which will be broken down into a check that:

```
likes (joe, elizabeth)
```

If this succeeds, then the initial query will succeed.

In summary, then, the effect of an 'or' is achieved by writing separate clauses for each different case, where a clause can be either a fact or a conditional rule. During query answering, the different clauses will be tried in the order in which they were defined, until one succeeds or until all have been tried.

The fact that different definitions of the same predicate are tried in a fixed order means that the earlier definitions have some sort of priority. This usually shows up only when the clause is used to find a solution rather than to check one. For instance, if we want to find someone who likes either Mary or Elizabeth, we would ask:

```
?-a_likes_b_or_c (Person, mary, elizabeth).
```

and wait for the first answer. Since the first definition to be tried will always be:

```
likes (Person, mary)
```

we are more likely to be given someone who likes Mary, rather than Elizabeth, even though the question probably intended no bias. When generating all solutions by repeatedly typing semicolon, the order of the definitions is shown up in the order of the different answers. This feature is not usually a problem, and indeed can be used to advantage, as the next example shows.

3.3 Example: employment agency revisited

Let us change the ground rules a little for our simple employment agency program. Instead of looking only for someone who has already been trained in a particular skill, employers may also be prepared to do the training themselves. They are therefore also on the look-out for candidates who have the potential to become the sort of person the employer needs.

An employer's position might be stated by saying that, if the employer has a vacancy, then the first preference is for someone who *has* been trained in the required skill; but if no trained person is available, then someone who has the necessary personal qualities would be acceptable. This means that there are really two alternative conditions under which a person is acceptable to an employer to do a given job. One condition is that the candidate has already been trained; the other is that the candidate has not been trained, but could be. In Prolog, we will have a separate clause to cover each of these alternative cases. And since the first case is preferred, then the clause for the first case is defined first. We want to define a predicate:

```
acceptable (Candidate, Employer, Skill)
```

which states the conditions under which `Candidate` is acceptable to `Employer` to do a job requiring the given skill. The two alternative definitions of this predicate are therefore as follows:

```
acceptable (Candidate, Employer, Skill) :-
     has_vacancy (Employer, Skill),
     trained_as (Candidate, Skill).

acceptable (Candidate, Employer, Skill) :-
     has_vacancy (Employer, Skill),
     not (trained_as (Candidate, Skill)),
     could_be_trained_as (Candidate, Skill).
```

The predicates `has_vacancy` and `trained_as` would be defined by simple statements of fact, and not as conditional rules. The remaining predicate:

```
could_be_trained_as (Candidate, Skill)
```

is however conditional: it depends on whether or not `Candidate` has the necessary personal qualities to learn `Skill`. To define the predicate

`could_be_trained_as` fully, it is necessary to define the required qualities for each skill separately. This might give us the following set of clauses:

```
could_be_trained_as (X, secretary) :-
     accurate (X),
     literate (X),
     out_going (X).

could_be_trained_as (X, programmer) :-
     clear_thinking (X),
     accurate (X),
     intelligent (X).

could_be_trained_as (X, driver) :-
     co_ordinated (X),
     hard_working (X).

could_be_trained_as (X, salesman) :-
     out_going (X),
     hard_working (X).

could_be_trained_as (X, author) :-
     imaginative (X),
     literate (X).
```

Selecting a clause

Note that when a program contains multiple clauses defining a predicate like this, and the system is given the goal:

```
could_be_trained_as (X, Y)
```

then it will automatically begin by trying to apply the first definition (thus looking to see if X could be trained as a secretary). If this does not succeed, it will try subsequent definitions in order, until one succeeds or until all the definitions have been tried.

Note that applying a rule (as distinct from just matching a fact) takes place in two stages, both of which must succeed:

(i) The goal is matched with the left-hand side of the rule. The objects in the goal must all match the templates in the rule. This is the same unification process as used in matching the goal with unconditional facts. Thus, an attempt to match:

```
could_be_trained_as (X, driver)
```


with the first definition of `could_be_trained_as` will fail, because `driver` does not match `secretary`.

(ii) If the left-hand side matches, then the conditions are checked, in order.

