Prolog Programming

A First Course

Paul Brna

January 24, 2001

Abstract

The course for which these notes are designed is intended for undergraduate students who have some programming experience and may even have written a few programs in **Prolog**. They are not assumed to have had any formal course in either propositional or predicate logic.

At the end of the course, the students should have enough familiarity with **Prolog** to be able to pursue any undergraduate course which makes use of **Prolog**.

This is a rather ambitious undertaking for a course of only twelve lectures so the lectures are supplemented with exercises and small practical projects wherever possible.

The **Prolog** implementation used is SICStus Prolog which is closely modelled on Quintus Prolog (SICS is the Swedish Institute of Computer Science). The reference manual should also be available for consultation [SICStus, 1988].

Contents

List of Figures

Preface

A Warning

These notes are under development. Much will eventually change. Please help to make these notes more useful by letting the author know of any errors or missing information. Help future generations of AI2 students!

The Intended Audience

The course for which these notes are designed is intended for undergraduate students who have some programming experience and may even have written a few programs in **Prolog**. They are not assumed to have had any formal course in either propositional or predicate logic.

The Objective

At the end of the course, the students should have enough familiarity with **Prolog** to be able to pursue any undergraduate course which makes use of **Prolog**.

The original function was to provide students studying Artificial Intelligence (AI) with an intensive introduction to **Prolog** so, inevitably, there is a slight bias towards AI.

The Aims

At the end of the course the students should be:

- familiar with the basic syntax of the language
- able to give a declarative reading for many **Prolog** programs
- able to give a corresponding procedural reading
- able to apply the fundamental programming techniques
- familiar with the idea of *program as data*
- able to use the facilities provided by a standard trace package to debug programs
- familiar with the fundamental ideas of the predicate calculus
- familiar with the fundamental ideas specific to how **Prolog** works

Course Structure

This is a rather ambitious undertaking for a course of only twelve lectures so the lectures are supplemented with exercises and small practical projects wherever possible.

Acknowledgements

These notes are based on a previous version used with the students of the AI2 course in **Prolog** during the session 1985–87 and 1988–89 at the Department of Artificial Intelligence, Edinburgh University. My thanks for the feedback that they supplied.

Chapter 1

Introduction

Prolog is **PRO**gramming in **LOG**ic

A few points must be cleared up before we begin to explore the main aspects of **Prolog**.

These notes are supplemented with exercises and suggestions for simple practicals. It is assumed that you will do most of this supplementary work either in your own time, for tutorials or during practical sessions.

Each chapter will start with a simple outline of its content and finish with a brief description of what you should know upon completion of the chapter and its exercises.

Important points will be boxed and some technical and practical details which are not immediately essential to the exposition will be

written in a smaller font.

1.1 Declarative vs Procedural Programming

Procedural programming requires that the programmer tell the computer what to do. That is, *how* to get the output for the range of required inputs. The programmer must know an appropriate algorithm.

Declarative programming requires a more descriptive style. The programmer must know *what* relationships hold between various entities.

Pure[1] **Prolog** allows a program to be read either declaratively or procedurally. This dual semantics is attractive.

1.2 What Kind of Logic?

Prolog is based on **F**irst **O**rder **P**redicate **L**ogic —sometimes abbreviated to FOPL.

[1] **Prolog**, like LISP, has a pure subset of features. The implication is that some features of both languages are regarded as *impure* —these are often provided for efficiency or for useful, but strictly unnecessary features. The impure features of **Prolog** damage the pleasing equality between the declarative and procedural readings of **Prolog** programs.

First order *predicate logic* implies the existence of a set of predicate symbols along with a set of connectives.

First order predicate logic implies that there is no means provided for "talking about" the predicates themselves.

Prolog is based on FOPL but uses a restricted version of the *clausal form*. Clausal form is a particular way of writing the propositions of FOPL. The restriction is known as *Horn clause form*.

Prolog is a so-called *logic programming* language. Strictly, it is not the only one but most such languages are its descendents.

We will spend a little time outlining the basic ideas underlying both propositional and predicate logic. It is not the intention to use **Prolog** as a vehicle to teach logic but some appreciation of the issues is invaluable.

1.3 A Warning

Prolog is known to be a difficult language to master. It does not have the familiar control primitives used by languages like RATFOR, ALGOL and PASCAL so the system does not give too much help to the programmer to employ structured programming concepts.

Also, many programmers have become used to strongly typed languages. **Prolog** is very weakly typed indeed. This gives the programmer great power to experiment but carries the obvious responsibility to be careful.

Another major difference is the treatment of variables —special attention should be given to understanding variables in **Prolog**.

Prolog provides a search strategy for free —there is a cost. The programmer has to develop a methodology to handle the unexpected consequences of a faulty program. In particular, pay careful attention to the issue of backtracking.

It is usual to assume that telling people how they can go wrong is an encouragement to do exactly that —go wrong. The approach taken here is to make the known difficulties explicit.

1.4 A Request

These notes are slowly being improved. Many further exercises need to be added along with more example programs and improvements to the text.

If you have any comments that will help in the development of these notes then please send your comments to:

Paul Brna
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN

Chapter 2

Knowledge Representation

We take a very brief and informal look at both the propositional calculus and first order predicate logic.
We then restrict our attention to a form of predicate logic which translates directly into **Prolog**.
This requires that we introduce a simple vocabulary that describes the syntax of **Prolog**.
Here, we concentrate on an informal description of the fundamental units which are:

clause, rule, fact,
goal, subgoal,
logical variable, constant, atom,
functor, argument, arity.

An explanation as to how statements can be represented in **Prolog** form is given.

How do we represent what we know? The simplest analysis requires that we distinguish between *knowledge how* –procedural knowledge such as how to drive a car— and *knowledge that* —declarative knowledge such as knowing the speed limit for a car on a motorway.

Many schemes for representing knowledge have been advanced —including full first order predicate logic. The strong argument for classical (first order predicate) logic is that it has a well understood theoretical foundation.

2.1 Propositional Calculus

The propositional calculus is based on statements which have truth values (**true** or **false**).

The calculus provides a means of determining the truth values associated with statements formed from "atomic" statements. An example:

If **p** stands for "fred is rich" and **q** for "fred is tall" then we may form statements such as:

Symbolic Statement	Translation
$p \vee q$	p or q
$p \wedge q$	p and q
$p \Rightarrow q$	p logically implies q
$p \Leftrightarrow q$	p is logically equivalent to q
$\neg p$	not p

Note that $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are all binary connectives. They are sometimes referred to, respectively, as the symbols for disjunction, conjunction, implication and equivalence. Also, $\neg$ is unary and is the symbol for negation.

If propositional logic is to provide us with the means to assess the truth value of compound statements from the truth values of the 'building blocks' then we need some rules for how to do this.

For example, the calculus states that **p**$\vee$**q** is **true** if either **p** is **true** or **q** is **true** (or both are **true**). Similar rules apply for all the ways in which the building blocks can be combined.

A Problem

If **p** stands for "all dogs are smelly" and **p** is **true** then we would like to be able to prove that "my dog fido is smelly".

We need to be able to get at the structure and meaning of statements. This is where (first order[1]) predicate logic is useful.

2.2 First Order Predicate Calculus

The predicate calculus includes a wider range of entities. It permits the description of relations and the use of variables. It also requires an understanding of *quantification*.

The language of predicate calculus requires:

Variables

Constants —these include the *logical constants*

Symbol	Meaning
$\vee$	or
$\wedge$	and
$\neg$	not
$\Rightarrow$	logically implies
$\Leftrightarrow$	logically equivalent
$\forall$	for all
$\exists$	there exists

The last two logical constants are additions to the logical connectives of propositional calculus —they are known as quantifiers. The *non-logical* constants include both the 'names' of entities that are related and the 'names' of the relations. For example, the constant **dog** might be a relation and the constant **fido** an entity.

[1] Do not worry about the term *first order* for now. Much later on, it will become relevant.

Predicate —these relate a number of entities. This number is usually greater than one. A predicate with one argument is often used to express a property *e.g.* **sun(hot)** may represent the statement that "the sun has the property of being hot".

If there are no arguments then we can regard the 'predicate' as standing for a statement à la the propositional calculus.

Formulæ —these are constructed from predicates and formulæ[2]. The logical constants are used to create new formulæ/ from old ones. Here are some examples:

Formula(e)	New Formula
dog(fido)	$\neg$ dog(fido)
dog(fido) and old(fido)	dog(fido)$\vee$ old(fido)
dog(fido) and old(fido)	dog(fido)$\wedge$ old(fido)
dog(fido) and old(fido)	dog(fido)$\Rightarrow$ old(fido)
dog(fido) and old(fido)	dog(fido)$\Leftrightarrow$ old(fido)
dog(X)	$\forall$X.dog(X)
dog(X)	$\exists$X.dog(X)

Note that the word "and" used in the left hand column is used to suggest that we have more than one formula for combination —and *not* necessarily a conjunction.

In the last two examples, "dog(X)" contains a variable which is said to be *free* while the "X" in "$\forall$X.dog(X)" is *bound*.

Sentence —a formula with no free variables is a sentence.

Two informal examples to illustrate quantification follow:

$\forall$X.(man(X)$\Rightarrow$mortal(X))	All men are mortal
$\exists$X.elephant(X)	There is at least one elephant

The former is an example of *universal* quantification and the latter of *existential* quantification.

We can now represent the problem we initially raised:

$$\forall X.(dog(X)\Rightarrow smelly(X))\wedge dog(fido)\Rightarrow smelly(fido)$$

To verify that this is correct requires that we have some additional machinery which we will not discuss here.

2.3 We Turn to Prolog

Prolog provides for the representation of a subset of first order predicate calculus. The restrictions on what can be done will become clearer later. We will now explain how we can write logical statements in **Prolog**.

[2]Note that this is a *recursive* definition.

