4.900

Prolog Tutorial 2

by Hui Wang

Prolog Tutorial 2

The contents of this manual describe the product, ***WIN-PROLOG***, version 4.500, and are believed correct at time of going to press. They do not embody a commitment on the part of Logic Programming Associates Ltd (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Logic Programming Associates Ltd
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England

phone: +44 (0) 20 8871 2016
fax: +44 (0) 20 8874 0449
web: http://www.lpa.co.uk

4 February, 2005

Contents

Introduction to Prolog

Introductory notes originated by Hui Wang (h.wang@ulst.ac.uk) and updated by Clive Spenser, LPA.

Introduces Prolog syntax and its relationship to logic.

Also introduces various built-in predicates.

Chapter 1 Introduction to Prolog

1.1 What is Logic Programming?

In conventional, sometimes called *imperative*, *procedural* or *algorithmic* programming the computer is given a detailed sequence of instructions - the code - which if carried out step by step in the order given will enable a task to be automated. The key features of this type of programming are that:

- programs require so much detail that they are difficult and time-consuming to write.
- they are error prone.
- they are difficult to read and therefore to understand and debug.
- programs are relatively inflexible - they can only be used directly for the purpose for which they are written. Changing or extending the program can often require considerable effort.

In this type of programming, the programmer can be likened to a manager, who, when she asks one of her staff to do something has to spell out every individual step. In real life a manager would issue a brief request and assume that the employee knew what was involved and how to go about the task. In other words it would be assumed that the employee had *knowledge* of how things are to be done and what is involved in a particular task and could make use of this as appropriate for the problem in hand.

Logic programming is motivated by the desire to bring this kind of managerial activity to programming. But whereas people gather knowledge in life and work as they go along, the computer must be provided with the basic knowledge needed for the tasks we wish it to perform.

To do this, a way must be found to write down or represent knowledge which can then be fed to the computer. Ideally the form that this knowledge takes should be very readable to humans and correspond to the way they naturally work with knowledge themselves.

The best way achieve this is to use a logic system. People generally - not just computer scientists - are familiar with logic statements and their manipulation and use them all the time.

A logic statement is just a sentence which can be true or false. Statements may be joined together using 'and' and 'or' to give more complex statements or can be negated. Much of our knowledge about the world can be captured in simple logical statements:

> Paris is the capital of France.
>
> The pass mark for every exam in the university is 40%

Two basic logic systems are **propositional logic** and **first order predicate logic** (**FOPL**).

Given some logical statements we can *reason* with them to *infer* or *prove* new

statements not in the original set, e.g. given:

- Eric Clapton is a successful rock guitarist.
- A rock guitarist is a rock musician.
- Successful rock musicians are rich.

we can infer that Eric Clapton is rich.

The aim of logic programming is to automate this kind of reasoning. A logic programming system has a language in which we can write down what we know (the knowledge) and an automated reasoning system to do the proving for us.

By giving the system the appropriate knowledge any problem for which you would normally write a program can be tackled - payroll, database, simulation, graphics.

An interesting feature of a logic program is that, unlike a procedural program, there is no clear distinction between programs and data. Knowledge includes both programs, i.e. 'how to do' type statements and data.

This is how it is in everyday life. We have routines in our heads such as how to get money out of an autobank. We also have data such as the PIN number of our card. The automatic reasoner can handle both of these knowledge types, often using the 'data' as input to the implications.

Another way to think of a logic program is as a recipe book containing a series of individual recipes. Some recipes may refer to other recipes. For example a recipe for apple tart might refer the reader to the section on making a pastry base. The recipe for steak and kidney pudding might also make use of this pastry recipe.

Logic programming is effective because:

- it is easier to read and modify the knowledge than it is to do the same for the instructions in a conventional ordinary program and
- the automated reasoning system saves having to write so much program code leaving time to concentrate on the problem.

Typically, a logic program can be written in about a tenth of the time that a procedural program takes to write and the code is also much shorter - again a tenth or even more.

Logic programming was developed, in the early 1970s, by the Artificial Intelligence community - originally to help with the machine processing of language. It is widely used in the academic world but also in industry by large organisations such as BT, Hewlett-Packard and British Aerospace.

The best-known logic programming language is **Prolog**.

1.2 The basics of Prolog

In the rest of this chapter we introduce the basics of Prolog: *objects, relations, arguments, predicates, facts, rules, procedures, queries, variables, conjunctions, disjunctions, negations, knowledge base, search and backtracking*. At the end of this chapter you should be able to read and write simple programs in Prolog. You should also be able to understand how your programs run in the Prolog system.

Prolog is a programming language suitable for solving problems that involves *objects* and the *relations* between objects. We use Prolog when we wish the computer to solve problems that can be expressed in the form of objects and their relations. For example, when we say

Tom likes the book.

we are declaring that a relation, *likes,* exists between an object *Tom* and another object *book*. The relation also indicates a specific order between objects: *Tom likes the book, but the book doesn't like Tom!* When we make the query

Does Tom like the book?

we are trying to find out about a relation.

We are all familiar with using rules to describe relations between objects. For example, the rule

Two people are sisters

if they are both female and

have the same parents.

tells us something about what it means to be sisters. It also tells us how to find out if two people are sisters: simply check to see if they are both female and have the same parents.

Prolog can be easily used to represent facts and rules about a given problem and then queries about the problem can be made to it. In general, programming in Prolog consists of:

- declaring some facts about objects and their relations,
- defining some rules about objects and their relations, and
- making queries about objects and their relations.

For example, suppose we told Prolog the rule about sisters. We could then make the query whether Mary and Jane are sisters. Prolog would search through what we told it about Mary and Jane, and come back with answer *yes* or *no*, depending on what we told it earlier.

In the following sections in this chapter, we will introduce briefly how to declare facts, define rules, make simple queries, and use variables, conjunctions, disjunctions and negations to make more complicated queries.

1.2.1 Declaring Facts

Suppose we want to tell Prolog the fact that

Tom likes Mary.

This fact consists of two objects *Mary* and *Tom*, and a relation *likes*. This fact can be expressed in Prolog as follows:

```
likes(tom,mary).
```

When we write Prolog facts like the above, we need to remember the following points:

1. The names of all relations and objects must begin with a lower-case letter. For example, **likes, tom, mary.**
2. The relation is written first. The objects are separated by commas and enclosed by a pair of round brackets.
3. The full stop character `.' must come at the end of a fact.

The following are some examples of facts, together with possible interpretations:

```
teaches(tom,prolog).
```
Tom teaches Prolog.

```
female(mary).
```
Mary is female.

```
sister(mary,jane).
```
Mary and Jane are sisters.

```
father(tom,mary).
```
Tom is the father of Mary.

```
man(socrates).
```
Socrates is a man.

In Prolog terminology, the names of the objects that are enclosed within the round brackets in each fact are called the arguments. The name of the relation is called the predicate. So, female is a predicate having one argument, and sister is a predicate having two arguments.

The names of the objects and relations are completely arbitrary. Instead of writing a fact in the form of father(tom,mary), we could just as well represent it as a(b,c), and remember that a means father, b means Tom, and c means Mary. We should, however, normally select names that help us to remember what they represent. We also need to decide in advance what our names mean, and what the order of arguments shall be. We must remain consistent in using the names.

Relations can have an arbitrary number of arguments. If we want to define a predicate play, where we mention two players and a game they play with each other, we need three arguments. Here are two examples of this:

```
play(tom,gerry,football).

play(jane,jim,badminton).
```

1.2.2 Making Queries

Once we have the facts, we can make some queries about them. In Prolog, a query looks just like a fact, except that we put a special symbol before it. The special symbol consists of a question mark and a hyphen. Consider the query:

```
?- likes(tom,book).
```

If we interpret tom to be a person called Tom, and book to be some particular book, this query is asking "Does Tom like the book?", or "Is it a fact that Tom likes the book?"

When a query is made to Prolog, it will search through the knowledge base that contains all facts we told it. It looks for facts that *match* the fact in the query. Two facts match if their predicates are the same, and if their corresponding arguments each are the same. If Prolog finds a fact that matches the query, Prolog will respond `yes`. If no such fact exists in the knowledge base, Prolog will respond `no`. Consider the following knowledge base:

```
likes(joe,fish).

likes(joe,mary).

likes(mary,book).

likes(john,book).
```

If we told Prolog all those facts, we could make the following queries, and Prolog would give the answers:

```
?- likes(joe,money).  NO

?- likes(mary,joe).   NO

?- likes(mary,book).  YES

?- king(john,france). NO
```

In Prolog, the answer `no` means *nothing matches the query*. In other words, Prolog does not know the answer to the query. It is important to remember that `no` is not the same as *false*.

1.2.3 Using Variables

If we want to find out what things that Tom likes, it is tiresome to ask

- Does Tom like books?
- Does Tom like Mary?

with Prolog giving a `yes-or-no` answer each time. It is more sensible to ask Prolog to tell us something that Tom likes. We could make a query of this form as,

```
Does Tom like X?
```

When Prolog uses a variable, the variable can be either instantiated or not instantiated. A variable is instantiated when there is an object that the variable stands for. A variable is not instantiated when what the variable stands for is not yet known.

Prolog can distinguish variables from names of particular objects because any name beginning with a capital letter is taken to be a variable.

When a query containing a variable is made to Prolog, Prolog searches through all the facts in its knowledge base to find an object that the variable could stand for. So when we ask

Does Tom like X?

Prolog searches through all its facts to find things that Tom likes. Consider the following knowledge base of facts:

```
likes(tom,flowers).

likes(tom,mary).

likes(paul,mary).
```

The query asks, *Is there anything that Tom likes?* and Prolog will respond as follows:

```
?- likes(tom,X).
  X = flowers ? ;

  X = mary ? ;

  no
```

How does this work? When this query is made to Prolog, the variable is initially not instantiated. Prolog searches through the knowledge base, looking for a fact that *matches* the query. Now if an uninstantiated variable appears as an argument in the query, Prolog will allow that argument to match any other argument in the same position in the fact. What happens here is that Prolog searches for any fact where the predicate is `likes`, and the first argument is `tom`. The second argument in this case may be anything, because the query was made with an uninstantiated variable as the second argument. When such a fact is found, then the variable `X` now stands for the second argument in the fact, whatever it may be. Prolog searches through the knowledge base in the order it was told so the fact `likes(tom,flowers)` is found first. Variable `X` now stands for the object `flowers`. Or, we say that `X` is instantiated to flowers. Prolog now marks the place in the knowledge base where a matching fact is found.