Let us now complete our employment agency program. To obtain a complete program for our revised employment agency, we now only have to add factual information about what vacancies there are, who has been trained as what, and the personal qualities of each of the candidates. The following set of facts might describe a typical situation:

```
has_vacancy (harvard, secretary).
has_vacancy (prentice_hall, author).
has_vacancy (ibm, salesman).
has_vacancy (hertz, driver).
has_vacancy (nasa, programmer).
has_vacancy (prentice_hall, secretary).

trained_as (michael, programmer).
trained_as (fred, taxidermist).
trained_as (mary, secretary).

accurate (elizabeth).
accurate (mary).
accurate (michael).
accurate (fred).

out_going (michael).
out_going (mary).
out_going (elizabeth).

co_ordinated (joe).

hard_working (mary).
hard_working (joe).
hard_working (michael).

clear_thinking (elizabeth).

intelligent (mary).

imaginative (michael).
```

The following are some example queries which could be presented to this program, and the answers which would be produced assuming all solutions are requested:

(i) *Who is acceptable to Hertz?* In Prolog:

```
?-acceptable (Candidate, hertz, _).

Candidate = joe;
no
```

(ii) *To which employers is Michael acceptable, and for which jobs?* In Prolog:

```
?-acceptable (michael, Employer, Job).

Employer = nasa
Job = programmer;
Employer = ibm
Job = salesman;
no
```

(iii) *Are any employers competing for the same candidate?*

```
?-acceptable (Candidate, E1, _),
  acceptable (Candidate, E2, _),
  not (E1 = E2).
```

If we know from the outset that this is likely to be a reasonably common query, then it would be better to define who an employer's competitors are *within the program*. The following is a predicate which states the conditions under which an employer `E1` has a competing employer `E2` over a given candidate:

```
competitor (E1, E2, Candidate) :-
     acceptable (Candidate, E1, _),
     acceptable (Candidate, E2, _),
     not (E1 = E2).
```

The query which a user has to enter is now much simpler. For instance, to discover which employers might be fighting over Elizabeth, the query would be asked:

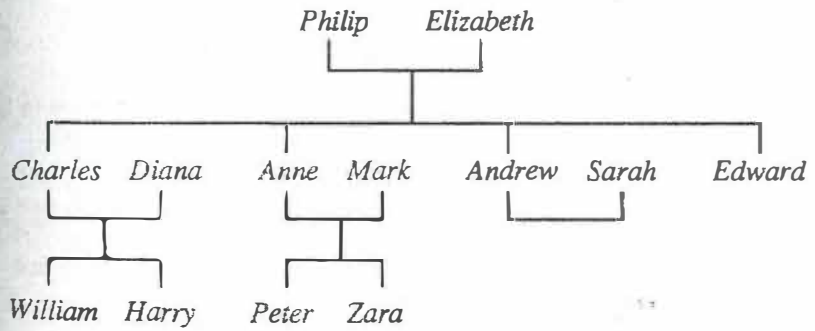
```
?-competitor (E1, E2, elizabeth).

E1 = harvard
E2 = prentice_hall;
E1 = prentice_hall
E2 = harvard;
no
```

Note here the effect of the Prolog system being unable to recognise that different orderings of the same employers are not really new solutions. Prolog cannot recognise that the relation `competitor` is symmetric.

This approach of defining extra predicates to facilitate common queries is not only more convenient for the user: it is also safer. Where possible, any relations which are likely to be required by a user should be identified beforehand, and defined within the program like this.

3.4 Example: family relationships

Being a member of a family automatically involves a person in all kinds of different relationships: child, parent, grandchild, brother, sister, cousin, mother-in-law, great-great-grandchild, and so on. We will show how these relationships can be defined in Prolog. As an example, consider the family tree for the well-known family mentioned in Exercise 2 at the end of the previous chapter:



The relationship between, say, Charles and Philip can be defined by the Prolog fact:

```
child_of (charles, philip).
```

which states that Charles is a child of Philip. But the relationship between Charles and Philip is also one of father-to-son; so perhaps we should define this further relationship by another, separate fact:

```
parent_of (philip, charles).
```