If "the capital of france is paris" then we can represent this in predicate calculus form as[3]:

france has_capital paris

We have a binary relationship (two things are related) written in infix form. That is, the relationship is written between the two things related.

The relationship (or predicate) has been given the name "has_capital" —hence we say that the predicate name is "has_capital".

And in **Prolog** form by such as:

has_capital(france,paris).

where we write a *prefix* form and say that the relationship takes two *arguments*. Prefix because the relationship is indicated before the two related things.

Note that, in **Prolog**, if the name of an object starts with a lower case letter then we refer to a specific object. Also, there must be no space between the predicate name and the left bracket "(". The whole thing also ends in a "." as the last character on the line.

> The exact rule for the termination of a clause is that a clause must end with a "." followed by *white space* where white space can be any of {space,tab,newline,end_of_file}. It is safest to simply put "." followed by newline.

Also note that relations do not need to hold between exactly two objects. For example,

meets(fred,jim,bridge)

might be read as

fred meets jim by the bridge

Here, three objects are related so it makes little sense to think of the relation *meets* as binary —it is ternary.

If we can relate two objects or three then it is reasonable to relate n where $n \geq 2$. Is there any significance to a relationship that relates one or even zero objects? A statement like

jim is tall

might be represented either as

[3]The failure to capitalise "france" and "paris" is quite deliberate. In **Prolog**, named, specific objects (*i.e.* the atoms) usually start with a lower case letter.

tall(jim)

or

jim(tall)

It is, perhaps, preferable to see **tall**ness as being a property which is possessed by **jim**.

A 'relation' that has no arguments can be seen as a single proposition. Thus the binary relation "france has_capital paris" above might be rewritten as the statement "france_has_capital_paris" —a relation with no arguments.

2.4 Prolog Constants

If we have

loves(jane,jim).

then **jane** and **jim** refer to specific objects. Both **jane** and **jim** are *constants*. In particular, in DEC-10 **Prolog** terminology, both are *atoms*. Also, "loves" happens to be an atom too because it refers to a specific relationship. Generally speaking, if a string of characters starts with a lower case letter, the DEC-10 family of **Prolog**s assume that the entity is an atom.

There are constants other than atoms —including integers and real numbers.

A constant is an atom or a number. A number is an integer or a real number[4]. The rules for an atom are quite complicated:

quoted item	'anything but the single quote character'
word	lower case letter followed by any letter, digit or _ (underscore)
symbol	any number of {+, -, *, /, \, ^, <, >, =, ', ~, :, ., ?, @, #, $, &}
special item	any of { [], {}, ;, !, %}

So the following are all atoms:

likes_chocolate, fooX23, ++*++, ::=, 'What Ho!'

By the way, you *can* include a single quote within a quoted atom —just duplicate the single quote. This gives the quoted atom with a single quote as:

''''

A practical warning: remember to pair off your (single) quote signs when inputing a quoted atom or **Prolog** may keep on swallowing your input looking for that elusive single quote character. This is one of the most common syntactic errors for beginners.

While we are on the subject, another common error is to assume that a double quote (") behaves like a single quote —*i.e.* that the term `"Hello"` is an atom just like `'Hello'`. This is not so. When you do find out what sensible things can be done with the double quote then remember to pair them off.

[4]Referred to as a *float* in the SICStus **Prolog** manual [SICStus, 1988].

Because **Prolog** is modelled on a subset of first order predicate logic, all predicate names must be constants —but not numbers. In particular,

No predicate may be a *variable*

That is, we cannot have **X(jane,jim)** as representing the fact that **jane** and **jim** are related in some unknown way.

2.5 Goals and Clauses

We distinguish between a **Prolog** *goal* and **Prolog** *clause*. A clause is the syntactic entity expressing a relationship as required by **Prolog** —note that we will regard the '.' as terminating a clause (this is not strictly correct).

loves(jane,jim)	is a **goal**
loves(jane,jim).	is a *unit* **clause**

The adjectives *unit* and *non-unit* distinguish two kinds of clause —intuitively, *facts* and *rules* respectively.

Exercise 2.1 *Here is the first opportunity to practice the representation of some statement in* **Prolog** *form.*

1. *bill likes ice-cream*
2. *bill is tall*
3. *jane hits jimmy with the cricket bat*
4. *john travels to london by train*
5. *bill takes jane some edam cheese*
6. *freddy lives at 16 throgmorton street in london*

The failure to capitalise "freddy", "london" etc. is a reminder that the version of **Prolog** *that we are using requires that constants should not start with an upper case letter.*

Note that there may be several ways of representing each of these statements.

2.6 Multiple Clauses

A predicate may be *defined* by a set of clauses with the same predicate name and the same number of arguments.

We will therefore informally describe the way in which this is handled through an example. The logical statement (in first order form)

$$\text{squared}(1,1) \wedge \text{squared}(2,4) \wedge \text{squared}(3,9)$$

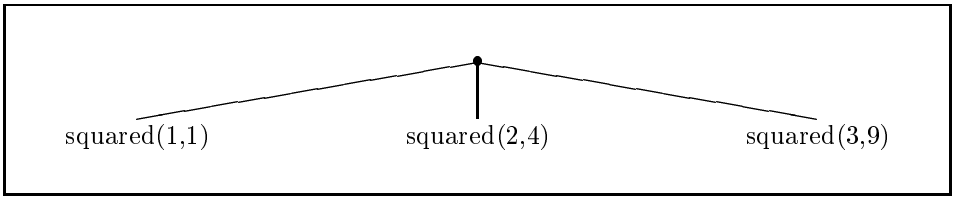
is to be represented as three distinct **Prolog** clauses.

```
squared(1,1).
squared(2,4).
squared(3,9).
```

Note that this way of turning a conjunctive statement into **Prolog** is one of the fundamental restrictions previously mentioned. There are more to follow.

All the above clauses are unit clauses —this is not a necessary requirement. See section 2.12 for an example with both unit and non-unit clauses.

We now introduce a graphical representation which will be used in a number of different ways. The idea we use here is to represent a program (in this case, consisting of a set of unit clauses) as a tree.



This tree is an example of an OR tree.

It might have been expected that we would call this an AND tree but, when we are trying to determine whether a statement such as **squared(1,1)** is true then we might use *either* the first clause *or* the second *or* the third and so on.

Exercise 2.2 *Represent each of these statements as a set of* **Prolog** *clauses.*

1. *bill only eats chocolate, bananas or cheese.*
2. *the square root of 16 is 4 or -4.*
3. *wales, ireland and scotland are all countries.*

2.7 Rules

The format is:

```
divisible_by_two:-
                  even.
```

This is a *non-unit* **clause**.

In general, a clause consists of two parts: the *head* and the *body*[5].

The *head* is **divisible_by_two** and the *body* is **even** —**even** is sometimes referred to as a **subgoal**.

Note that the symbol ":-" is read as *if*. An informal reading of the clause is "**divisible_by_two** is **true** if **even** is **true**" which is equivalent to "**even** $\Rightarrow$ **divisible_by_two**".

Any number of subgoals may be in the body of the rule.

[5]These two body parts are 'joined' by the neck. There is an analogous concept in the **Prolog** literature.

No more than one goal is allowed in the head

This is another way in which **Prolog** is a restriction of full first order predicate calculus. For example, we cannot translate **rich(fred)** $\Rightarrow$ **happy(fred)**$\wedge$**powerful(fred)** directly into the **Prolog** version **happy(fred),powerful(fred) :- rich(fred)**.

See section 2.10 for an example of a clause with more than one subgoal in the body. A *fact* is effectively a *rule* with no subgoals.

You may have noticed that, even if it is held that "even" is a relation, it does not seem to relate anything to anything else.

The rule is not as much use as it might be because it does not reveal the more interesting relationship that

> A number is divisible by two if it is even

We can express this with the help of the *logical variable*. Here is the improved rule:

```
divisible_by_two(X):-
                    even(X).
```

This is also a *non-unit* **clause**. The named logical variable is **X**. This **Prolog** clause is equivalent to the predicate calculus statement $\forall$ **X. (even(X)** $\Rightarrow$ **divisible_by_two(X))**.

2.8 Semantics

Here is an informal version of the procedural semantics for the example above:

> If we can find a value of **X** that satisfies the goal **even(X)** then we have also found a number that satisfies the goal **divisible_by_two(X)**.

The declarative semantics.

> If we can prove that **X** is "even" then we have proved that **X** is "divisible_by_two".
>
> Note that there is an implicit universal quantification here. That is, *for all objects* those that are even are also divisible by two.
>
> $\forall$X.(even(X)$\Rightarrow$ divisible_by_two(X))
>
> Also note that the *head* goal is found on the right of the standard logical implication symbol. It is a common error to reverse the implication.

Two final examples of a single rule. The first:

> all scots people are british

can be turned into:

```
british(Person):-
                scottish(Person).
```

Note that **Person** is another logical variable. Now for the final example:

> if you go from one country to another they you are a tourist

turns into:

```
tourist(P):-
                move(P,Country1,Country2).
```

where **move(P,A,B)** has the informal meaning that a person **P** has moved from country **A** to country **B**.

There is a problem here. We really need to specify that **Country1** and **Country2** are legitimate and distinct countries[6].

Exercise 2.3 *Represent these statements as single non-unit clauses (rules):*

1. *all animals eat custard*
2. *everyone loves bergman's films*
3. *jim likes fred's possessions*
4. *if someone needs a bike then they may borrow jane's*

2.9 The Logical Variable

In the DEC-10 **Prolog** family, if an object is referred to by a name starting with a capital letter then the object has the status of a *logical variable*. In the above rule there are two references to X. All this means is that the two references are to the *same* object —whatever that object is.

The *scope rule* for **Prolog** is that two uses of an identical name for a logical variable *only* refer to the same object if the uses are within a single clause. Therefore in

```
happy(X):-
                healthy(X).
wise(X):-
                old(X).
```

[6]This could be enforced by the **move/3** relation (predicate) but this would produce an unnaturally specific version of moving. The real solution is to provide some predicate such as **not_same/2** which has the meaning that **not_same(P1,P2)** precisely when **P1** is not the same as **P2**.

the two references to **X** in the first clause do not refer to the same object as the references to **X** in the second clause. By the way, this example is a sort that is discussed in section 2.11.