Once Prolog finds a fact that matches a query, it prints out the objects that the variables now stand for. In this case, the only variable was `X`, and it matches the object `flowers`. Now Prolog waits for further instructions. If we press the carriage return key, meaning we are satisfied with the answers we have so far, then Prolog will stop searching for more. If instead we press the semicolon key followed by the carriage return key, Prolog will resume its search through the knowledge base as before, starting from where it left the place-marker. When Prolog begins searching from a place-marker, instead of from the beginning of the knowledge base, we say that Prolog is attempting to *re-satisfy* the query.

Suppose in response to Prolog's first answer:

```
X=flowers
```

We asked it to carry on by typing the *semicolon (or spacebar)*. This means we want to

satisfy the query in another way; we want to find another object that X could stand for. Prolog then must `forget' that X stands for flowers, and resume searching with X uninstantiated again. Because we are searching for an alternative solution, the search is continued from the place-marker. The next matching fact found is likes(tom,mary).

The variable X is now *instantiated* to mary. Prolog will print

```
X=mary,
```

and wait for further instructions. If we relentlessly type another semicolon, Prolog will continue the search. In this example there is nothing more that Tom likes. So Prolog will stop its search, and allow us to make more queries or declare more facts.

1.2.4 Using Conjunctions, Disjunctions and Negations

Suppose we want to answer queries about more complicated relations, such as, *Do Tom and Mary like each other*? One way to do this would be first ask if Tom likes Mary, and if Prolog tells us yes, then we ask if Mary likes Tom. So, this problem consists of two separate *goals* that Prolog must satisfy. In Prolog, we can use a **conjunction** to satisfy these two goals. Suppose we have the following knowledge base:

```
likes(mary,food).

likes(mary,wine).

likes(tom,wine).

likes(tom,mary).
```

We want to ask if Tom and Mary like each other. To do this, we ask, "*Does Tom like Mary and does Mary like Tom*?" The *and* expresses the fact that we are interested in the *conjunction* of the two goals -- we want to satisfy them both one after another. We represent this by putting a comma between the goals:

```
?- likes(tom,mary), likes(mary,tom).
```

The comma is pronounced ``and'', and it serves to separate any number of different goals that have to be satisfied in order to answer a query. When a sequence of goals is given to Prolog, Prolog attempts to satisfy each goal in turn by searching for a matching fact in the knowledge base. All goals have to be satisfied in order for the sequence to be satisfied. Using the above list of facts, what should Prolog answer? The answer is no. It is a fact that Tom likes Mary, so the first goal is true. However, the second goal is not true, since there is nowhere in the list of facts where likes(mary,tom) occurs. Since we wanted to know if they *both* like each other, the whole query is answered no.

Conjunctions and the use of variables can be combined to make quite interesting queries. Now that we know that it cannot be shown that Tom and Mary like each other, we ask: *Is there anything that Tom and Mary both like*? This query also consists of two goals:

First, find out if there is some X that Mary likes.

Then, find out if Tom likes whatever X is.

In Prolog the two goals would be put in a conjunction like this:

```
?- likes(mary,X), likes(tom,X).
```

Notice the use of the same, shared variable to connect the two sub-queries.

Prolog answers the query by attempting to satisfy the first goal. If the first goal is satisfied by a fact in the knowledge base, then Prolog will mark the fact in the knowledge base, and attempt to satisfy the second goal. If the second goal is satisfied by another fact, then Prolog marks that fact in the knowledge base, and we have found a solution that satisfies both goals.

It is most important to remember that each goal keeps its own place-marker. If however, the second goal is not satisfied, then Prolog will attempt to re-satisfy the first goal. Remember that Prolog searches the knowledge base completely for each goal. If a fact in the knowledge base happens to match, satisfying the goal, then Prolog will mark the place in the knowledge base in case it has to re-satisfy the goal at a later time. But when a goal needs to be re-satisfied, Prolog will begin the search from the goal's own place-marker, rather than from the start of the knowledge base. Our above query *"Is anything liked by Mary also liked by Tom?"* illustrates an example of this **backtracking** behaviour in the following way:

The knowledge base is searched for the first goal.

The first matching fact is likes(mary,food).

So, now X is instantiated to food *everywhere* in the query where X appears. Prolog marks the place in the knowledge base where it found the fact, so it can return to this point in case it needs to resatisfy the goal. Furthermore, we need to remember that X became instantiated here, so Prolog can 'forget' X if we need to re-satisfy this goal.

Now, the knowledge base is searched for likes(tom,food). This is because the next goal is likes(tom,X), and X stands for food. As we can see, no such fact exists, so the goal fails. Now when a goal fails, we must try to re-satisfy the previous goal, so Prolog attempts to re-satisfy likes(mary,X), but this time starting from the place that was marked in the knowledge base. But first we need to make X uninstantiated once more, so X may match against anything.

The marked place is likes(mary,food), so Prolog begins searching after that fact. Because we have not reached the end of the knowledge base yet, we have not exhausted the possibilities of what Mary likes, and the next matching fact is likes(mary,wine). The variable X is now instantiated to wine, and Prolog marks the place in case it must re-satisfy likes(mary,X).

As before, Prolog now tries the second goal, searching this time for likes(tom,wine). Prolog is not trying to re-satisfy this goal. It is entering this goal again. So it must start searching from the beginning of the knowledge base. The matching fact likes(tom,wine) is found and Prolog marks its place in the knowledge base.

At this point, both goals have been satisfied. Variable X stands for the name wine. The first goal has a place-marker in the knowledge base at the fact likes(mary,wine), and the second goal has a place-marker in the knowledge base at the fact likes(tom,wine).

As with any other queries, as soon as Prolog finds one answer, it stops and waits for further instructions. If we type the semicolon key, Prolog will search for more things

that both Tom and Mary like.

There are some other ways of making complex queries. First of all, instead of ask whether a fact is true, we can ask whether its negation is false. Or the other way around. This can be done by using a special built-in predicate `not/1`. For example,

Does Mary like anything which Tom doesn't like?

```
?- likes(mary,X), not(likes(tom,X)).
```

Does Mary not like Tom?

```
?- not(likes(mary,tom)).
```

We can also make a query for some alternative answers. For example,

Does anybody like Tom or food?

```
?- likes(_,tom); lives(_,food).
```

In the above conjunction, two goals are separated by a semicolon. It means that the conjunction of goals is satisfied if either of the goals is satisfied.

1.2.5 Defining Rules

Suppose we want to state the fact that *'All men are fallible'*. One way to do this would be to write down separate facts like:

```
fallible(alfred).

fallible(tom).

fallible(charles).

fallible(socrates).

  .

  .

  .
```

for every man in the knowledge base. This could be tedious, especially if there are hundreds of people there. Another way to say that all men are fallible is to say, *Any object is fallible provided it is a man*. This statement is in the form of a *rule* about what are fallible, instead of listing all men who are fallible.

In Prolog, rules are used when we want to say that a fact depends on a group of other facts. In English, we can use the word *if* to express a rule. For example,

I use an umbrella if it rains.

Tom buys the wine if it is less expensive than the beer.

Rules are also used to express definitions, for example:

X is a bird if:

X is an animal, and

X has feathers.

or

X is a sister of Y if:

X is female, and

X and Y have the same parents.

It is important to remember that a variable stands for the same object whenever it occurs in a rule. A rule is a *general statement about objects and their relations*. We can allow a variable to stand for a different object in each different *use* of the rule.

In Prolog, a rule consists of a *head* and a *body*. The head and body are connected by the symbol ":-", which is made up of a colon ":" and a hyphen "-". The ":-" is pronounced *if*.

Now let us consider several examples.

Tom likes anyone who likes wine

or, in other words,

Tom likes something if it likes wine

or, with variables,

Tom likes X if X likes wine.

This example can be written in Prolog as:

```
likes(tom,X) :-
  likes(X,wine).
```

Notice that rules also are ended with a full stop. The head of this rule is `likes(tom,X)`. The head of the rule describes what fact the rule is intended to define. The body, in this case, `likes(X,wine)`, describes the conjunction of goals that must be satisfied, one after another, for the head to be true. For example, we can make Tom more choosy about whom he likes, simply by adding more goals onto the body, separated by commas:

```
likes(tom,X) :-
  likes(X,wine),
  likes(X,food).
```

Or, in words, *Tom likes anyone who likes both wine and food*. Or, suppose Tom likes any female who herself likes wine:

```
likes(tom,X) :-
  female(X),
  likes(X,wine).
```

In the above rule, whenever `X` becomes instantiated to some object, all `X`'s within the *scope of* `X` are also instantiated. The scope of `X` is the whole rule, including the head, and extending to the full stop at the end of the rule. So, if `X` happens to be instantiated to `mary`, then Prolog will try to satisfy the goals `female(mary)` and `likes(mary,wine)`.

Now let us consider an example of family relations.

```
parent(pam,bob).  /* Pam is a parent of Bob. */
```

```
parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).


female(pam). % Pam is female.

female(liz).

female(pat).

female(ann).


male(tom).

male(bob).

male(jim).
```

We use a rule to define the sister relation.

```
sister(X,Y) :-
  parent(Z,X),

        parent(Z,Y),

        female(X).
```

The above rule expresses that:

For any X and Y,

X is a sister of Y if

(1) both X and Y have the same parent, and

(2) X is a female.

The above program defines several relations -- `parent`, `female`, `male` and `sister`. The `parent` relation, for example, is defined by six clauses. We say that these six clauses are *about* the `parent` relation. Sometimes it is convenient to consider the whole set of clauses about the same relation. Such a set of clauses is called a **procedure**.

We use the predicate/arity notation to represent procedures. In the above program, there are procedures. They are represented in the predicate/arity notation as `parent/2`, `female`/1, `male`/1 and `sister`/2. For instance, `parent`/2 shows that a relation is defined by a procedure in which each of the clauses consists of a predicate

parent and two arguments.