Logically, there should be no need to state explicitly who is a parent of who if we have already defined the `child_of` relationship, since one can always be inferred from the other. The general rule for working out the

parent_of relationship given the child_of relationship can be written as the following Prolog rule:

```
parent_of (X, Y) :- child_of (Y, X).
```

This states that if Y is a child of X, then X is a parent of Y. Note that we could equally well have chosen to define parent_of as a set of facts, and then defined child_of in terms of parent_of. There is no real difference. In general, some relationships will be 'fundamental', being specified explicitly by facts; the rest can be deduced from other relationships, and are written as general rules. The fundamental relationships necessary to define a family tree are:

```
child_of (X, Y)        { defines 'vertical' relationships }
married_to (X, Y)      { defines 'horizontal' relationships }
```

Since these are genderless relations, we will also need to know whether a person is male or female. This will enable us to distinguish a father from a mother, for instance. This requires two further predicates:

```
male (X)
female (X)
```

All these predicates must be defined as specific facts for each person in the family:

```
male (philip).
male (charles).
female (elizabeth).
married_to (philip, elizabeth).
married_to (elizabeth, philip).
child_of (charles, philip).
child_of (charles, elizabeth).
...
```

All other relationships can be defined in terms of these basic predicates. We now consider some further relationships.

(i) *Fathers and mothers.* The predicate parent_of makes no distinction between father or mother; it really means father_or_mother_of. We can define a father to be a male parent, and a mother to be a female parent:

```
father_of (X, Y) :- parent_of (X,Y),
                    male (X).

mother_of (X, Y) :- parent_of (X,Y),
                    female (X).
```


(ii) *Brothers and sisters.* A *sibling* is the term used for a brother or sister. We might define siblings as people who have a common parent:

```
sibling_of (X, Y) :- parent_of (P, X),
                     parent_of (P, Y),
                     not (X = Y).
```

Note the need for the last condition (not (X = Y)). This prevents someone being defined as a sibling of themselves. This definition actually has a drawback: if a query asks for *all* siblings of X and Y, then each pair will be duplicated, since each is related by two different parents. So if in our example family we asked for the siblings of Edward:

```
?-sibling_of (edward, X).
```

the answer would be:

```
X = charles;
X = anne;
X = andrew;
X = charles;
X = anne;
X = andrew;
no
```

The definition as it stands really defines all combinations of X, Y *and* P such that the right-hand side is satisfied, and there are two values of P for each X and Y. An improvement would be to define a sibling as someone who shares the same father and mother:

```
sibling_of (X, Y) :- father_of (F, X),
                     mother_of (M, X),
                     father_of (F, Y),
                     mother_of (M, Y),
                     not (X = Y).
```

Now, a brother is a male sibling, and a sister is a female one:

```
brother_of (X, Y) :- sibling_of (X, Y),
                     male (X).
sister_of (X, Y)  :- sibling_of (X, Y),
                     female (X).
```

(iii) *Aunts, uncles and cousins.* An aunt is the sister of a parent; an uncle is the brother of a parent:

```
aunt_of (X, Y)  :- parent_of (P, Y),
                   sister_of (X, P).
```

```
uncle_of (X, Y) :- parent_of (P, Y),
                   brother_of (X, P).
```

A cousin is the child of an aunt or an uncle. To handle the 'or', we have the option either of writing two clauses for a cousin, or of defining an intermediate relation:

```
aunt_or_uncle_of (X, Y) :- aunt_of (X, Y).
aunt_or_uncle_of (X, Y) :- uncle_of (X, Y).
```

with a cousin then defined as:

```
cousin_of (X, Y) :- child_of (X, P),
                    aunt_or_uncle_of (P, Y).
```

The latter solution is probably better, since it may prove useful to have the relation `aunt_or_uncle_of` in defining other relationships.

We could now check, for instance, that Andrew is the uncle of Harry by asking:

```
?-uncle_of (andrew, harry).
```

or ask for all cousins of William, by:

```
?-cousin_of (william, Cousin).
```

This latter query would produce the response:

```
Cousin = peter;
Cousin = zara;
no
```

(iv) *Grandchildren and grandparents.* Since a grandchild is the child of a child, its Prolog definition is quite straightforward:

```
grandchild_of (X, Y) :- child_of (X, P),
                        child_of (P, Y).
```

This in turn can be used to define the grandparent relationship:

```
grandparent_of (X, Y) :- grandchild_of (Y,X).
```

Having these various relationships defined as part of the program greatly simplifies the user's task of formulating queries. Things should always be made as simple and convenient as possible for the user!