Do not assume that the word *logical* is redundant. It is used to distinguish between the nature of the variable as used in predicate calculus and the variable used in imperative languages like BASIC, FORTRAN, ALGOL and so on. In those languages, a variable name indicates a storage location which may 'contain' different values at different moments in the execution of the program.

The logical variable *cannot* be overwritten with a new value

Although this needs some further comments, it is probably better to start with this statement and qualify it later.

For example, in Pascal:

X:= 1; X:= 2;

results in the assignment of 2 to X. In **Prolog**, once a logical variable has a value, then it cannot be assigned a different one. The logical statement

X=1 $\wedge$ X=2

cannot be true as X cannot be both '2' and '1' simultaneously. An attempt to make a logical variable take a new value will fail.

2.10 Rules and Conjunctions

A man is happy if he is rich and famous

might translate to:

```
happy(Person):-
                man(Person),
                rich(Person),
                famous(Person).
```

The ',' indicates the conjunction and is roughly equivalent to the $\wedge$ of predicate calculus. Therefore, read ',' as 'and'[7]. The whole of the above is one (non-unit) single clause.

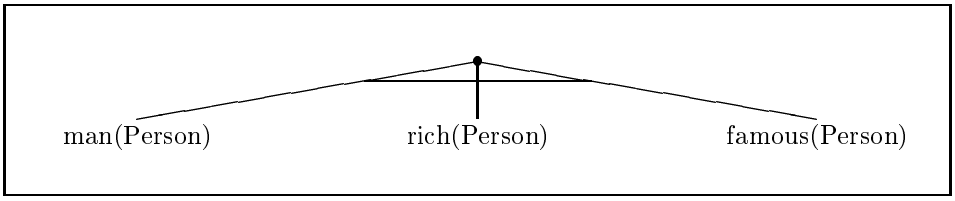
It has three subgoals in its body —these subgoals are 'conjoined'.

In this single clause, the logical variable **Person** refers to the same object throughout.

By the way, we might have chosen any name for the logical variable other than **Person**. It is common practice to name a logical variable in some way that reminds you of what kind of entity is being handled.

We now describe this clause graphically. In this case, we are going to represent conjunctions using an AND tree. Here is an AND tree that represents the above.

[7] It's meaning is more accurately captured by the procedural 'and then'.



The way in which we discriminate between an OR tree and an AND tree is the use of a horizontal bar to link the subgoals. We need this distinction because we are going to represent the structure of a program using a combined AND/OR tree.

Exercise 2.4 *A few more exercises. Each of these statements should be turned into a rule (non-unit clause) with at least two subgoals —even though some statements are not immediately recognisable as such:*

1. *you are liable to be fined if your car is untaxed*
2. *two people live in the same house if they have the same address*
3. *two people are siblings if they have the same parents*

2.11 Rules and Disjunctions

Someone is happy if they are healthy, wealthy or wise.

translates to:

```
happy(Person):-
                healthy(Person).
happy(Person):-
                wealthy(Person).
happy(Person):-
                wise(Person).
```

Note how we have had to rewrite the original informal statement into something like:

Someone is happy if they are healthy OR
Someone is happy if they are wealthy OR
Someone is happy if they are wise

We have also assumed that each clause is (implicitly) universally quantified. *i.e.* the first one above represents $\forall$**X.(healthy(X)**$\Rightarrow$**happy(X))**.

The predicate name "happy' is known as a *functor*.

The functor **happy** has one *argument*.

We describe a predicate with name "predname" with arity "n" as **predname/n**. It has one argument —we say its *arity* is 1.

The predicate **happy/1** is defined by three clauses.

Exercise 2.5 *Each of these statements should be turned into several rules:*

1. *you are british if you are welsh, english, scottish or northern irish*
2. *you are eligible for social security payments if you earn less than £ 28 per week or you are an old age pensioner*
3. *those who play football, rugger or hockey are sportspeople*

2.12 Both Disjunctions and Conjunctions

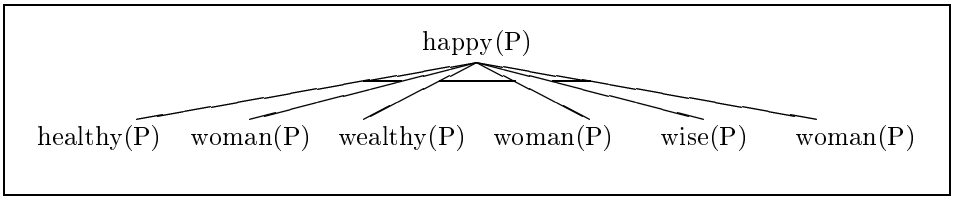
We combine both disjunctions and conjunctions together. Consider:

```
happy(Person):-
        healthy(Person),woman(Person).
happy(Person):-
        wealthy(Person),woman(Person).
happy(Person):-
        wise(Person),woman(Person).
```

This can be informally interpreted as meaning that

> A woman is happy if she is healthy, wealthy or wise

We now combine the OR tree representation together with an AND tree representation to form an AND/OR tree that shows the structure of the definition of **happy/1**.



Note that the logical variable in the diagram has been renamed to **P**. There is *no* significance in this renaming.

2.13 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to represent any simple fact in legal **Prolog**.
You should be able to split up a disjunctive expression into a set of **Prolog** clauses.
You should be able to express a simple conjunctive expression as a single clause.
You should be able to represent most rules in legal **Prolog**.

There is no perfect solution to the problem of representing knowledge. You may generate representations that differ wildly from someone else's answers. To find out which answer is best and in what context will require some deeper thought.

Exercise 2.6 *Here is a small set of problems that require you to convert propositions into* **Prolog** *clauses. Make sure you explain the meaning of your representation:*

1. $a \Rightarrow b$
2. $a \vee b \Rightarrow c$
3. $a \wedge b \Rightarrow c$
4. $a \wedge (b \vee c) \Rightarrow d$
5. $\neg a \vee b$

Exercise 2.7 *A simple collection of problems. Represent each statement as a single* **Prolog** *clause:*

1. *Billy studies AI2*
2. *The population of France is 50 million*
3. *Italy is a rich country*
4. *Jane is tall*
5. *2 is a prime number*
6. *The Welsh people are British*
7. *Someone wrote Hamlet*
8. *All humans are mortal*
9. *All rich people pay taxes*
10. *Bill takes his umbrella if it rains*
11. *If you are naughty then you will not have any supper*
12. *Firebrigade employees are men over six feet tall*

Chapter 3

Prolog's Search Strategy

So far we have concentrated on describing a fact or rule.
Now we have to discover how to make **Prolog** work for us.
Here, we informally introduce **Prolog**'s search strategy.
This requires introducing the ideas of **Prolog**'s top level, how to query **Prolog**, how **Prolog** copes with searching through a number of clauses, matching, unification, resolution, binding, backtracking and unbinding.

Search is a major issue. There are many ways to search for the solution to a problem and it is necessary to learn suitable algorithms that are efficient. **Prolog** provides a single method of search for free. This method is known as *depth first search.*

You should find that **Prolog** enables the programmer to implement other search methods quite easily.

Prolog's basic search strategy is now going to be outlined. To do this we need to consider something about the **Prolog** system.

Prolog is an interactive system. The interactions between the programmer and the **Prolog** system can be thought of as a conversation. When the **Prolog** system is entered we are at *top level.* The system is waiting for us to initiate a 'conversation'.

3.1 Queries and Disjunctions

Informally, a *query* is a goal which is submitted to **Prolog** in order to determine whether this goal is true or false.

As, at top level, **Prolog** normally expects queries it prints the prompt:

```
?-
```

and expects you to type in one or more goals. We tell the **Prolog** system that we have finished a query —or any clause— by typing "." followed by typing the key normally labelled "RETURN".

A very common syntax error for beginners is to press RETURN before ".". This is not a problem —just type in the missing "." followed by another RETURN.

We look at the case where we only want to solve one goal. Perhaps we would like to determine whether or not

woman(jane)

In this case we would type this in and see (what is actually typed is emboldened):

?- **woman(jane).**

Now **?- woman(jane).** is also a clause. Essentially, a clause with an empty head.

We now have to find out "if jane is a woman". To do this we must *search* through the facts and rules known by **Prolog** to see if we can find out whether this is so.

Note that *we* make the distinction between facts and rules —not **Prolog**. For example, **Prolog** does not search through the facts before the rules. Here are some facts assumed to be known[1]:

Program Database
woman(jean).
man(fred).
woman(jane).
woman(joan).
woman(pat).

In order to solve this goal **Prolog** is confronted with a *search problem* which is trivial in this case. How should **Prolog** search through the set of (disjunctive) clauses to find that it is the case that "jane is a woman"?

Such a question is irrelevant at the level of predicate calculus. We just do not want to know how things are done. It is sufficient that **Prolog** can find a solution. Nevertheless, **Prolog** is not pure first order predicate calculus so we think it important that you face up to this difference fairly early on.

The answer is simple. **Prolog** searches through the set of clauses in the same way that we read (in the west). That is, from top to bottom. First, **Prolog** examines

woman(jean).

and finds that

woman(jane).

does not *match*. See figure 3.1 for the format we use to illustrate the failure to match.

[1] At some point we had to input these facts into the system. This is usually done by creating a *file* containing the facts and rules needed and issuing a *command* that **Prolog** is to *consult* the file(s). Use the command

consult(filename).

where filename is the name of your file. A *command* is very like a query. A query is written something like **?- woman(X)**. The result (on the screen) is **X= something** followed by **yes** or the word **no** (if there is no such **X**). A command is written something like **:- woman(X)**. The result is that the system will not print the binding for **X** (if there is one) (or the word **yes**) or will print the symbol **?** if the query failed. The reason for the distinction between a query and a command will be explained later.