Comments are, in general, ignored by the Prolog system. They only serve as a further clarification to the person who reads the program. Comments are distinguished in Prolog from the rest of the program by being enclosed in special brackets `/*' and `*/`.

Another method, more practical for short comments, uses the percent character `%'. Anything between `%' and the end of the line is interpreted as a comment.

Now let us make a query.

```
?- sister(ann,pat).
```

When this query is made to the above knowledge base and the rule for sister, Prolog proceed as follows:

First, the query matches the head of the sister rule, so X in the rule becomes instantiated to ann, and Y becomes instantiated to pat. The place marker for the query is put against this rule. Now Prolog attempts to satisfy the two goals in the body, one by one.

The first goal is parent(Z,ann) because X was instantiated to ann in the previous step. This goal is satisfied by the fact parent(bob,ann). So Prolog marks this fact in the knowledge base for the first goal and the variable Z is instantiated to bob. Prolog now attempts to satisfy the next goal.

Now Prolog searches for parent(bob,pat). A matching fact is parent(bob,pat), so the goal succeeds. Prolog marks this matching fact for the second goal. Prolog now attempts to satisfy the next goal.

Now Prolog searches for female(ann) because X is known as ann from the query. The goal succeeds, because a matching fact is found. Since it is the last goal in the conjunction, the entire goal succeeds, and the fact sister(ann,pat) is established as true. Prolog answers 'yes'.

1.3. Using a Prolog System

1.3.1 Input and Output - providing data for programs

In conventional programming the distinction between program and data is very clear with data for a program to manipulate being obtained as input using a read statement and output being conveyed using a write statement.

In Prolog, input and output can be processed in this way too (this will be dealt with in chapter 5) but, since knowledge can also include data, the distinction between program and data is more blurred. For example data can be provided using facts.

Querying also provides a mechanism for input and output through the use of values for variables. Input data can be given as a value of an argument and the output can be the instantiation of variables in Prolog's answer.

Thus in the query *Who does Peter like?*:

```
?- likes(peter,Who).
```

the input is 'peter' and the output is 'carol' (Who=carol).

In conventional programming the parameters in a procedure are each dedicated to

either input or output (but not both). Often in Prolog, however, arguments of a predicate in a query may swap roles, taking input on one occasion and providing output on another.

If we now ask *Who likes carol?*

```
?- likes(Person,carol).
```

then the second argument of likes/2 is taking input and the first is providing output in contrast to the previous question. This feature of Prolog procedures is called **bi-directionality**.

1.3.2 Built-in predicates

As well as the predicates you write yourself, Prolog has a large number of system predicates already defined. These are available for use all the time and do not have to be loaded. They are referred to as **built-in predicates (bips)**. Examples of these will be encountered from time to time. At the moment we need only bips to load knowledge.

You cannot use a procedure of your own with the same name and arity of a bip. If you try to load such a predicate Prolog will accuse you of 'trying to re-define a system predicate'.

1.3.3 Consulting and re-consulting knowledge base

Knowledge can be loaded from files into the Prolog system, i.e. into the system's internal knowledge base, using the bips consult/1 and reconsult/1. Here **consult** means *load for the first time* while **reconsult** means *reload* or *update* the system's knowledge. The argument to either must be an atom or list of atoms. For example:

```
?- consult(music).
```

will consult music.pl (the default extension .pl is assumed).

If the file has a different extension then this must be written explicitly and the file specification enclosed in quotes (to make it an atom), e.g.

```
?- consult('relative2.txt').
```

A list of files may be given as the argument. You list the files enclosed in square brackets, e.g. `[music,music2,music3]`.

Thus a typical command might be:

```
?- consult([music,music2,music3]).
```

or just:

```
?- [music,music2,music3].
```

Use of reconsult/1 is similar to consult/1 except that the knowledge base is updated rather than added to (as it is with consult/1); it should be used on subsequent occasions after the file has been edited. Note that 'updating' means that *each edited procedure is completely overwritten with the new version*. Some Prolog systems will only allow a file to be consulted once with subsequent consults of the same file producing an error. If a second consult is allowed it will create duplicate clauses in the system. The consult/1 and reconsult/1 predicates can be 'anded' together:

```
?- consult(music),reconsult(music2).
```

Note that reconsult/1 can be denoted by a minus sign in front of the file in a list:

```
?- [music,-music2].
```

1.3.4 Debugging Prolog - the Box Model of Searching

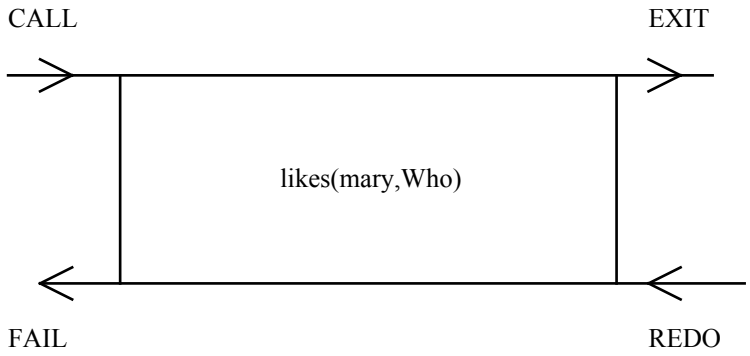
In the **box model** of the search procedure the current goal is regarded as being in a box with four **ports** which represent the possible means by which it came to be the current goal and will cease to be the current goal. These ports are:

CALL - the goal is 'called', i.e. searched for the first time.

- **EXIT** - a match for the goal is obtained, possibly with instantiation and this goal ceases to be the current one - Prolog moves on to the next goal.
- **FAIL** - the goal fails, i.e. a match was not found.

REDO - the goal is recalled, i.e. the goal has been the subject of a CALL before, but is now searched again for an alternative solution.

In the box model diagram of this process, success is represented by movement across the box from left to right (as in Prolog's searching) - hence the direction of arrows:



Suppose music.pl is loaded and Prolog is asked the question:

```
?- likes(mary,Who),likes_artist(Who,'The Cranberries').
```

The box model will yield the trace:

```
        1       (0)     CALL: likes(mary,_Who)

        1       (0)     EXIT:  likes(mary,john)

        2       (0)     CALL: likes_artist(john,The Cranberries)

        2       (0)     FAIL:  likes_artist(john,The Cranberries)

        1       (0)     REDO:likes(mary,john)

        1       (0)     EXIT:  likes(mary,carol)

        3       (0)     CALL: likes_artist(carol,The Cranberries)

        3       (0)     EXIT:  likes_artist(carol,The Cranberries)
Who = carol
```

```
More (y/n)? y
        3       (0)     REDO:likes_artist(carol,The Cranberries)
        3       (0)     FAIL:  likes_artist(carol,The Cranberries)
        1       (0)     REDO:likes(mary,carol)
        1       (0)     EXIT:  likes(mary,stephen)
        4       (0)     CALL:  likes_artist(stephen,The Cranberries)
        4       (0)     FAIL:  likes_artist(stephen,The Cranberries)
        1       (0)     REDO:likes(mary,stephen)
        1       (0)     FAIL:  likes(mary,_678)
                no
```

1.4 Declarative and procedural aspects of Prolog

1.4.1 Logical (Declarative) Meaning in Prolog

Prolog is an example of a **declarative** programming language. The emphasis is on declaring knowledge, i.e what is true, and leaving Prolog to figure out how to carry out the tasks it is asked to perform. In contrast the programmer in a conventional imperative language has always to be concerned with the *how*.

1.4.1.1 Truth in Prolog

The contents of the knowledge base (i.e. the knowledge that is loaded) constitute all that Prolog knows and form the basis of its answers to questions asked at the Prolog prompt. To Prolog, 'true' means 'is in the knowledge base' or can be inferred from what is in the knowledge base by reasoning. If something is not in the knowledge base (or cannot be inferred) then it is regarded as false. This is the **closed world assumption.** This is in marked contrast to human reasoning where if we do not know that something is true we just accept our ignorance - we do not assume it must be false.

1.4.1.2 The logical structure of the knowledge base

A knowledge base consists of individual clauses containing predicates joined by the logical connectives **and** (called **conjunction**) **or** (called **disjunction**), and **not** (called negation).

In logic, a statement made by joining several statements with 'and', is true if each of the individual statements is true; if a statement consists of several statements joined by 'or' it is true if at least one of them is true.

A negated statement is true if the statement itself is false; it is false if the latter is true.

Since each of the clauses in the knowledge base is true they are in effect 'anded' together and, from a logical point of view, form one large clause. The order in which the clauses are written therefore does not affect the logical meaning of the knowledge base because re-ordering 'anded' statements does not alter the overall truth content.

Looking at a knowledge base as a expression of the knowledge we wish to convey to Prolog is called the **declarative** view of the knowledge base - it is knowledge from a logical point of view.

1.4.1.3 Precedence of 'and' and 'or'

When dealing with a mixture of 'and' and 'or' in statements it is important to be clear about **precedence**. The precedence of 'or' is higher than 'and' thus to Prolog a,b;c means (a,b);c. It is as if (a,b) on the one hand and c on the other are two separate cases. This is the interpretation Prolog will make if you do not use brackets. Another way to think of this is that lower precedence operations join or **bind** first, i.e. a is 'anded' with b and the result is 'ored' with c.

In effect the statement is a structure which can be illustrated in a tree as shown in figure 4 below:

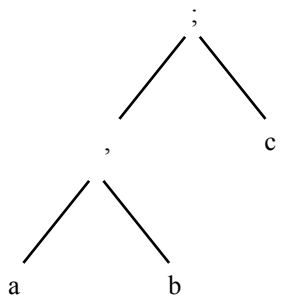


Figure 4: Tree diagram of the precedence of the structure a,b;c

If you wish to 'or' b with c and have the result 'anded' with a you must explicitly write a,(b;c) using brackets to make the intention clear. This precedence must be observed wherever use of 'and' and 'or' is permitted, i.e. in conditions of rules and in questions.

Thus with music.pl to ask : *Do Mary or Danny like anyone who likes R.E.M.?* we write:

```
?- (likes(mary,X);likes(danny,X)),likes_artist(X,'R.E.M.').
```

In logic, *and* and *or* behave very much like * and + respectively in arithmetic. Just as:

3*(4+5) = 3*4 + 3*5

so

a,(b;c) is the same as a,b ; a,c

similarly

a;(b,c) is the same as (a;b),(a;c).