3.5 Recursive rules

Section 3.5 is STARRED material!

We already have Prolog rules for `child_of` and `grandchild_of`. But what about subsequent generations? Suppose we want to define the conditions under which someone is a *descendant* of someone else. How do we define a descendant? We might be tempted to say that a descendant is a child, or a grandchild, or a great-grandchild, and so on:

```
descendant_of (X, Y) :- child_of (X, Y).
descendant_of (X, Y) :- grandchild_of (X, Y).
descendant_of (X, Y) :- great_grandchild_of
                                        (X, Y).
...
```

where each generation would need to be defined separately:

```
grandchild_of (X, Y) :-
                  child_of (X, Z),
                  child_of (Z, Y).
great_grandchild_of (X, Y) :-
                  child_of (X, Z),
                  grandchild_of (Z, Y).
great_great_grandchild_of (X, Y) :-
                  child_of (X, Z),
                  great_grandchild_of (Z,Y).
...
```

Not only is this approach very tedious, but we must also stop sooner or later. This means that we will have defined descendants only up to a fixed number of generations, and so will not really have solved the problem completely. A clue to a better solution is to notice a pattern in the definition of each generation above: note that each new generation is defined in terms of the previous one. So consider a new definition of what is meant by a descendant:

The descendants of Y are Y's children, along with their descendants

Note that we have defined *descendant* in terms of itself. A rule which is defined in terms of itself is called *recursive*. To turn this into a Prolog rule, it needs to be rephrased to state when X is a descendant of Y:

X is a descendant of Y either if X is a child of Y,
or if X is a descendant of a child of Y

This can now be formulated as a Prolog predicate:

```
descendant_of (X, Y) :- child_of (X, Y).
descendant_of (X, Y) :- child_of (C, Y),
                        descendant_of (X, C).
```

If the following query was asked, to find descendants of Elizabeth as defined by the previous example of a family tree:

```
?-descendant_of (X, elizabeth).
```

then the answer would be:

```
X = charles;
X = anne;
X = andrew;
X = edward;
X = william;
X = harry;
X = peter;
X = zara;
no
```

When a recursive use of a rule is encountered by the system, it is treated just like a reference to any other rule. This is illustrated by tracing the behaviour of the system when given the query:

```
?-descendant_of (harry, elizabeth).
```

Using our earlier notation for tracing, this would give the following:

```
descendant_of (harry, elizabeth) ?

{ Try first clause: }
      child_of (harry, elizabeth) ?
      - fails

{ Try again with second clause for descendant_of: }

      child_of (C, elizabeth) ?
      - succeeds, setting C to charles

      descendant_of (harry, charles) ?

            child_of (harry, charles) ?
            - succeeds
      - succeeds
- succeeds
```

Terminating recursion

Somehow when one sees a recursive definition such as `descendant_of`, there is a nagging doubt in the back of the mind: might

the system not go on for ever, trying to answer a recursive query? There certainly are times when this can happen. Suppose, for instance, that a person called Dave likes anyone who likes him. We could write this in Prolog as:

```
likes (dave, X) :- likes (X, dave).
```

But suppose there are *two* indecisive people like this: Dorothy also likes anyone who likes her:

```
likes (dorothy, X) :- likes (X, dorothy).
```

If two indecisive people like this come into contact, problems can arise. So suppose the query is now asked: *does Dave like Dorothy?*

```
?-likes (dave, dorothy).
```