We introduce the term *resolution table*. We use this term to represent the process involved in matching the current goal with the head goal of a clause in the program database, finding whatever substitutions are implied by a successful match, and replacing the current goal with the relevant subgoals with any substitutions applied.

We illuminate this using a 'window' onto the resolution process (the resolution table). If the match fails then no substitutions will apply and no new subgoals will replace the current goal.

The term *substitution* is connected with the concept of associating a variable with some other **Prolog** object. This is important because we are often interested in the objects with which a variable has been associated in order to show that a query can be satisfied.

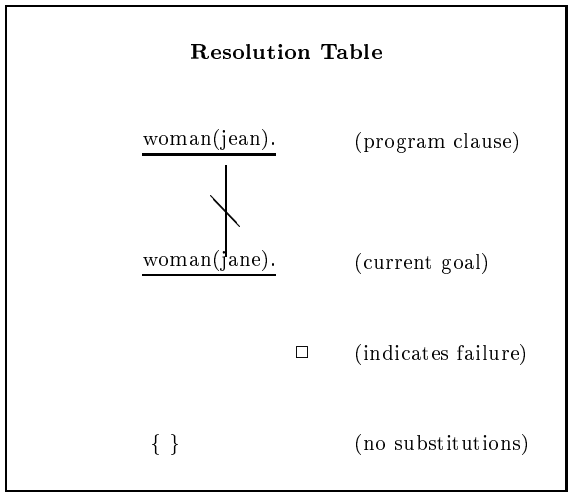


Figure 3.1: A Failed Match

This failure is fairly obvious to us! Also, it is obvious that the next clause **man(fred).** doesn't match either —because the query refers to a different relation (predicate) than **man(fred)**. From now on we will never consider matching clauses whose predicate names (and arities) differ.

Prolog then comes to look at the third clause and it finds what we want. All we see (for the whole of our activity) is:

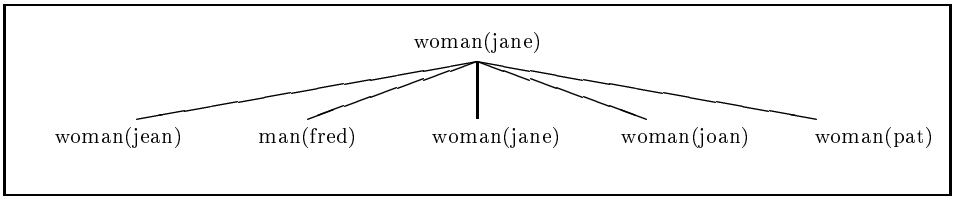
?- **woman(jane).**

yes

?-

Now think about how the *search space*[2] might appear using the AND/OR tree representation. The tree might look like:

[2]This term is used informally. The basic idea is that a program has an initial structure which can be represented as a tree. The nodes of the tree are goals and the arcs represent the rules used to invoke a particular goal or set of goals. A computation can be regarded very roughly as a path through this tree (really, a subtree).



We see that the search would zig zag across the page from left to right — stopping when we find the solution.

Note that we will normally omit facts from the representation of this 'search space'. In this case we would have a very uninteresting representation.

3.2 A Simple Conjunction

Now to look at a goal which requires **Prolog** to solve two subgoals. Here is our set of facts and one rule.

Program Database
woman(jean).
man(fred).
wealthy(fred).
happy(Person):-
woman(Person),
wealthy(Person).

We shall ask whether "jean is happy". We get this terminal interaction:

```
?- happy(jean).

no

?-
```

Now why is this the case? We said that we would not bother with clauses with differing predicate names. **Prolog** therefore has only one choice —to try using the single rule. It has to match:

happy(jean)

against

happy(Person)

We call this matching process *unification*. What happens here is that the logical variable **Person** gets *bound* to the atom **jean**. You could paraphrase "bound" as "is temporarily identified with". See figure 3.2 for what happens in more detail.

> In this case the match produces a substitution, **Person=jean**, and two subgoals replace the current goal. The substitution of **Person** by **jean** is known as a *unifier* and often written *Person/jean*. The process of replacing a single goal by one or more subgoals —with whatever substitutions are applicable— is part of the *resolution process*.

To solve our problem, **Prolog** must set up two subgoals. But we must make sure that, since **Person** is a logical variable, that everywhere in the rule that **Person** occurs we will replace **Person** by **jean**.

We now have something equivalent to:

```
happy(jean):-
                woman(jean),
                wealthy(jean).
```

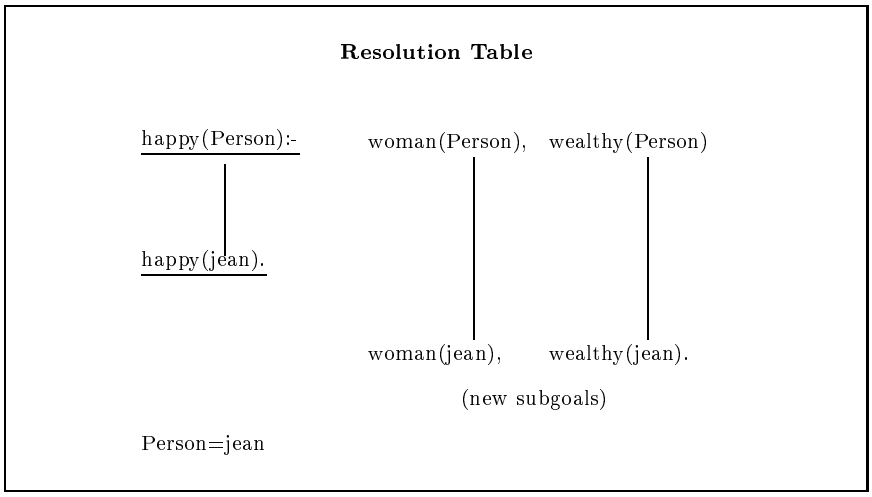


Figure 3.2: A Successful Match

So the two subgoals are:

```
woman(jean)
wealthy(jean)
```

Here we come to our next problem. In which order should **Prolog** try to solve these subgoals? Of course, in predicate logic, there should be no need to worry about the order. It makes no difference —therefore we should not need to know how **Prolog** does the searching.

Prolog is not quite first order logic yet. So we will eventually need to know what goes on. The answer is that the standard way to choose the subgoal to work on first is again based on the way we read (in the west)! We try to solve the subgoal **woman(jean)** and *then* the subgoal **wealthy(jean)**.

There is only one possible match for **woman(jean)**: our subgoal is successful. However, we are not finished until we can find out if **wealthy(jean)**.

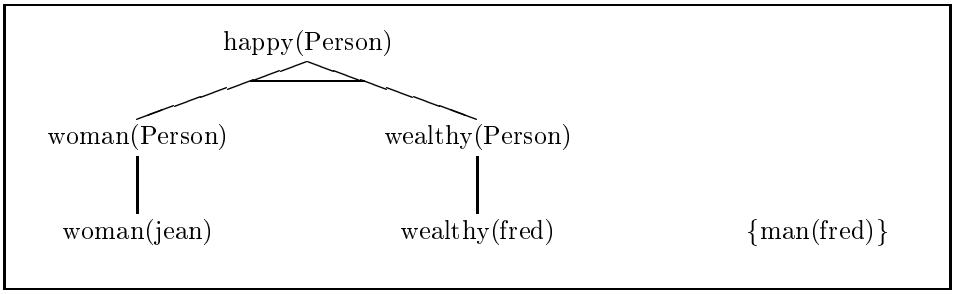
There is a possible match but we cannot *unify*

```
wealthy(fred)
```

with

```
wealthy(jean)
```

So **Prolog** cannot solve our top level goal —and reports this back to us. Things would be much more complicated if there were any other possible matches. Now to look at the (non-standard) AND/OR tree representation of the search space. Here it is:



> Note that it becomes very clear that knowing that "fred is a man" is not going to be of any use. That is why **man(fred)** is in braces. From now, we will exclude such from our 'search space'.

We can now see that the way **Prolog** searches the tree for AND choices is to zig zag from left to right across the page! This is a bit like how it processes the OR choices except that **Prolog** must satisfy all the AND choices at a node before going on.

> Zig zagging from left to right is not the whole story for this goal. Once we reach **wealthy(Person)** with *Person/jean* and it fails we move back (backtracking) to the goal **woman(Person)** and break the binding for **Person** (because this is where we made the binding *Person/jean*). We now start going from left to right again (if you like, forwardtracking).

3.3 Conjunctions and Disjunctions

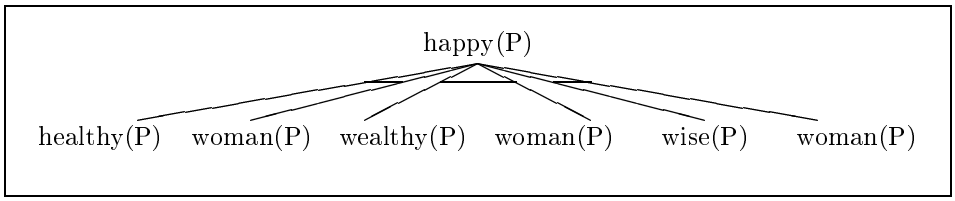
We are now ready for the whole thing: let us go back to the set of rules as found in section 2.12 and some basic facts.

Program Database
woman(jean).
woman(jane).
woman(joan).
woman(pat).
wise(jean).
wealthy(jane).
wealthy(jim).
healthy(jim).
healthy(jane).
healthy(jean).
happy(P):-
healthy(P),
woman(P).
happy(P):-
wealthy(P),
woman(P).
happy(P):-
wise(P),
woman(P).

and consider the solution of the goal

happy(jean)

Here is the standard AND/OR tree representation of the search space again:



and the goal succeeds.