Very complex statements may be constructed by using 'and' and 'or' several times, e.g. (a,b;c),d;e,f

Rules with a top-level 'or' in their condition (i.e. an 'or' in the root of the tree) may, for convenience and simpler reading, be split into several rules. For example:

```
w:- (a,b;c),d;e,f
```

may be written as:

```
w:- (a,b;c),d.
```

```
w:- e,f.
```

The new rules all have the same conclusion. Each component of a top-level 'or' becomes a condition.

1.4.2 Procedural View of Prolog

Ideally, in order to program effectively in Prolog it should be sufficient to load a knowledge base and give a query which would then be reasoned with. In practice, however, for reasons which will become clear, it is important to have an appreciation of how Prolog answers queries. Looking at a Prolog program in terms of how Prolog will use it is called the *procedural view*.

1.4.2.1 Goals and sub-goals

Something Prolog is searching for in the knowledge base is called a **goal**. If this is found we say the goal has **succeeded** or is **satisfied**; if it is not found we say the goal has **failed**. The goals Prolog has to deal with arise initially from queries, but also from conditions in rules, since these state things which Prolog has to establish.

In trying to satisfy a goal, Prolog breaks the task down into a sequence of **sub-goals**. The individual predicates in a question or condition are sub-goals. Typically a goal consists of several sub-goals joined using 'and', 'or' and 'not'. Prolog understands the meaning of these logical connectives, e.g. that to prove a goal consisting of 'anded' sub-goals, both sub-goals must be established.

1.4.2.2 Searching and matching

Since Prolog tries to establish goals by looking through the knowledge base, it must have strategies for searching systematically and recognising what it is looking for when it is found. These are called the **search** and **match** processes respectively. Searching and matching are fundamental computing activities in any KBS or AI language.

At any one time Prolog is trying to satisfy a single sub-goal involving one structure (referred to as a term). This term may have arisen in the query itself or in the condition of a rule (as will be seen below). Prolog searches for the term by beginning *at the top of the knowledge base* and looking at each fact or conclusion of a rule (i.e. head of a clause) to see if it matches the term. Predicates match if they have:

1. the same principal functor and
2. the same arity and
3. have the same arguments.

Either (or both) the goal or the matching head of a clause may contain one or more variables. In this case the variable(s) are set to have the value in the appropriate argument. For example, if the query is:

```
?- owns(Who,car(Manufacturer,Model,blue,Year).
```

and Prolog finds a fact:

```
owns(tim,car(ford,escort,blue,1987))
```

in the knowledge base, then these match with Who=tim, Manufacturer=ford,

Model=escort and Year=1987. We say that the variable Who is **instantiated** or **bound** to tim and likewise the other variables. Thus bringing about a match may involve instantiation of variables.

In this example because the clause that Prolog found is a fact, the answer :

```
Who=tim,

Manufacturer=ford,

Model=escort

Year=1987.
```

is reported immediately to the user. Had the term that Prolog found been the head of a rule then this would not be 'true' until the condition of the rule had been established.

When Prolog matches with the head of a rule it *immediately starts to establish the condition* - thus the condition becomes a new sub-goal. In trying to satisfy this new sub-goal, Prolog may find another head of a rule and this will start a new sub-goal and so on.

When a variable is instantiated in a clause to bring about matching, *that instantiation is made for all other occurrences of the variable in the same clause (either in the knowledge base or the query)*. For the time being, then, Prolog has a value for that variable and this value is used in new sub-goals, e.g. the condition of a rule.

To illustrate this, suppose music.pl is loaded and the query:

```
?- likes_artist(Who,'Tanita Tikaram')
```

is made. The first matching clause Prolog will find is:

```
likes_artist(brenda,Artist):-
  music_type(Artist,blues);music_type(Artist,folk).
```

The instantiations Who=brenda, Artist='Tanita Tikaram' are made; also this instantiation for Artist is also passed along to the condition which becomes:

```
music_type('Tanita Tikaram',blues);music_type('Tanita Tikaram',folk)
```

This is the new sub-goal for Prolog. If true (which it is) then the conclusion likes_artist(brenda,'Tanita Tikaram') has been established and so the original goal (the query itself) has been proven with Who=brenda.

Sometimes matching requires a variable to be matched to another variable, for example a variable called What in a query might match with a variable X in the head of rule. In this situation the variables are said to **share -** meaning they share values. When one of them is instantiated, the other will get the same instantiation.

For example consider the query:

```
?- gotoconcert(Who,carol,Artist).
```

where a matching rule in the knowledge base is:

```
gotoconcert(X,Y,Z):-
   likes(X,Y),
   likes(Y,X),
```

```
likes_artist(X,Z),
likes_artist(Y,Z).
```

Prolog will find the head of this clause and match it to the query with:

```
Who=X    (Who shares with X),  Y=carol, Artist=Z (Artist shares with Z).
```

Prolog will now attempt to establish the condition of the rule with Y=carol. If this is successful, instantiations will be obtained for X and Z (say X=mary, Z='The Cranberries') and these instantiations will be shared with Who and Artist respectively thus giving Who=mary, Artist='The Cranberries' which would be reported to the user.

1.4.2.3 The order in which sub-goals are searched

When tackling a question or condition of a rule, Prolog works from left to right across the various sub-goals with due regard for the logical meaning, i.e. to prove 'anded' sub-goals both must be established, whereas to prove 'ored' sub-goals it is sufficient to establish one of them. At any one time Prolog can search for a single predicate only using matching as described above. For example the goal:

```
?- a,b;c,d
```

can be broken down into two 'ored' sub-goals a,b and c,d each of which is further decomposable into two 'anded' sub-goals.

Prolog starts by trying to prove a; if this sub-goal succeeds it will try to prove b, whereas if it fails there is no need to try b since the sub-goal a,b cannot now succeed. If b does succeed then the question has been satisfactorily answered 'yes' and this will be reported to the user - there is no need to establish the sub-goal c,d. If b fails (or indeed if a fails) then Prolog will try c,d. The same routine is used to establish the condition of a rule.

1.4.2.4 Depth-first searching

In trying to satisfy a sub-goal Prolog may have to consider other rules. For the query above there may be a rule in the knowledge base for a:

```
a:- (g;h),i.
```

Thus having initially started to search for a and found this rule, Prolog has now to try to establish the condition (g;h),i. The first sub-goal it has here is g; there may be a rule for g and so on… .An initial goal a has given way to a new sub-goal g. This search strategy is called **depth first searching** because *once a goal is started it is pursued to its depths before another sub-goal at the same level is tried*.

In this example the goal b is said to be at the same level as a (they are at the same level in a tree diagram of the structure). On the other hand, the goal g is at a lower level than a because a is defined in terms of g. A lower level goal is called a **child** goal; a higher level goal ia called an **ancestor** or **parent** goal.

Thus having started goal a, Prolog will pursue all child goals arising from it in an attempt to satisfy a. If it eventually proves a, only then will b be started; if a eventually fails then c will be tried.

Depth-first is not the only search strategy. Another simple strategy is **breadth-first**. This says: *if you cannot immediately establish a goal (i.e. it is not in the knowledge*

base as a fact) then give up temporarily and try another goal at the same level and so on... . Only when everything has been tried at the first level without overall success do you go back and pursue goals at the next lower level.

1.4.3 Backtracking

When a goal is satisfied, one or more variables may be instantiated. These instantiations are then passed to the other occurrences of those variables in the clause. When new sub-goals in which these variables appear are started, it is these instantiations which will be used.

Prolog, however, may not be able to satisfy a new sub-goal using these instantiations. Logically, this does not mean there is no possible solution - *only that the instantiations obtained so far do not lead to an answer*. In this situation, Prolog must **backtrack** to try and find alternative instantiations.

A place in the knowledge base where Prolog finds a match is called a **choice point**. When a sub-goal fails, Prolog returns to the most recent relevant choice point (i.e. one involving instantiations used in the sub-goal) and releases all the instantiations made at that point. It then continues on down the database looking for an alternative match.

It is thus re-doing a previous goal. If a new match is found, new instantiations are made and these are passed along to the sub-goal that failed and it is tried again (we say that Prolog is moving **forward** again). If the bottom of the knowledge base is reached without a match being found then Prolog must backtrack still further and redo an older goal and so on.

Backtracking as described above applies to conditions in rules as well as questions. Prolog searches a condition because it has matched with the conclusion of that rule; this corresponds to a choice point, so if the condition fails, Prolog will return to the choice point and seek a new match. If a question is answered 'true' with instantiations given to the user, Prolog will offer to find other answers (using the y/n prompt). If other answers are requested then it will backtrack through the question from the last sub-goal looking for an alternative solution. Thus backtracking is started automatically by Prolog when necessary during the course of a search, or may be started by the user as a means of finding out if there are other possible answers.

1.4.3.1 Examples of backtracking

Consider the following question involving music.pl:

```
?-likes(michael,Who),likes_artist(Who,'Eric Clapton').
```

The search is as follows:

Prolog will search first for likes(michael,Who).

1. A match will be obtained with Who=carol.
2. The next goal is likes_artist(carol,'Eric Clapton'). Prolog searches from the top of the knowledge base for this goal but cannot find it, i.e. the goal fails.
3. It returns to the choice point where Who was instantiated and frees the instantiation Who=carol. It continues from this point down the knowledge base attempting to find a new match for likes(michael,Who); this succeeds in the next clause with Who=brenda.

4. Prolog moves forward to the likes_artist predicate again, except that now the goal is likes_artist(brenda,'Eric Clapton'). This goal will eventually succeed as Brenda likes blues or folk music and Eric Clapton is a blues artist.
5. The user will be given the answer Who=brenda.

If the user requests an alternative answer:

1. Prolog will backtrack to the choice point where Who=brenda was obtained and search further for the goal likes(michael,Who) finding Who=danny.
2. It will now move forward to the goal likes_artist(danny,'Eric Clapton') which succeeds.
3. The user will be given the answer Who=danny.

Further backtracking will take place but there are no more solutions to likes(michael,Who) and so Prolog will report that there are no more answers.