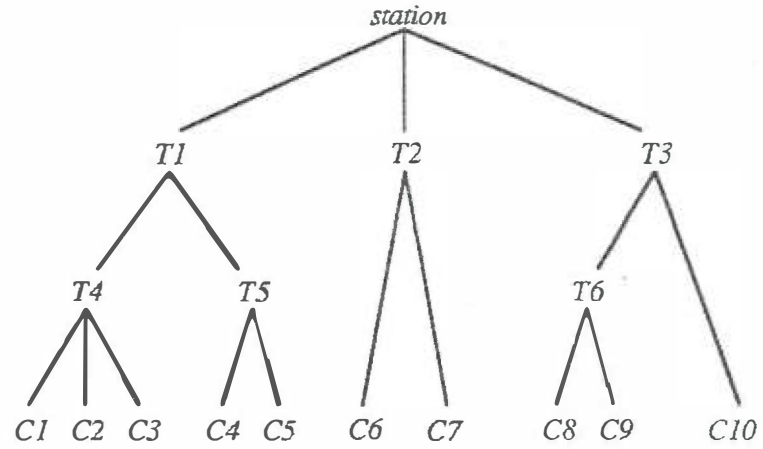
So Dave says he will like Dorothy if she likes him; and Dorothy says she will like Dave if he likes her. If we are not careful, we could end up oscillating between one and the other, waiting for one of them to make up their mind one way or the other. As humans, we are intelligent enough to recognise when we are in an infinite cycle like this, and can stop the process. But the Prolog system is not, and it will blindly alternate between asking if Dave likes Dorothy and asking if Dorothy likes Dave. So no answer would come back to this query: the computer would just sit there. In practice, it will eventually come back with a message to the effect that it has run out of memory.

There is no danger of this situation arising when working out descendants, however. To find the descendants of a person, we work down through that part of the family tree headed by the person in question. Since a person cannot be the parent of one of their ancestors, we are always going down the tree, and there can be no connections back to people higher up the tree. We must therefore eventually reach the bottom of the family tree, and stop that line of search. This can be seen from the definition of `descendant_of`: note that both clauses in its definition start with `child_of`. Eventually the system reaches the bottom of the tree, where a person has no children. At this point, `child_of` will fail. So because there are no 'loops' or cycles in a family tree, the process will eventually stop.

Section 3.6 is STARRED material!

3.6 Example: a power distribution network

Electricity consumers in a district are supplied with electricity from an electricity generating station. This power is distributed from the station to the various consumers through a system of transformers, as shown in the diagram:



C1 to C10 are consumers; T1 to T6 are transformers. From time to time, one of the transformers may malfunction, or may need to be taken out of service temporarily. If this should happen, the management would naturally wish to know which consumers will be affected. We will therefore write a simple Prolog program to help the management in their task.

The fact that one point in the distribution network directly feeds another point is shown on the diagram by a line connecting the two points. In Prolog, this same information could be defined by the fact:

```
feeds (X, Y).
```

Thus, the fact that the station feeds transformer T1 is defined by:

```
feeds (station, t1).
```

Doing this for every connection gives a representation of the entire network:

```
feeds (station, t1).
feeds (station, t2).
feeds (station, t3).
feeds (t1, t4).
feeds (t1, t5).
feeds (t3, t6).
feeds (t4, c1).
feeds (t4, c2).
feeds (t4, c3).
feeds (t5, c4).
feeds (t5, c5).
feeds (t2, c6).
```



```
feeds (t2, c7).
feeds (t6, c8).
feeds (t6, c9).
feeds (t3, c10).
```

Since we intend to ask questions which distinguish between consumers, transformers and the station, this information should also be defined:

```
generator (station).
transformer (t1).
transformer (t2).
transformer (t3).
transformer (t4).
transformer (t5).
transformer (t6).
consumer (c1).
consumer (c2).
consumer (c3).
consumer (c4).
consumer (c5).
consumer (c6).
consumer (c7).
consumer (c8).
consumer (c9).
consumer (c10).
```

The predicate `feeds` refers only to *direct* connections. But one point can feed or supply another *indirectly*. Note, for instance, that T1 supplies T4 and T5 and consumers C1 to C5. We therefore wish to define a predicate:

```
supplies (X, Y)
```

which states when point X supplies point Y either directly or indirectly. The conditions under which X supplies Y can be expressed as follows:

X supplies Y <u>either</u> if X feeds Y directly,
<u>or</u> if X feeds directly a point which supplies Y

Note that this is a recursive definition. It is a relatively simple matter to express this in Prolog:

```
supplies (X, Y) :- feeds (X, Y).
supplies (X, Y) :- feeds (X, Point),
                   supplies (Point, Y).
```

If we now want to know which consumers will be affected by a fault at one point in the network, we can use `supplies` to find out. For instance, if transformer T1 goes down, the query would be:

```
?-supplies (t1, C), consumer (C).
```

which would yield the answer:

```
C = c1;
C = c2;
C = c3;
C = c4;
C = c5;
no
```

The second condition in the query avoids the appearance of intermediate transformers in the list of points supplied by T1.