Note that

1. Both the subgoal **healthy(jean)** and **woman(jean)** have to succeed for the whole goal to succeed.
2. We then return to the top level.

Now consider the top level goal of

happy(joan)

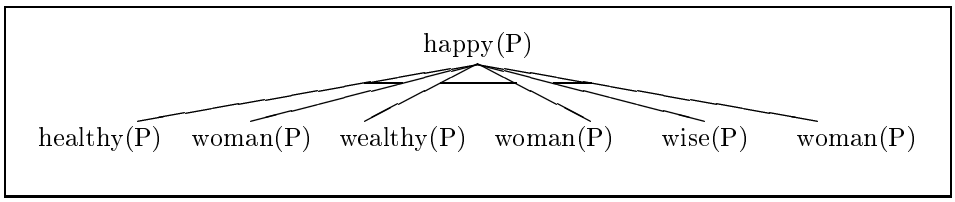
The resolution process generates the subgoals **healthy(joan)** and **woman(joan)** from the first clause for **happy/1**. In all, **Prolog** tries three times to match **healthy(joan)** as there are three clauses for **healthy/1**. After failing **healthy(joan)**, however, **Prolog** does not try to solve **woman(joan)** —there is no point in doing so.

There is another way of trying to prove **happy(joan)** using the second clause of **happy/1**. The resolution process again generates subgoals —**wealthy(joan)** and **woman(joan)**— and **wealthy(joan)** fails. A third attempt is made but this founders as **wise(joan)** fails. Now back to top level to report the complete failure to satisfy the goal.

Now consider

happy(P)

as the top level goal.



Much more complicated. First, **healthy(P)** succeeds *binding* **P** to **jim** (*P/jim*) but when the conjunctive goal **woman(jim)** is attempted it fails. **Prolog** now *backtracks*[3]. It reverses along the path through the tree until it can find a place where there was an alternative solution.

[3]See chapter 5 for more details.

Of course, **Prolog** remembers to *unbind* any variables exactly at the places in the tree where they were bound.

In the example we are using we again try to resolve the goal **healthy(P)** — succeeding with **P** bound to **jane**. Now the conjunction can be satisfied as we have **woman(jane)**. Return to top level with **P** bound to **jane** to report success. What follows is what appears on the screen:

?- **happy(P).**

P=jane
yes

Prolog offers the facility to *redo* a goal —whenever the top level goal has succeeded and there is a variable binding. Just type ";" followed by RETURN —";" can be read as *or*. If possible, **Prolog** finds another solution. If this is repeated until there are no more solutions then we get the sequence of solutions:

jane
jean
jane
jean

It is worth trying to verify this.

Basically, trying to follow the behaviour of **Prolog** around the text of the program can be very messy. Seeing how **Prolog** might execute the search based on moving around the AND/OR tree is much more coherent *but* it requires some effort before getting the benefit.

3.4 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to load in a **Prolog** program.
You should be able to issue a legal **Prolog** query.
You should be able to generate successive solutions to a goal (provided that any exist).
You should be able to apply a depth-first search strategy to simulate the **Prolog** execution of a goal in relation to a simple program.
You should have an idea about the way in which **Prolog** uses matching.
You should be aware of the effects of *backtracking* when a goal fails.

Exercise 3.1 *Here is the first opportunity to try to follow the execution of some* **Prolog** *query. For each of these problems, the aim is to follow the execution for a number of different queries. Each query gives rise to a sequence of subgoals which either fail outright or succeed —possibly binding some variables.*

The answers should use a standard format which is illustrated.

Program Database
a(X):-
b(X,Y),
c(Y).
a(X):-
c(X).
b(1,2).
b(2,2).
b(3,3).
b(3,4).
c(2).
c(5).

Use the following format for your answer:

Subgoals	**Comment**	**Result**
a(5)	uses 1st clause	new subgoals
b(5,Y)	tries 1st clause	fails
b(5,Y)	tries 2nd clause	fails
b(5,Y)	tries 3rd clause	fails
a(5)	using 1st clause	fails
a(5)	uses 2nd clause	new subgoal
c(5)	tries 1st clause	fails
c(5)	tries 2nd clause	succeeds
a(5)	using 2nd clause	succeeds

Note that, if a variable is bound, then indicate with a phrase such as **with Y=2**.

Repeat for the following goals:

1. *a(1)*
2. *a(2)*
3. *a(3)*
4. *a(4)*

Exercise 3.2 *As in the previous exercise, for the new program:*

Program Database
a(X,Y):-
b(X,Y).
a(X,Y):-
c(X,Z),
a(Z,Y).
b(1,2).
b(2,3).
c(1,2).
c(1,4).
c(2,4).
c(3,4).

1. *a(1,X)*
2. *a(2,X)*
3. *a(3,X)*

4. *a(X,4)*
5. *a(1,3)*

Chapter 4

Unification, Recursion and Lists

We describe the matching process known as *Unification* that has already been met.
We review the basic idea of recursion as a programming technique.
We apply these ideas to list processing.

4.1 Unification

Unification is the name given to the way **Prolog** does its matching. We will not do more than sketch the basic ideas here. Basically, an attempt can be made to *unify* any pair of valid **Prolog** entities or *terms*.

Unification is more than simple matching. A naive view of the matching process might be represented by the question "can the target object be made to fit one of the source objects". The implicit assumption is that the source is not affected —only the target is coerced to make it look like some source object.

Unification implies mutual coercion. There is an attempt to alter both the target and the current source object to make them look the same.

Consider how we might match the *term* **book(waverley,X)** against some clause for which **book(Y, scott)** is the head. The naive approach might be that *X/scott* is the correct substitution —or even that the matching cannot be done. Unification provides the substitutions *X/scott* and *Y/waverley*. With these substitutions both terms look like **book(waverley,scott)**.

Unification is a two way matching process

The substitution *X/scott* and *Y/waverley* is known as a *unifier* —to be precise, the *most general unifier*. If we *unify* **X** with **Y** then one unifier might be the substitution *X/1* and *Y/1* but this is not the most general unifier.

Consider the infix predicate **=/2**.

Certain 'built-in' **Prolog** predicates are provided that can be written in a special infix or prefix form (there are no postfix ones provided —that is not because they could not be!) For example, **1=2** is written as **=(1,2)** in standard **Prolog** form.

Prolog tries to unify *both* the arguments of this predicate. Here are some possible unifications:

X=fred	succeeds
jane=fred	fails because you can't match two distinct atoms
Y=fred, X=Y	succeeds with **X=fred**, **Y=fred**
X=happy(jim)	succeeds
X=Y	succeeds —later, if **X** gets bound then so will **Y** and vice versa

It is worth making a distinction here between the *textual* name of a logical variable and its run-time name. Consider a query **likes(jim,X)**. Suppose there is one clause: **likes(X,fred)** —this has the reading that "everyone likes fred" and mentions a variable with the textual name of **X**. The query also mentions a specific variable by the textual name of **X**. By the *scope* rule for variables, we know that these two variables, although textually the same, are really different. So now consider whether the head of the clause **likes(X,fred)** unifies with the current goal **likes(jim,X)**.

We might then reason like this: the task is to decide whether or not **likes(jim,X)=likes(X,fred)** succeeds. If this is so then, matching the first arguments, we get X=jim. Then we try to match the second arguments. Now can **X=fred**? If X=jim then the answer is no. How is this? The answer we expect (logically) is that "jim likes fred". We really ought to distinguish every variable mentioned from each other *according to the scope rules*. This means that the query is better thought of as, say, **likes(jim,**$\mathbf{X}_1$ and the clause is then **likes(**$\mathbf{X}_2$**,fred)**. In the literature the process of making sure that variables with the same textual name but in different scopes are really different is known as *standardisation apart*!

Exercise 4.1 *Here are some problems for which unification sometimes succeeds and sometimes fails. Decide which is the case and, if the unification succeeds, write down the substitutions made.*

1. **2+1=3**
2. **f(X,a)=f(a,X)**
3. **fred=fred**
4. **likes(jane,X)=likes(X,jim)**
5. **f(X,Y)=f(P,P)**

4.2 Recursion

Recursion is a technique that must be learned as programming in **Prolog** depends heavily upon it.

We have already met a recursive definition in section 2.2. Here are some more:

One of my ancestors is one of my parents or one of their ancestors.

A string of characters is a single character or a single character followed by a string of characters.

A paragraph is a sentence or a sentence appended to a paragraph.

To decouple a train, uncouple the first carriage and then decouple the rest of the train.

An example recursive program:

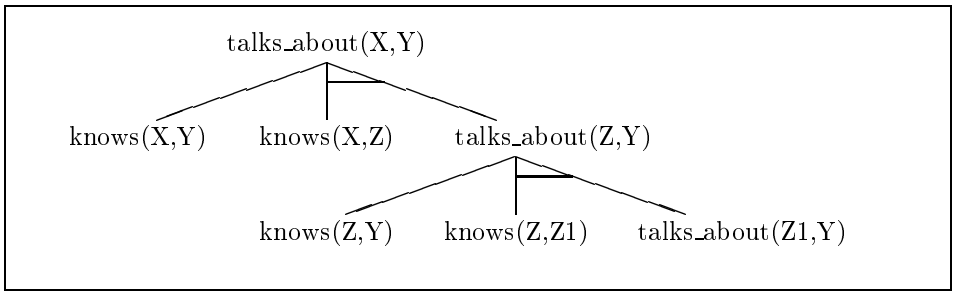
```
talks_about(A,B):-
        knows(A,B).
talks_about(P,R):-
        knows(P,Q),
        talks_about(Q,R).
```

Roughly translated:

> You talk about someone either if you know them or you know someone who talks about them

If you look at the AND/OR tree of the search space you can see that

- There is a subtree which is the same shape as the whole tree reflecting the single recursive call to **talks_about/2**.
- The solution of a given problem depends on being able to stop recursing at some point. Because the leftmost path down the tree is not infinite in length it is reasonable to hope for a solution.