When Prolog returns to a choice point, *all* values instantiated at that point must be released. Consider the question:

```
?- likes(A,B),likes_artist(B,'Cliff Richard').
```

The first goal will succeed with A=mary, B=John.

Prolog will move to the next goal which is likes_artist(john,'Cliff Richard'). This will fail causing backtracking to the most recent choice point. Here, *both* the instantiations A=mary,B=john will be freed and Prolog will search down the knowledge base for an alternative finding A=mary,B=carol.

This will create a new goal likes_artist(carol,'Cliff Richard') which succeeds.

1.4.3.2 Non-deterministic programming

Prolog's facility to backtrack endows it with a **non-deterministic** property - the program does not determine a particular single answer. In general, through backtracking, a number of possible solutions are possible and the Prolog is capable of finding them all if requested to do so.

Chapter 2 Prolog Syntax and Notation

Prolog programs are built from **terms**. A *term* is either a *constant*, *variable*, or a *structure*. Each term is written as a string of *characters*.

The following characters are allowed in Prolog:

upper-case letters	A, B, ..., Z
lower-case letters	a, b, ..., z
digits	0, 1, 2, ..., 9
special characters such as	+ - * / < > = : . & _ ~

2.1 Constants

Constants are used to name special objects or relations. There are two types of constants: **atoms** and **numbers**.

For example, the names of the following objects and relations are atoms:

pam, ann, liz, bob, jim, parent, female, sister, predecessor

Atoms can be built in three ways:

1. Strings of letters, digits and the underscore character _ starting with a lower-case letter.

ann, parent_relation, bill_clinton, isun20

2. Strings of special characters.

<--->, ===>, :-, ?-

Among the above atoms, :- and ?- are reserved ones having predefined meanings.

3. Strings of characters enclosed in single quotas.

'John', 'Co. Antrim', 'Logic Programming'

If we want to use an atom starting with a capital letter, then the atom has to be built in this quoted form.

Numbers consist of integer numbers and real numbers. The range and treatment of numbers are dependent on Prolog implementations. The range of integer numbers is at least between -16383 and 16383. We will assume that the simple syntax of real numbers is allowed as shown below:

3.14, -0.0064, 10.8

The use of real numbers in Prolog is limited because Prolog is primarily a language for symbolic and non-numeric computation.

2.2 Variables

Variables can be used to stand for some objects that we are not able to name. Variables are built of strings of letters, digits and underscore character starting with an upper-case letter.

For example, the following strings of characters are variables:

```
X, Y, Z, X25, TaxCode, _, Student_Number
```

When we want a variable to stand for some object but we are not interested in the name of the object, we can use the so-called `anonymous' variable, which is written as a single underscore character. Each time a single underscore character occurs in a clause it represents a new anonymous variable.

The lexical scope of variables is one clause. If the same strings of characters occur in two clauses as a variable, then they are two different variables. We should, however, remember that the constant always means the same object in any clause throughout the whole program.

2.3 Structures

Structures in Prolog are used to represent structured objects that have several components. The components in the structures can also be structures. A **functor** is used to combine the components into a single object. For example, the following objects are structures.

```
car(mercedes,3.5,Colour)

class(jun,prolog,students(1st,bsc),period(96,semester1))

date(30,sep,96)
```

Components can be constants. They can be also be variables or other structures. For example, in `car(mercedes,3.5,Colour)`, the variable `Colour` represents any colour. In the second structure above, the component `students` and `period` themselves are structures.

2.4 Equality and Matching

Equality has its special meaning in Prolog, which is given as an important built-in operator written as ``=". This operator actually tells Prolog to match `X` and `Y` instead of deciding whether they are exactly the same when a goal of the form `X=Y` is given, where `X` and `Y` are any two terms which may contain variables. The rules for deciding whether `X` and `Y` are equal are as follows:

If X is an uninstantiated variable, and if Y is instantiated to any term, then X and Y are equal. As a consequence, X becomes instantiated to whatever Y is. For example, when the following goal succeeds with X instantiated to likes(mary,wine):

```
X = likes(mary,wine)
```

Constants (integers and atoms) are always equal to themselves. For example, the execution of the following goals is shown below:

```
pen = pencil   fails
```

```
345 = 345     succeeds

345 = 543     fails

wine = wine   succeeds
```

Two structure are equal if they have the same functor and number of components, and all the corresponding components are equal. For example, the following goal succeeds with `X` instantiated to `wine`:

```
likes(mary,wine) = likes(mary,X)
```

If `X` and `Y` are both uninstantiated, the goal `X = Y` succeeds and the two variables share so that whenever one of them becomes instantiated to some term, the other one is automatically instantiated to the same term. For example, in the following rule, one argument will always be instantiated to the other whatever it is:

```
equal(X,Y) :- X = Y.
```

2.5 Arithmetic and Comparative Operators

It is convenient to represent arithmetic and comparative expressions in Prolog. Three things about an operator are important: **position**, **precedence**, and **associativity**.

According their positions, arithmetic operators are classified into three types:

infix operators: `+ - * / (real division) div mod`

prefix operator: `-`

postfix operator: `!`

Operators like **+, -,** *, and / are called **infix** operators. The operator -, when used in arithmetic expressions to denote negation, is called **prefix** operator. The operator **!** is called **postfix** operator.

The precedence of an operator indicates which operation is carried out first. Each arithmetic operator in Prolog has a **precedence class** associated with it. In Prolog, the multiplication and division operators are in a higher precedence class than addition and subtraction.

An operator can be either **left associative** or **right associative**. In Prolog, all the arithmetic operations are left associative. Appropriate round brackets can be used in complex arithmetic expressions to avoid confusion caused by the precedence and associativity rules.

An arithmetic calculation is carried out in Prolog using the `is' operator. This is undertaken as either a goal in a query or a subgoal in a rule.

```
?-X is 3*4.
X = 12

sum(X,Y,Z) :-
 Z is X + Y.
```

The left hand side of `is' is usually a variable. The right hand side is an arithmetical expression and can involve the usual mathematical operators, e.g. `sqrt, log, sin`:

```
?- Ans is sqrt(9/4)*log15.2.
```

There are several ``built-in'' predicates in Prolog, which can be used to make comparisons between two arithmetic expressions. These predicates are written as infix operators as follows:

X =:= Y	the values of X and Y are equal
X =\= Y	the value of X and Y are not equal
X < Y	X is less than Y
X > Y	X is greater than Y
X =< Y	X is less than or equal to Y
X >= Y	X is greater than or equal to Y

A comparison can be undertaken as either a goal in a query or a subgoal in a rule.

2.6 Abstracting Data from Structures

Consider a knowledge base containing structures in the form of:

```
person(FirstName,Date_of_Birth,Employment).
```

and that Date_of_Birth has the structure

```
dob(Day,Month,Year)
```

while Employment has the structure:

```
works(Company, Salary) or

unemployed
```

An example person would be:

```
person(joe,dob(27,apr,1953),works(uuj,15000)).
```

We can then make different queries to `abstract` certain piece of data from the knowledge base.

Who works in uuj?

```
?- person(Who,_,works(uuj,_)).
```

Who is unemployed?

```
?- person(Who,_,Works), Works = unemployed.
```

Who was born before 1960?

```
?- person(Who,dob(Year,_,_),_), Year < 1960.
```

How old is Joe?

```
?- person(joe,dob(Year,_,_),_), Age is 1996 - Year.
```

Who works in the same organisation as mary?

```
?- person(mary,_,work(O1,_)), person(Who,_,work(O2,_)), O1 = O2.
```

or

```
?- person(mary,_,work(Org,_)),person(Who,_,work(Org,_)).
```

2.7 Lists

The list is an *ordered* sequence of elements. The **elements** of a list can be any terms -- constants, variables, structures. The list can have any length. So although lists are very simple data structures, they are extremely flexible to use and therefore very powerful. All the elements in a list are enclosed in a pair of brackets. An empty list is written as **[]**. For example, the following are lists:

```
[a,b,c]

[ann,tom,bob,[like,fish]]

[a,b,[a,b],[a,b,[a,b]]]

[hello,likes(john,mary)]

[]
```

A list can be divided into two parts: the first element in the list is called the *head* and the rest of the elements in the list is called the *tail*. The head of a list is a single element and the tail of a list itself is a list. The head and tail of a list can be separated by a vertical bar **|**. For example, **[a|[b,c]]** is the same as **[a,b,c]**. So if we use two variables **Head** and **Tail** to represent the head and tail of a list respectively, when **[Head|Tail]** is made to match the list **[a,b,c]**, the **Head** is instantiated to **a** and the **Tail** is instantiated to **[b,c]**. An empty list has no head and tail.

The list is a special kind of structure. It has a special functor **.** (full stop) and the head and tail of a list as its components. So the list **[a,b,c]** can be written in the form of structure as follows:

```
.(a,.(b,.(c,[])))
```

In Prolog we can use these two forms of representation. But it is often easier to represent a list like **[a,b,c]**. *But we should remember that the last element of a list is an empty list* **[]**.

Chapter 3 Recursion

Now let us define another family relation predecessor. This relation will be defined in terms of the parent relation. The whole definition can be expressed in two rules. The first rule will define the direct (immediate) predecessors and the second rule the indirect predecessors. We say that some X is an indirect predecessor of some Z if there is a parentship chain of people between X and Z.

The first rule is simple and can be formulated as:

> For all X and Z,
>
> > X is a predecessor of Z if
> >
> > X is a parent of Z.

This is straightforwardly translated into Prolog as:

```
predecessor(X,Z) :-
  parent(X,Z).
```

The second rule, on the other hand, is more complicated because the chain of parents may present some problems. The predecessor relation would be defined by a set of clauses as follows:

```
predecessor(X,Z) :-
  parent(X,Z).

predecessor(X,Z) :-
  parent(X,Y),
  parent(Y,Z).

predecessor(X,Z) :-
  parent(X,Y1),
  parent(Y1,Y2),
  parent(Y2,Z).

predecessor(X,Z) :-
  parent(X,Y1),
  parent(Y1,Y2),
  parent(Y2,Y3),
  parent(Y3,Z).

...
```

This program is lengthy and, more importantly, it only works to some extent. It would only discover predecessors to a certain depth in a family tree because the length of the chain of people between the predecessor and the successor would be limited according to the length of our predecessor clauses.