Alternatively, a consumer (C8, say) may ring up to report a power loss. To find out which of the transformers could be responsible for the power cut, the query could be asked:

```
?-supplies (T, c8), transformer (T).
```

which asks for which transformer supplies C8. This would produce the answer:

```
T = t6;
T = t3;
no
```

Again, we can be sure that this recursive definition of `supplies` will not cause the system to get into an infinite cycle, provided the network has no *feedback*. Feedback would be indicated by having a line going back up the diagram; this would form a closed path, which could be followed *ad infinitum*. Whenever the system reaches a consumer, `supplies` will fail, thus ending that line of recursive checking. A consumer does not feed any other point.

EXERCISES

1 Compatible interests

Using a predicate:

```
enjoys (person, activity)
```

make up and define a database of facts for an imaginary group which states who enjoys doing what. As in Exercise 1 in Chapter 2, the people in this group are:

Margaret, Nancy, Jane, Denis, Ronald and Robert

and the activities which they enjoy are selected from:

music, soccer, baseball, reading, travel and painting

If you did Exercise 1 in Chapter 2, your answer will do nicely for this.

Now, define a Prolog predicate:

```
compatible1 (X, Y)
```

which succeeds if the two people X and Y have at least one common activity which they enjoy. Then define:

```
compatible2 (X, Y)
```

which succeeds only if X and Y enjoy at least *two* common activities.

Suppose some activities require three players, and you wish to know if there are three members of the group who enjoy one of these activities, so that they can do it together. Write a rule:

```
three_for (Activity) :- ...
```

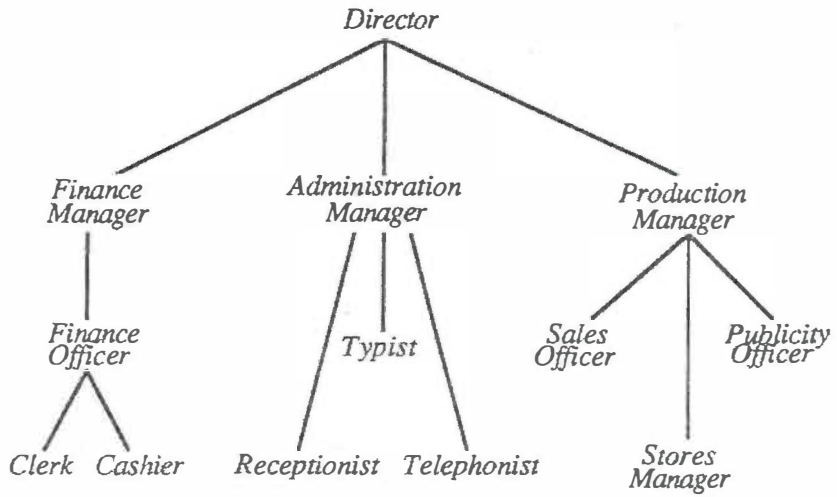
which succeeds only if at least three people in the group enjoy the given activity.

2 More family relationships

Using the predicates defined earlier in the chapter, write Prolog rules to define the following relationships:

```
nephew_of (X, Y) :- ...
neice_of (X, Y) :- ...
mother_in_law_of (X, Y) :- ...
brother_in_law_of (X, Y) :- ...
```

3 Management structures



The diagram shows the management structure for a firm. The fact that one person works for another is indicated in the diagram by one person being directly below the other. This fact will be stated in Prolog using the predicate:

```
works_for (X, Y)
```

Using this predicate, write a set of Prolog facts which defines the above management structure.

Now write Prolog rules for the following:

(i) *X is the (immediate) boss of Y.*
(ii) *X is under Y in the firm* (either directly or indirectly).
(iii) *X is over Y in the firm* (either directly or indirectly).