In searching the tree with a number of facts along with the clauses for **talks_about/1**:

Program Database	
talks_about(A,B):-	
	knows(A,B).
talks_about(P,R):-	
	knows(P,Q),
	talks_about(Q,R).
knows(bill,jane).	
knows(jane,pat).	
knows(jane,fred).	
knows(fred,bill).	

using the goal

```
talks_about(X,Y)
```

If we ask for repeated solutions to this goal, we get, in the order shown:

X= bill Y= jane
X= jane Y= pat
X= jane Y= fred
X= fred Y= bill
X= bill Y= pat
and so on

The search strategy implies that **Prolog** keep on trying to satisfy the subgoal **knows(X,Y)** until there are no more solutions to this. **Prolog** then finds that, in the second clause for **talks_about/2**, it can satisfy the **talks_about(X,Y)** goal by first finding a third party who **X** knows. It satisfies **knows(X,Z)** with **X=bill**, **Z=jane** and then recurses looking for a solution to the goal **talks_about(jane,Z)**. It finds the solution by matching against the second **knows/2** clause.

The above AND/OR tree was formed by taking the top level goal and, for each clause with the same predicate name and arity, creating an OR choice leading to subgoals constructed from the bodies of the matched clauses. For each subgoal in a conjunction of subgoals we create an AND choice.

Note that we have picked up certain relationships holding between the (logical) variables but we have had to do some renaming to distinguish between attempts to solve subgoals of the form **talks_about(A,B)** recursively.

4.3 Lists

Lists, for now, can be regarded as special **Prolog** structures that can be used to represent an ordered sequence of **Prolog** terms. For example, here are some legal lists:

[ice_cream, coffee, chocolate]	a list with three elements (all atoms)
[a, b, c, c, d, e]	a list with six elements (all atoms)
[ ]	a list with no elements in it (it is an atom)
[dog(fido), cat(rufus), goldfish(jimmy)]	a list with three elements (all **Prolog** terms)
[happy(fred),[ice_cream,chocolate],[1,[2],3]]	a list with three elements!

The last example is a little difficult to decipher: the first element is **happy(fred)**, the second is **[ice_cream,chocolate]**, a list, and the third is **[1,[2],3]**, another list.

Note that the "," used in the construction of a list is *just* an argument separator as in the term **foo(a,b)**. Also note that, because order is preserved, the list **[a,b,c]** is *not* the same as **[a,c,b]**.

How to construct/deconstruct a list

Given an arbitrary list, we need ways of adding to it and taking it apart[1].

The basic approach provides a simple way of splitting a list into two bits: the first element (if there is one!) and the rest of the list. The corresponding way of joining two bits to form a list requires taking an element and a list and inserting the element at the front of the list.

[1] We also need ways of accessing an arbitrary element, but this can wait

List Destruction: first, we show how to remove the first element from a list.

[X|Y] = [f,r,e,d]

will result in

X=f

—the first element of the list is known as the **HEAD** of the list.

Y=[r,e,d]

—the list formed by deleting the head is the **TAIL** of the list. This list has been reduced in length and can be further destructed or constructed.

List Construction: the construction of a list is the reverse: take a variable bound to any old list —say, **X=[r, e, d]** and add the element, say, **b** at the front with:

Result_Wanted = [b|X]

Bigger Chunks: it is possible to add (or take away) bigger chunks onto (from) the front of a list than one element at a time. The list notation allows for this. Suppose you want to stick the elements **a**, **b** and **c** onto the front of the list **X** to make a new list **Y**. then this can be done with **Y=[a,b,c|X]**.

Conversely, suppose you want to take three elements off the front of a list **X** in such a way that the remaining list, **Y**, is available for use. This can be done with **X=[A,B,C|Y]**

> A limitation of this approach is that there is no direct way of evading specifying how many elements to attach/rip off. Using the list notation, there is no way of saying "rip off N elements of this list X and call the remainder Y". This has to be done by writing a program and since this is very straightforward, this limitation is not a severe one —but, see later.

The Empty List

Simply written

[ ]

This list ([ ]) has no elements in it: it cannot therefore be destructed. An attempt to do this will fail.

The empty list ([ ]) is an atom.

Some Possible Matches

We now illustrate how two lists unify and in what circumstances two lists fail to unify.

1.	**[b,a,d]=[d,a,b]**	fails —as the order matters
2.	**[X]=[b,a,d]**	fails —the two lists are of different lengths
3.	**[X|Y]=[he,is,a,cat]**	succeeds with **X=he**, **Y=[is,a,cat]**
4.	**[X,Y|Z]=[a,b,c,d]**	succeeds with **X=a**, **Y=b**, **Z=[c,d]**
5.	**[X|Y]=[]**	fails —the empty list can't be deconstructed
6.	**[X|Y]=[[a,[b,c]],d]**	succeeds with **X=[a,[b,c]]**, **Y=[d]**
7.	**[X|Y]=[a]**	succeeds with **X=a]**, **Y=[]**

Exercise 4.2 *Here are some more problems for which unification sometimes succeeds and sometimes fails. Decide which is the case and, if the unification succeeds, write down the substitutions made.*

1. **[a,b\|X]=[A,B,c]**
2. **[a,b]=[b,a]**
3. **[a\|[b,c]]=[a,b,c]**
4. **[a,[b,c]]=[a,b,c]**
5. **[a,X]=[X,b]**
6. **[a\|[]]=[X]**
7. **[a,b,X,c]=[A,B,Y]**
8. **[H\|T]=[[a,b],[c,d]]**
9. **[[X],Y]=[a,b]**

A Recursive Program Using Lists

We make use of a built-in predicate called **write/1** to write out all the elements of a list in order. Note that the argument of **write/1** must be a legal **Prolog** term.

> **write/1** is a *side-effecting* predicate. It captures the logical relation of always being true but it also produces output which has no part to play in the logical interpretation. It is therefore hard to produce a declarative reading for this predicate despite its utility from the procedural point of view. There are a fair number of other predicates which suffer from this problem including **consult/1** and **reconsult/1**.

To write out a list of terms, write out the first element and then write out the remainder (the tail).

```
print_a_list([]).
print_a_list([H|T]):-
                write(H),
                print_a_list(T).
```

Note that this can be improved by printing a space between elements of the list. This requires you to add the subgoal **write(' ')** into the body of the second clause and *before* the recursive call to **print_a_list/1**.

This will write the elements out on a single line. If you wanted to write each element on a different line then you would need the built-in predicate **nl/0**.

> The second clause of **print_a_list/1** roughly captures the meaning above. Then what does the first clause achieve? Without the first clause, **print_a_list/1** would produce the required output and then fail because it would have to handle the empty list ([]) which cannot be deconstructed. Although **print_a_list/1** is a *side-effecting* predicate, the natural (procedural) reading is that it succeeds once it has printed the list of terms. The first clause handles the case of the empty list so that the predicate will always succeed if it is given a list of terms to print. Quite reasonably, it will fail if given a non-list.

4.4 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to determine whether or not two **Prolog** terms *unify*.
You should be able to identify programs that are *recursive*.
You should be able to build and take apart list structures.
You should be able to write simple list processing programs using recursion.

Exercise 4.3 *For each of these problems, the aim is to define a predicate using one or two clauses. Each of the problems is a list processing problem.*

1. *Write a predicate* **print_every_second/1** *to print every other element in a list, beginning at the second element —i.e. the 2nd, 4th, 6th elements etc. It should always succeed provided it is given a list as its argument.*

2. *Write a predicate* **deconsonant/1** *to print any element of a list that isn't a consonant (i.e. we want to print out the vowels {a,e,i,o,u}). It should always succeed provided it is given a list as its argument (we assume that the input list only contains vowels and consonants).*

3. *Write a predicate* **head/2** *which takes a list as its first argument and returns the head of the list as its second argument. It should fail if there is no first element.*

4. *Write a predicate* **tail/2** *which takes a list as its first argument and returns the tail of the list as its second argument. It should fail if there is no first element.*

5. *Write a predicate* **vowels/2** *which takes a list as its first argument and returns a list (as its second argument) which consists of every element of the input list which is a vowel (we assume that the input list only contains vowels and consonants).*

6. *Write a predicate* **find_every_second/2** *which takes a list as its first argument and returns a list (as its second argument) which consists of every other element of the input list starting at the second element.*

   You should note that we have turned the *side-effecting* predicates of the first two problems above into predicates which do not make use of side-effects and can now be given a declarative reading.

Chapter 5

The Box Model of Execution

We describe the Byrd box model of **Prolog** execution.
We illustrate *backtracking* in relation to the Byrd box model of execution and then in relation to the AND/OR execution and proof trees.

5.1 The Box Model

As this model is a model of **Prolog** execution, we can think in terms of *procedures* rather than *predicates*.

We represent each call to a procedure by a box. Note that, as a procedure may be executed thousands of times in a program, we need to distinguish between all these different invocations. In the diagram in figure 5.1 a box represents the invocation of a single procedure and which is therefore associated with a specific goal. The top level query is **parent(X,Y), X=f**.

We regard each box as having four ports: they are named the **Call**, **Exit**, **Fail** and **Redo** ports. The labelled arrows indicate the control flow in and out of a box via the ports. The **Call** port for an invocation of a procedure represents

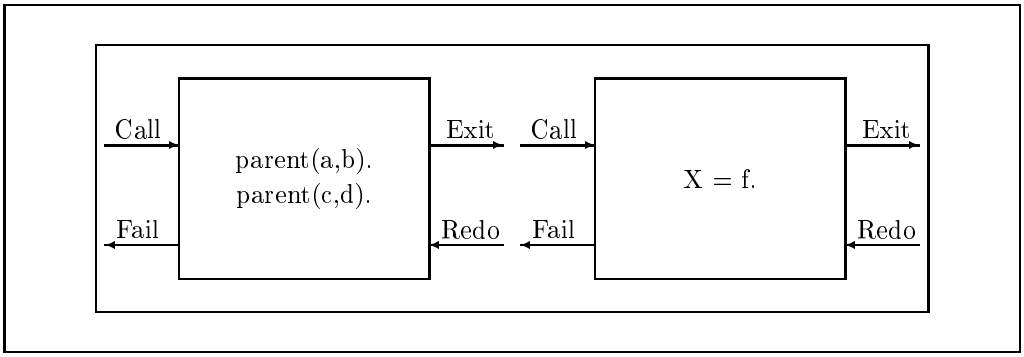


Figure 5.1: The Byrd Box Model Illustrated

the first time the solution of the associated goal is sought. Control then 'flows' into the box through the **Call** port.