There is, however, an elegant and correct formulation of the predecessor relation: it will be correct in the sense that it will work for predecessors at any depth. The key idea is to define the predecessor relation in terms of itself.

For all X and Z,

X is a predecessor of Z if

there is a Y such that

(1) X is a parent of Y and

(2) Y is a predecessor of Z.

A Prolog clause with the above meaning is:

```
predecessor(X,Z) :-
  parent(X,Y),
  predecessor(Y,Z).
```

Putting the two rules of the `predecessor` relation together, we have

```
predecessor(X,Z) :-
  parent(X,Z).

predecessor(X,Z) :-
  parent(X,Y),
  predecessor(Y,Z).
```

The first rule is called the stopping case because it stops the recursion eventually when there is a parent relation between `X` and `Z`. In a stopping case there is no further use of the defined relation itself.

The second rule is called the recursive rule. The key to the formulation of this rule is the use of the defined relation itself in the rule. Recursive programming is one of the fundamental principles of programming in Prolog. It is not possible to solve tasks of any significant complexity in Prolog without the use of recursion.

The final form of the program for the family relations is as follows:

```
parent(pam,bob).

parent(tom,bob).

parent(tom,liz).

parent(bob,ann).

parent(bob,pat).

parent(pat,jim).

female(pam).

male(tom).

male(bob).

female(liz).

female(ann).

female(pat).
```

```
male(jim).

offspring(Y,X) :-
  parent(X,Y).

mother(X,Y) :-
  parent(X,Y),
  female(X).

grandparent(X,Z) :-
  parent(X,Y),
  parent(Y,Z).

sister(X,Y) :-
  parent(Z,X),
  parent(Z,Y),
  female(X),
  different(X,Y).

predecessor(X,Z) :-
  parent(X,Z).

predecessor(X,Z) :-
  parent(X,Y),
  predecessor(Y,Z).
```

The following are some sample recursive programs.

1. Program to sum the integers from 1 to N.

```
sum(1,1).

sum(N,Ans) :-
  N1 is N-1,
  sum(N1,Ans1),
  Ans is N + Ans1.
```

2. Program to calculate N!.

```
factorial(0,1).

  factorial(N,Ans) :-
  N1 is N - 1,
  factorial(N1,Ans1),
  Ans is N * Ans1.
```

3. Program to calculate X to the power N.

```
power(X,0,1).

power(X,N,Ans) :-
  N1 is N -1,
  power(X, N1, Ans1),
  Ans is A * Ans1.
```

Chapter 4 List Operations

In the following sections, we will first discuss how to do some operations on lists using built-in predicates (bip), namely, member/2, member/3, length/2, append/3, remove/3, reverse/2, sort/2, findall/3. We will then discuss how to define recursive rules and how they work in Prolog.

4.1 Membership

member/2 is used to define the membership relation as

```
member(X,L)
```

where X is a term and L is a list. The goal member(X,L) succeeds if X is an element of L.

There are two main uses of this relation:

1. To provide each member of the list successively.

```
?- member(X, [a,b,c]).

    X = a; % Prolog backtracks through the list.

    X = b;

    X = c;

    no
```

2. To determine if a given element is in the list.

```
    ?- member(b, [a,b,c]).

    yes

    ?- member(d, [a,b,c]).

    no
```

Assume a knowledge base paint.pl is loaded as follows:

```
/* Colours and pricing for a range of paints */

/* colours/2 gives the names of paints available in a particular colour. Some of
these paints are also classified as off-white. All paints are available in vinyl
matt or silk; some are also available in gloss. */

colours(red,[crimson,pillarbox,sunset,rosepetal]).

colours(blue,[sky,lilac,ice,ocean,cornflower]).

colours(green,[apple,seasplash,holly,leaf]).
colours(yellow,[buttercup,primrose,cream,sunlight,lemon]).
```

```
colours(brown,[biscuit,oak,magnolia]).

colours(offwhite,[ice,rosepetal,apple,cream,magnolia,sunlight]).

/*  gloss/1 gives a list of the paints available in gloss */

gloss([pillarbox,lilac,holly,buttercup,oak,ice,crimson,cornflower,
      cream]).
```

```
/* price/2 has a list of Size/Price in the second argument for a finish specified in
the first argument.  */

price(vinyl(matt),[2.5/8.99,5/15.99,10/28.50]).

price(vinyl(silk),[2.5/9.55,5/16.49,10/29.50]).

price(gloss,[1/5.65,2.5/10.49,5/18.00]).
```

Now we can make queries to the knowledge base.

What are the names of the blue paints?

```
?- colours(blue,L), member(Paint,L).

L = [sky,lilac,ice,ocean,cornflower],

Paint = sky;

L = [sky,lilac,ice,ocean,cornflower],

Paint = lilac;

L = [sky,lilac,ice,ocean,cornflower],

Paint = ice;

etc.
```

What blue paints are available in gloss?

```
?- colours(blue,L1), gloss(L2), member(Paint,L1), member(Paint,L2).

L1 = [sky,lilac,ice,ocean,cornflower],

Paint = Lilac,

L2 = [pillarbox,lilac,holly,buttercup,oak,ice,crimson,cornflower,cream];

etc.
```

What colour is apple?

```
?- colours(Colour,L), member(apple,L).

Colour = green,

L = [apple,seasplash,holly,leaf];

Colour = offwhite,

L = [ice,rosepetal,apple,cream,magnolia,sunlight];

no
```

How much in total is a 5 litre tin of vinyl silk and a 1 litre tin of gloss?

```
?- price(vinyl(silk),Slist), member(5/P1,Slist), price(gloss,Glist),
member(1/P2,Glist), P is P1 + P2.
```

```
Slist = [2.5/9.55,5/16.49,10/29.5],

P1 = 16.49,

Glist = [1/5.65,2.5/10.49,5/18],

P2 = 5.65,

P = 22.14
```

Which yellow paints are not off-white?

```
?- colours(yellow,YL), colours(offwhite,OWL), member(Paint,YL), \+
member(Paint,OWL).

YL = [buttercup,primrose,cream,sunlight,lemon],

OWL = [ice,rosepetal,apple,cream,magnolia,sunlight],

Paint = buttercup;

YL = [buttercup,primrose,cream,sunlight,lemon],

OWL = [ice,rosepetal,apple,cream,magnolia,sunlight],

Paint = primrose;

etc.
```

member/3 behaves much like member/2 except the extra argument is the position of the element in the list:

What is the third paint in the yellow list?

```
?- colours(yellow,L), member(Paint,L,3).

L = [buttercup,primrose,cream,sunlight,lemon],

Paint = cream
```

What is the position of oak in the brown list?

```
?- colours(brown,L), member(oak,L,Position).

L = [biscuit,oak,magnolia],

Position = 2
```

4.2 Matching Using Head and Tail

What is the first blue paint?

```
?- colours(blue,[First|Rest]).

First = sky,

Rest = [lilac,ice,ocean,cornflower]

?- colours(blue,[First|_]).

First = sky
```

Head and tail can be used in a matching goal:

```
?- [biscuit,oak,magnolia] = [H|Tail].

H = biscuit,

Tail = [oak,magnolia]
```

A single element list has the empty list as its tail:

```
?- [a] = [H|Tail].

H = a, Tail = []
```

The tail separator can be placed after several elements:

```
?- [biscuit,oak,magnolia] = [First, Second|Tail].

First = biscuit,

Second = oak,

Tail = [magnolia]
```

4.3 Other Built-In Predicates for List Operations

There are built-in predicates (bips) to do the following:

Find the length of a list.

Append a second list to the end of a first list.

Remove an element from a list.

Reverse the elements in a list.

Sort the elements in a list.

4.3.1 length/2

`length(L,N)` returns the length, i.e. the number of elements, in the list `L`.

For what colours of paint are 5 or more shades offered?

```
?- colours(Col,L), length(L,N), N >= 5.

Col = blue,
```

```
L = [sky,lilac,ice,ocean,cornflower],

N = 5;

Col = yellow,

L = [buttercup,primrose,cream,sunlight,lemon],

N = 5;

Col = offwhite,

L = [ice,rosepetal,apple,cream,magnolia,sunlight],

N = 6
```

4.3.2 append/3

Two lists may be joined with:

```
append(L1, L2, JoinedList).
```

Here L2 is joined on to the end of L1 to make JoinedList:

```
?- append([a,d,e], [h,k,m], NewList).

NewList =[a,d,e,h,k,m].
```

To join several lists requires repeated use of append:

Join lists [sun,mon], [tues,weds,thurs] and [fri,sat]

```
?- append([sun,mon], [tues,weds,thurs], L1), append(L1, [fri,sat], L).

L1 = [sun,mon,tues,weds,thurs],

L = [sun,mon,tues,weds,thurs,fri,sat]
```

append/3 can be used to split a given list into two lists:

```
?- append(L1, L2, [sun,mon,tues,weds,thurs,fri,sat]).

L1 = [],

L2 = [sun,mon,tues,weds,thurs,fri,sat];

L1 = [sun],

L2 = [mon,tues,weds,thurs,fri,sat];

L1 = [sun,mon],

L2 = [tues,weds,thurs,fri,sat];

etc.
```

We can add further conditions:

Split [sun,mon,tues,weds,thurs,fri,sat] into two lists where the first has 4 elements:

```
?- append(L1,L2,[sun,mon,tues,weds,thurs,fri,sat]), length(L1,4).

L1 = [sun,mon,tues,weds],

L2 = [thurs,fri,sat]
```

In [sun,mon,tues,weds,thurs,fri,sat], find the elements after tues:

```
?- append(_, [tues|T], [sun,mon,tues,weds,thurs,fri,sat]).

T = [weds,thurs,fri,sat]
```

4.3.3 remove/3

remove(X, L, RemainderList) removes X from L (if it is there) to leave RemainderList, e.g.

```
?- remove(book,[comic,paper,book,magazine], RemainderList).

RemainderList = [comic,paper,magazine].
```

If the element to be removed is not in the list, the goal fails. Deleting more than one element from a list requires repeated use of remove/3. If a variable is used in the first argument then the first element in L is deleted and with backtracking this becomes the second, third, etc. - but not cumulatively:

```
?- remove(X, [have,a,nice, day], L).

X = have,

L = [a,nice,day];

X = a,

L = [have,nice,day];

X = nice,

L = [have,a,day];

X = day,

L = [have,a,nice]
```

remove/3 can be used to add an element to a list:

```
?- remove(nice, L, [have,a,day]).

L = [nice,have,a,day];

L = [have,nice,a,day];

L = [have,a,nice,day];

L = [have,a,day,nice]
```

Find a list of green colours other than seasplash?

```
?- colours(green, GList), remove(seasplash, GList, Rest_of_Green).

GList = [apple,seasplash,holly,leaf],

Rest_of_Green = [apple,holly,leaf]
```

4.3.4 reverse/2

reverse(L, RevL) returns RevL as the list in which the order of elements in L is reversed.

```
?- reverse([apple,seasplash,holly,leaf], RevL).

RevL = [leaf,holly,seasplash,apple]
```

It is useful for finding the last element in a list as follows:

Reverse the list then pick out the first element of the reversed list.

Find the last paint in the green list.

```
?- colours(green, GList), reverse(GList, [Last|_]).

GList = [apple,seasplash,holly,leaf],

Last = leaf
```

4.3.5 sort/2

sort(List, SortedList) returns SortedList as the sorted elements of List and removes duplicates.

There is a standard order of all Prolog terms

for atoms it is just alphabetical,

for numbers it is just increasing.

```
?- sort([apple,seasplash,holly,leaf], S).

S = [apple,holly,leaf,seasplash]

?- sort([3.45, 10, 4.75, 10, 1], S).  \% 10 is repeated

S = [1,3.45,4.75,10]
```

4.3.6 Collecting Answers to a Query

A query can have several answers:

```
?- likes(michael, Who).

Who = carol;

Who = brenda;

Who = danny
```

We can get all the answers one at a time using backtracking. As each answer is

obtained, the previous ones are lost. But for some queries we need all the answers together, e.g.

- Does Michael like more people than Mary?

- How many types of music are there?

- Make a list of people who like each other.

- Produce an alphabetically-ordered list of people who like either a folk or rock artist.

A built-in predicate for collecting answers is `findall/3` which will run a query and collect all answers in a list. The format is:

```
findall(Ans, Goal, AnsList)
```

where

- `Ans` is the required format of an individual answer,
- `Goal` is the query to be run,
- `AnsList` is the list of answers to `Goal`.

The 1st argument is usually the variable(s) whose answers you want collected:

Find a list of all people who like Michael.

```
?- findall(Who, likes(michael,Who), L).

Who = _,

L = [carol,brenda,danny]
```

Note that no value is returned for `Who` - it is just a dummy variable to indicate what you want to collect.

The answers occur in the list in the same order as they would if the query was run in the normal way.

If the query has no answers, the empty list is returned.

There may be other variables in the query which you do not want to collect:

Find a list of all people who are liked by somebody.

```
?- findall(Who, likes(X,Who), L).

Who = _ ,

X = _,

L = [john,carol,stephen,michael,carol,mary,stephen,

     stephen,danny,carol,mary,carol,brenda,danny]
```

Repetitions in the list can be removed using `sort/2`:

```
?- findall(Who, likes(X,Who), L), sort(L,SortedList).

SortedList = [brenda,carol,danny,john,mary,michael,stephen]
```

The query may involve compound sub-goals:

Make a list of rock artists liked by Brenda.

```
?- findall(A, (likes_artist(brenda,A), music_type(A,rock)), L).

A = _,

L = ['Eric Clapton','Bob Dylan']
```

Put the query in brackets to show that all its sub-goals are part of the 2nd argument. If there are many sub-goals, spread `findall` over several lines with indenting to make it more readable.

An answer may involve several variables. A structure must be specified to hold these:

Make a list of pairs of people who would go to a concert together.

```
?- findall(X-Y, (likes(X,Y), likes(Y,X), likes_artist(X,A),

   likes_artist(Y,A)), PairsList).

X = _,

Y = _,

A = _,

PairsList = [mary-carol, danny-michael, carol-mary, michael-danny]
```

Chapter 5 More advanced program control

Greater control over Prolog's searching mechanism can be obtained using several bips especially the 'cut' goal.

Unlike conventional programming languages, Prolog has little in the way of facilities to control program flow. Since the 'program' in Prolog is really the search and match operation by which reasoning takes place, control features are concerned with exerting influence over this - especially **backtracking**.

The common control facilities in Prolog are: the 'cut' (written '!'), 'fail', 'true' and 'repeat'. They are treated as built-in predicates. The 'cut', 'true' and 'repeat' always succeed while 'fail' always fails. What is useful of these bips is their side effects on backtracking. The 'cut' is just like an unidirectional 'valve': once the 'cut' is crossed it can never be crossed back in backtracking.

Placed as a goal in a query or condition it will immediately fail and cause Prolog to backtrack to the previous goal -- 'fail' artificially causes backtracking. Even though Prolog moves forward again after backtracking it can never get past the fail goal. Such a control strategy is called a failure-driven loop.

It is in most cases used in combination with other control bips for purposes like looping.

The 'true' is used mainly to make programs more readable and the answer of queries more sensible. It is sometimes used as a goal 'ored' with a condition involving 'fail' in order to make the overall goal succeed.

The 'repeat' is similar to 'cut' except that 'repeat' can be indefinitely re-satisfied while 'cut' cannot be re-satisfied. The 'true' cannot be re-satisfied either. The use of these control bips is exemplified below.

5.1 The cut

Prolog is good at automatic backtracking but uncontrolled backtracking can sometimes be a nuisance. Prolog provides a control facility, called "cut" for preventing backtracking.

Syntactically, a use of cut in a conjunction of goals looks just like the appearance of a goal which has the predicate ! and no arguments. As a goal, this succeeds immediately and cannot be re-satisfied. However, it also has side-effects which alter the way backtracking works afterwards. The effect is to make inaccessible the place markers for certain goals so that they cannot be re-satisfied.

When a cut is encountered as a goal, the system thereupon becomes committed to all choices made since the parent goal was involved. All other alternatives are discarded. Hence an attempt to re-satisfy any goal between the parent goal and the cut goal will fail.

A parent goal is the goal that caused the use of the rule containing the cut. We can look at the cut symbol as being rather like a fence that separates goals. Let us consider the following rule that has a conjunction of goals including the cut.

```
foo :- a, b, c, !, d, e, f.
```

Prolog will quite happily backtrack among goals, a, b, and c, until the success of c causing the "fence" to be crossed to the right to reach goal d. Then, backtracking can occur among d, e and f, perhaps satisfying the entire conjunction several times. However, if d fails, causing the "fence" to be crossed to the left, then no attempt will be made to re-satisfy goal c the entire conjunction of goals will fail, and the goal foo will also fail.

We can divide the common uses of "cut" into three main areas:

The first concerns places where we want to tell Prolog that it has found the right rule for a particular goal. Here, the cut says "if we get this far, we have picked the correct rule for this goal."

The second concerns places where we want to tell Prolog to fail a particular goal immediately without trying for alternative solutions. Here, we use the cut in conjunction with the fail predicate to say "if we get to here, we should stop trying to satisfy this goal."

The third concerns places where we want to terminate the generation of alternative solutions through backtracking. Here, the cut says "if we get to here, we have found the only solution to this problem, and there is no point in ever looking for alternatives.

5.1.1 Confirming the Choice of a Rule

Assume that we have a database of results of tennis games played by members of a club.

```
beat(tom,jim).

beat(ann,tom).

beat(pat,jim).
```

We can classify the players in the club into three categories.

Winner: every play who won all his/her games is a winner.

Fighter: any player that won some games and lost some is a fighter.

Sportsman: any player who lost all his/her games is a sportsman.

Now we can define a relation class(Player,Category) to classify the players.

```
class(X,fighter) :-
        beat(X,_),
        beat(_,X),!.

class(X,winner) :-
        beat(X,_),!.

class(X,sportsman) :-
        beat(_,X),!.

sum_to(1,1) :- !.

sum_to(N,Res) :-
```

```
N1 is N -1,
sum_to(N1,Res),
Res is Res + N.
```

5.1.2 The "cut-fail" Combination

How can we say something in Prolog like

Mary likes all animals but snakes.

We can generally express one part of this statement in Prolog:

Mary likes any X if X is an animal.

```
likes(mary,X) :- animal(X).
```

But in order to exclude snakes, we have to use a different expression:

If X is a snake then `Mary likes X' is not true,

otherwise if X is an animal then Mary likes X.

That something is not true can be said in Prolog by using a special goal, **fail**, which always fails, thus forcing the parent goal to fail.

```
likes(mary,X) :-
        snake(X),!,fail.

likes(mary,X) :-
        animal(X).
```

The first rule says that if X is a snake then the cut will prevent backtracking (thus excluding the second rule) and `fail` will cause the failure.

5.1.3 Terminating a "Generate and Test"

```
young_person(FirstName) :-

        age(FirstName,Age),

        Age < 20, !.
```

There are two parts in the above program. The first part is for getting a person and his/her age from the database and the second part is for testing whether his/her age is younger than 20 years old. If he/she is younger than 20 years old, we have already found somebody who is young and stop looking for the young person. Otherwise, continue to get another person from the database and test whether he/she is qualified to be the young person.

The use of 'true'

```
do_something:-

        likes(Person,Liked), DO SOMETHING, fail;

        true.
```

This is in fact a typical failure-driven control structure. Since 'likes/2' is re-satisfiable, the DO SOMETHING operation is repeated until there is no more clause in the KB satisfying

likes(Person,Liked). The 'true' is there to guarantee do_something/0 is always true.

The DO SOMETHING operation is usually a goal that has a side effect, like writing something onto the screen or a file (see next Chapter).

The combination of 'repeat' and 'fail'

```
do_something:-

        repeat,

        (

         DO SOMETHING, fail;

         true

        ).
```

The DO SOMETHING operation here is usually some goal which is not re-satisfiable but has a side effect, like writing something onto the screen or a file. If DO SOMETHING is re-satisfiable, like the example in Section 5.2, the 'repeat' bip is not needed.

Chapter 6 Input/Output and Use of Streams

We sometimes need to have communication with files. The Prolog program can read data from several input files called **input streams** and output data to several output files called **output streams**. Data from the terminal is treated as an input stream. Data output to the terminal is treated as an output stream.