We then seek a clause with a **head** that unifies with the goal. Then, we seek solutions to all the subgoals in the body of the successful clause.

If the unification fails for all clauses (or there are no clauses at all) then control would pass out of the **Fail** port. There are also other ways to reach the **Fail** port.

Control reaches the **Exit** port if the procedure succeeds. This can only occur if the initial goal has been unified with the head of one of the procedure's clauses and all of its subgoals have been satisfied.

The **Redo** port can only be reached if the procedure call has been successful and some subsequent goal has failed. This is when **Prolog** is *backtracking* to find some alternative way of solving some top-level goal.

Basically, *backtracking* is the way **Prolog** attempts to find another solution for each procedure that has contributed to the execution up to the point where some procedure fails. This is done back from the failing procedure to the first procedure that can contribute an alternative solution —hence, *backtracking*.

When backtracking is taking place, control passes through the **Redo** port. We then, with the clause which was used when the procedure was previously successful, backtrack further back through the subgoals that were previously satisfied. We can reach the **Exit** port again if either one of these subgoals succeeds a different way —and this leads to all the subgoals in the body of the clause succeeding— or, failing that, another clause can be used successfully. Otherwise, we reach the **Fail** port. Note that, for this to work out, the system has to remember the clause last used for each successful predicate.

> The system can throw this information away only if it can convince itself that we will never revisit a procedure that succeeds. We can always force this to happen by using the cut (!/0) (which is explained in chapter 9) —but this is a last resort as most implementations of **Prolog** can do some sensible storage management. An understanding of this mechanism can help you avoid the use of cut.

We reach the **Fail** port

- When we cannot find any clauses such that their heads match with the goal
- If, on the original invocation, we can find no solution for the procedure
- On backtracking, we enter the box via the **Redo** port but no further solution can be found

5.2 The Flow of Control

We illustrate the above with a textual representation of the simple program found in figure 5.1 using the Byrd box model. The flow of control is found in figure 5.2. The indentation is used here only to suggest an intermediate stage in the mapping from the visual representation of the boxes into their textual sequence.

> Many **Prolog** trace packages that use this box model do no indenting at all and those that use indentation use it to represent the 'depth' of processing. This depth is equivalent to the number of arcs needed to go from the root of the AND/OR execution tree to the current node.

Below, we have a snapshot of how the execution takes place —"taken" at the moment when **Prolog** *backtrack*s to find another solution to the goal **parent(X,Y)**. We show the backtracking for the same program using an AND/OR execution tree.

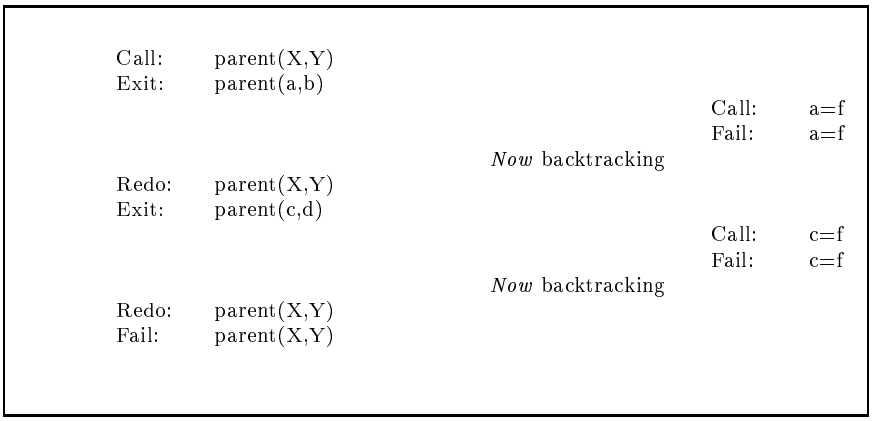
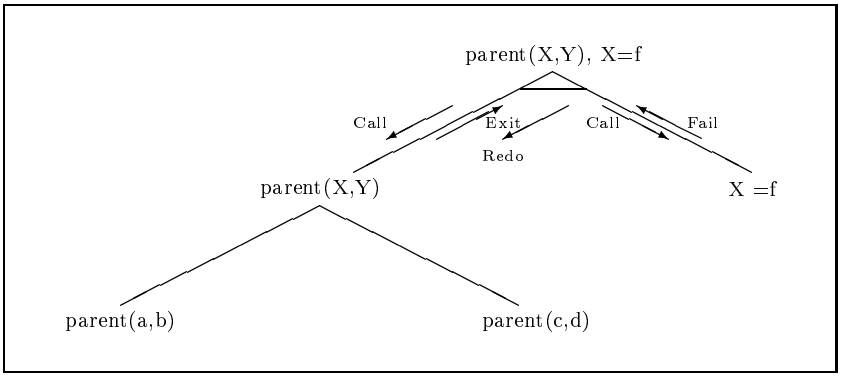


Figure 5.2: Illustrating Simple Flow of Control



5.3 An Example using the Byrd Box Model

We use a simple program with no obvious natural interpretation to contrast the Byrd box model with the AND/OR execution tree. See figure 5.3 for the program and for a graphical representation of the program's structure using the Byrd box model. Figure 5.4 shows the same program's structure as an AND/OR tree.

We consider how the goal **a(X,Y)** is solved.

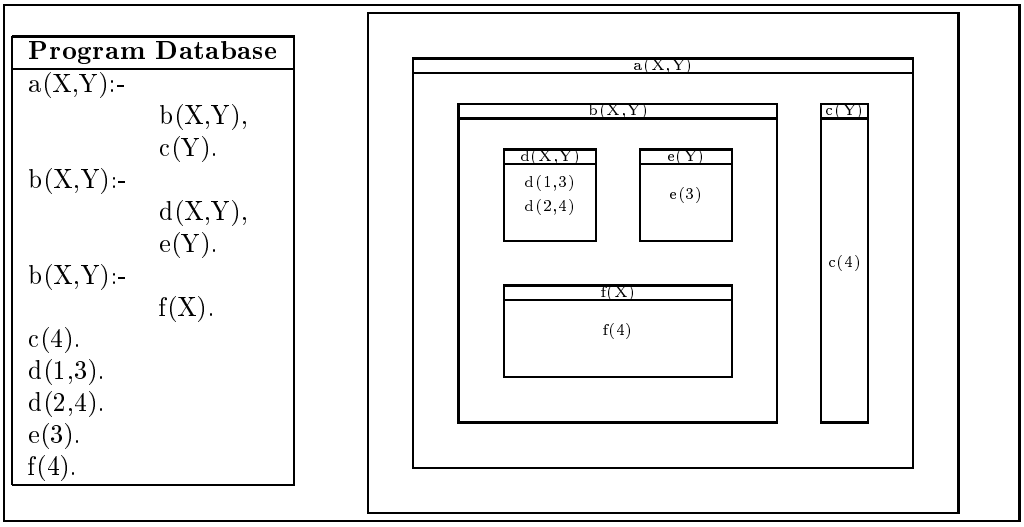
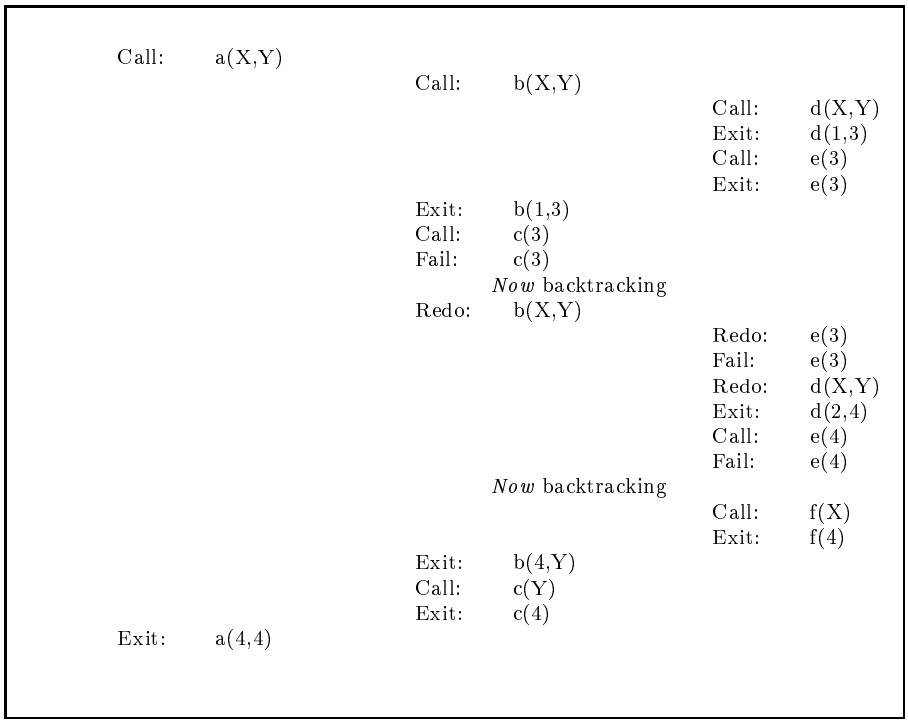


Figure 5.3: Program Example with Byrd Box Representation



5.4 An Example using an AND/OR Proof Tree

We now use the same example program to show how the proof tree grows. We choose a proof tree because we can delete any parts of the tree which do not contribute to the final solution (which is not the case for the execution tree).