At any time during the execution of the Prolog program, only two files are `active': one is for input and another for output. These two files are called the **current input stream** and **current output stream** respectively. These two streams by default correspond to the terminal but they can be changed to some files from time to time.

The current input stream can be changed to a file by the goal:

```
see(Filename)
```

Such a goal succeeds and causes, as a side effect, that the current input stream to be switched to `Filename`. For example, if we want to read something from `input_file` and then switch back to the terminal, we can use the following goals:

```
...

see('input_file'),

...                /* Read from the file. */

seen,

...
```

The current output stream can be changed to a file by the goal:

```
tell(Filename)
```

Such a goal succeeds and causes, as a side effect, that the current output stream to be switched to `Filename`. For example, if we want to write something to `output_file` and then switch back to the terminal, we can use the following goals:

```
...

tell('output_file'),

...                 /* Write to the file. */

told,

...
```

The goal `seen/0` closes the current input file. The goal `told/0` closes the current output file. The files are assumed to be sequential files which behave in the same way as the terminal. Each request to read something from the current input stream will cause reading at the current position in the current input stream. After that, the current position will be moved to the next item to read. If a reading request is made at the end

of the current input stream, the the atom `end_of_file` will be returned.

Similarly, each request to write something to the current output stream will put it at the end of the current output stream.

Data is input or output in one of two ways:

- a whole term at a time.
- character by character.

6.1 Processing Terms Using read and write

read/1 is used for reading terms from the current input stream. The goal

```
read(X)
```

will succeed if the next term T is read which is made to match X. If X is a variable, it will become instantiated to T. If the matching can not be made then the goal read(X) fails. If the goal fails then there will be no backtracking to read another term. Each term in the input file must be followed by a full stop and a carriage-return.

If read(X) is executed when the end of the current input stream has been reached then X will become instantiated to the atom end_of_file.

write/1 is used for writing terms to the current output stream. The goal

```
write(X)
```

will succeed if the term X is written to the current output stream.

Remember that text can be output by enclosing it in single quotes (thus making it an atom) and using the write predicate. For example,

```
write('Have a nice day !').
```

6.2 Formatted read and write

For formatted read and write some Prolog systems have additional predicates. LPA Prolog has fread/4 and fwrite/4:

fread(Format,Field_Width,Modifier,Term)

fwrite(Format,Field_Width,Modifier,Term)

The arguments are:

Format: a lower case letter denoting the type of term, e.g. a for atom, f for floating point number.

Field_Width,

Modifier,

Term, i.e. the term to be written or read.

fwrite/4

When writing atoms, use a +ve integer for a fixed field width (right justified), -ve for left-justified, and

0 for free width; and set the modifier to 0.

```
?- fwrite(a, 16, 0, hello).

            helloyes

?- fwrite(a, -16, 0, hello).

hello           yes
```

When the modifier is 0 or +ve the output must fit the width or an error will occur. But if the modifier is negative the output will be truncated to fit the width. Use a field width of 0 for free width output:

```
?- fwrite(a, 0, 0, 'Have a nice day').

Have a nice dayyes
```

Output of floating point numbers works similarly except that the modifier specifies the number of decimal places:

```
?- fwrite(f, 16, 3, 647.3456).

        647.346yes

?-  fwrite(f, -16, 3, 647.3456).

647.346         yes
```

fread/4

With fread/4, the field width specifies the number of characters read (which may include control characters). Again a field width of 0 means free width. But if the modifier is +ve, control characters will be read whereas they will not if it is -ve.

```
?- fread(a,5,0,Term).

grapefruit

Term = grape

?- fread(a,0,0,Term).

grapefruit

Term = 'grapefruit~M'

?- fread(a,0,-1,Term).

grapefruit

Term = grapefruit
```

For reading floats, the input must fit the width (if fixed width is used). The modifier again specifies the number of decimal places. Truncation of decimal places can occur.

```
?- fread(f,6,3,Float).

34.678

Float = 34.678

?-  fread(f,4,3,Float).

34.678

no
```

```
?- fread(f,0,3,Float).

345234.678

Float = 345234.678
```

When a variable appears in the argument of write/1, its value (if known) is written. Output goals do not re-succeed on backtracking - they just fail. They are said to be deterministic. Output is an example of a side effect of the query, i.e. something that continues to exist after the query has finished.

6.3 Manipulating Characters Using get0/1, get/1 and put/1

get0/1 is used for reading a single character from the current input stream. The goal get0(C) causes the current character to be read from the current input stream, and the variable C becomes instantiated to the ASCII code of this character.

get/1 is a variation of get0/1, which is used for reading only printable character. So it will skip over all non-printable characters from the current position in the input stream up to the first printable character.

put/1 is used for writing a single character to the current output stream. The goal put(A) causes a character to be written to the output stream where A gives the ASCII code of the character.

The following ASCII codes are useful:

put(46) gives full stop.

put(9) gives tab.

put(32) gives space.

6.4 Formatting the Output

nl/0 is used for starting a new line at output. tab/1 is used for inserting spaces into the output stream. The goal tab(N) causes N spaces to be output. The following are some sample programs.

Print (on screen) a list of names held on file.

```
printnames :-
        ee('names.dat'),
        repeat,
        read(Name),
        (Name = end_of_file, seen;
        write(Name), nl, fail).
```

Enter names at keyboard (terminated by `end') and output to a file.

```
filenames :-
        tell('names.dat'),
        repeat,
        read(Name),
        (Name = end, told;
        write(Name), put(46), nl, fail).
```

It should be noted that all the built-in Prolog predicates dealing with reading and writing

are deterministic, that is, when backtracking occurs, they cannot be re-satisfied. In Prolog, it is necessary to avoid repeated input or output.

6.5 Examples

6.5.1 A simple interface

Write a query to request a number, calculate its cube and output the answer.

```
?- write('Enter a number to be cubed '), read(N),
        Cube is N*N*N,
        write('The cube of your number is: '),write(Cube),nl,nl.
```

- Interaction:

```
Enter a number to be cubed |: 4.
The cube of your number is: 64
N = 4 ,
Cube = 64
```

Create a repeat…until loop to cycle back and ask for more input

Note that

read/1 does not re-succeed. Instead use repeat/0 which re-succeeds indefinitely.

If the user enters stop, the interaction will terminate.

```
?- repeat, write('Enter a number to be cubed (or stop. to quit): '),
    read(N),
    (N=stop,!;          %  ored goal: terminate or calculate cube
     Cube is N*N*N,
     write('The cube of your number is: '),write(Cube),nl,nl,
     fail
    ).
```

Sample interaction:

```
Enter a number to be cubed (or stop. to quit): |: 4.
The cube of your number is: 64

Enter a number to be cubed (or stop. to quit): |: 6.
The cube of your number is: 216
```

```
Enter a number to be cubed (or stop. to quit): |: stop.

N = stop ,

Cube = _
```

6.5.2 Reading from a file

Suppose people.pl contains first names of people:

```
george.

danny.

carol.

michael.

brenda.

john.
```

We can use read/1 to read names and check whether Mary likes them. When the end of the file is reached, read/1 will return the atom end_of_file.

```
?- see('people.pl'),

    repeat,

    read(X),

    (

    X=end_of_file,  write('Finished   reading file'), nl, ! ;

    likes(mary,X), write('Mary likes '), write(X), nl,

    fail

    ),

    seen.
```

Sample interaction:

```
Mary likes carol

Mary likes john

Finished reading file

X = end_of_file
```

6.6 Reading Programs and Updating Knowledge Base

In the previous section we discussed how Prolog communicates data with the files or

the terminal. In this section, we discuss how Prolog loads programs from the files or the terminal, and updates programs loaded in the knowledge base. Prolog provides several built-in predicates for carrying out the above tasks.

6.6.1 Reading Programs

There are two built-in predicates: consult and reconsult. We tell Prolog to read programs from a file F with the goal:

```
?- consult(F).
```

This goal succeeds with the effect that all clauses in F are read into the knowledge base and will be used by Prolog to answer further questions. If another file, e.g. F1, is read at some later time with the goal:

```
?- consult(F1).
```

all clauses from this file are added at the end of the knowledge base. For example,

```
consult('prog.pl').
```

The file name should be enclosed by single quotes. Prolog files usually have an extension pl.

The use of consult could cause some problems if some relation has defined differently in the two files. In that case, two definitions are both kept in the database that may cause some inconsistent uses of the relation.

reconsult can be used to solve this problem. A goal

```
?- reconsult(F).
```

will also read all clauses from the file F except that if there are clauses in F about a relation that has been previously defined in the knowledge base, the old definition will be superseded by the new clauses about this relation in F. So consult always *adds* new clauses while reconsult *redefines* previously defined relations. reconsult(F) will, however, not affect any relation about which there is no clauses in F.

6.6.2 Updating Knowledge Base

Several built-in predicates in Prolog can be used to add a clause to or remove a clause from the knowledge base.

A goal

```
?- assert(Clause).
```

will add the 'Clause' to the knowledge base.

A goal

```
?- asserta(Clause).
```

will add the 'Clause' at the beginning of the knowledge base.

A goal

```
?- assertz(Clause).
```

will add the 'Clause' at the end of the knowledge base.

A goal

```
?- retract(Clause).
```

will remove the first occurrence of the 'Clause' from the knowledge base. If the 'Clause' contains variables these will be instantiated.

A goal

```
?- retractall(Clause).
```

will remove all the occurrences of the 'Clause' from the knowledge base.

Before a predicate can be updated, we need to use a directive `:-dynamic` to instruct the Prolog compiler to regard it as dynamic, that is, its definition can be changed dynamically through the use of predicates such as `assert/1`, `asserta/1`, `assertz/1`, `retract/1` or `retractall/1`. For example, we can declare that the predicate `likes/2` can be changed dynamically as follows:

```
:- dynamic likes/2.

likes(joe,fish).

likes(joe,mary).

likes(mary,book).

likes(mary,wine).

likes(john,book).
```

So in the following queries, we can either assert or retract some facts which declare the relation likes/2.

```
?- assert(likes(joe,book)).

?- retract(likes(joe,mary)).

?- retractall(likes(mary,X)).
```