The search space as an AND/OR tree is shown in figure 5.4. We now develop the AND/OR proof tree for the same goal. We show ten stages in order in figure 5.5. The order of the stages is indicated by the number marked in the top left hand corner.

The various variable bindings —both those made and unmade— have not been represented on this diagram.

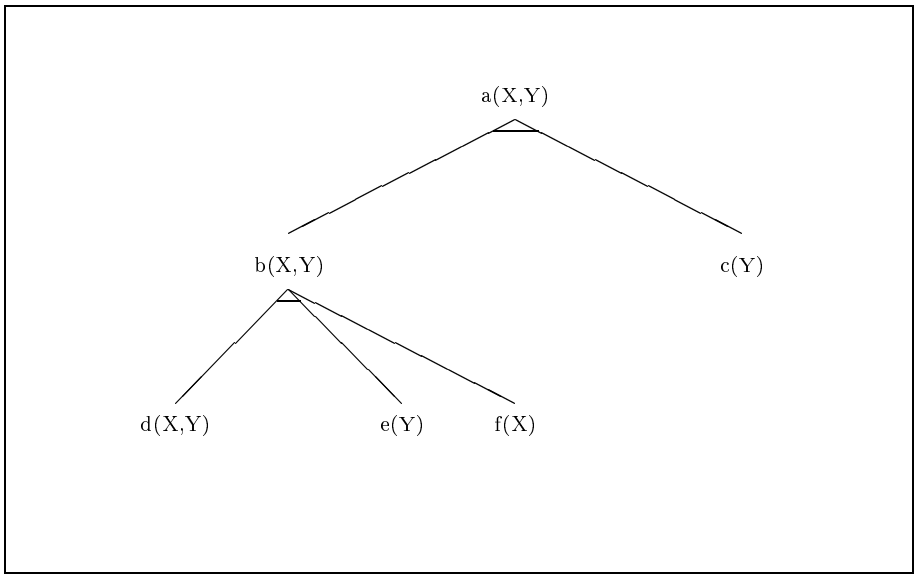


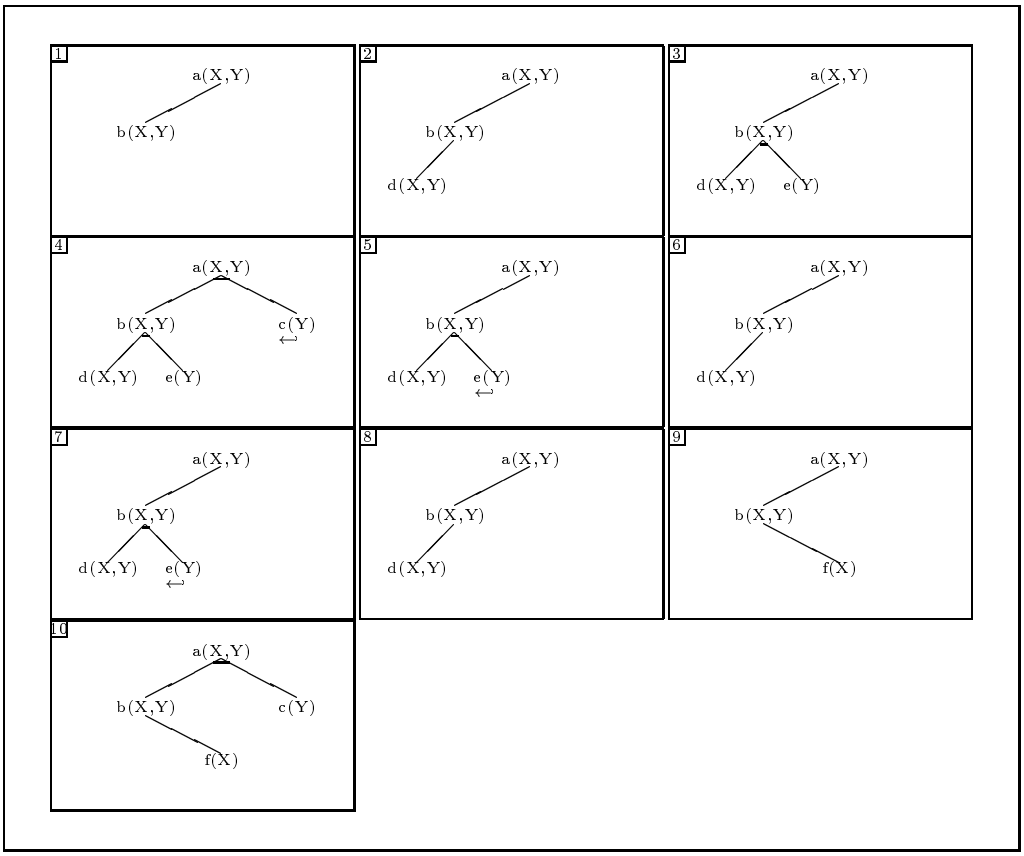
Figure 5.4: The AND/OR Tree for the Goal **a(X,Y)**

5.5 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to describe the execution of simple programs in terms of the Byrd box model. You should be able to follow backtracking programs in terms of the Byrd box model. You should also be construct the AND/OR execution and proof trees for programs that backtrack.

Exercise 5.1 *We use the same two programs as found at the end of chapter 3. For each of these problems, the aim is to predict the execution first using the development of the AND/OR proof tree and then using the Byrd box model for each of the different queries.*



Note that ↩ indicates the start of backtracking.

Figure 5.5: The Development of the AND/OR Proof Tree

1. *Predict the execution behaviour —developing the AND/OR proof tree and then using the Byrd box model— for the following goals:*

   (a) *a(1)*
   (b) *a(2)*
   (c) *a(3)*
   (d) *a(4)*

Program Database
a(X):-
b(X,Y),
c(Y).
a(X):-
c(X).
b(1,2).
b(2,2).
b(3,3).
b(3,4).
c(2).
c(5).

2. *As in the previous exercise, for the new program:*

(a) *a(1,X)*
(b) *a(2,X)*
(c) *a(3,X)*
(d) *a(X,4)*
(e) *a(1,3)*

Program Database

```
a(X,Y):-
        b(X,Y).
a(X,Y):-
        c(X,Z),
        a(Z,Y).
b(1,2).
b(2,3).
c(1,2).
c(1,4).
c(2,4).
c(3,4).
```

Interlude: Practical Matters

We describe some matters relating to the practical use of **Prolog**.
We show how to invoke **Prolog** and also how to exit.
We describe how to load programs as files
We show how to develop a program and avoid some common errors.
We outline the Input/Output features provided by **Prolog**.
We then illustrate the use of the debugger and provide some information about the debugging process.

Exiting and Leaving Prolog

The **Prolog** system you will be using is known as SICStus **Prolog** and you are using it within the UNIX environment (DYNIX(R) V3.0.17.9) provided on a Sequent computer. All that follows is intended for this context only.

Prolog is entered with the command:

unix prompt: **prolog**[1]

The most reliable way to exit **Prolog** is with the command:

| ?- **halt.**

Note that the prompt is really | **?-** for this **Prolog**. For simplicity, we have assumed this is **?-** in the main body of the notes.

In passing, there is a small problem associated with pressing the **Return** key before typing the '.'. This is what happens:

```
| ?- halt
|:
```

Prolog is still waiting for the '.'. All you have to do is type in the '.' and then press **Return**.

This is guaranteed to work but the other ways can fail depending on circumstances. Here are two other ways of exiting **Prolog** from the top level.

[1] This is supposed to produce a banner a variant on
`SICStus 0.6 #11: Tue Jul 3 15:40:37 BST 1990`
If the system produces some other system, contact the course organiser.

```
| ?- ^D
| ?- end_of_file.
```

Note that **^D** is the keyboard character obtained by holding down the **Control** key, pressing the **Shift** key and then the **d** key. This character is the default character to signal that the end of a file has been encountered. It can be changed.

> The reason why these last two ways are not general depends on one of the sophisticated features of the **Prolog** system: viz., that the command **break** initiates a new incarnation of the **Prolog** interpreter. Repeated **break**s will generate further levels. The command **halt** exits **Prolog** from any level while the above two commands only exit one level at a time and only exit **Prolog** if at top level.

Loading Files

A program should normally be considered as a sequence of files. Consequently, it is usually necessary for **Prolog** to read in one or more files at the beginning of a session.

The standard command is

```
| ?- consult(filename).
```

where "filename" is some legal unix filename. Because some legal unix filenames contain characters that **Prolog** may find syntactically illegal it is often necessary to 'protect' the filename using single quotes. Here are some arbitrary examples:

```
| ?- consult(foo).
| ?- consult('/u/ai/s2/ai2/aifoo/program').
| ?- consult('foo.pl').
```

It is also possible to consult a set of files as in:

```
| ?- consult([foo,baz,'foobaz.pl']).
```

> There is a shorthand for the command **consult** which can be confusing. The abbreviation overloads the symbols associated with list notation. The command **consult(foo)** can be abbreviated to [**foo**] and the command **consult([foo,baz])** can be rewritten [**foo,baz**]. There is quite a subtle syntax error that can cause difficulties when the file you want to read in needs to be protected with single quotes. Consider:
>
> ```
> | ?- ['foo.pl].
> |:
> ```
>
> **Prolog** is still waiting for the closing single quote. All you have to do is type in the closing single quote and then the **].** and press **Return**. **Prolog** will produce an error message because you have asked to load a *very* strangely named file.
>
> Another error is to use double quotes instead of single quotes.

```
| ?- ["foo.pl"].

{ERROR: absolute_file_name(102,_45) - invalid file spec}
```

This weird error will not be explained here —just note that double quotes have a special interpretation in **Prolog** which results in the above command being interpreted as the desire to consult three files: 102, 111 and 111. Can you guess the meaning of double quotes?

Each syntactically correct clause that is found on reading the file will be loaded. On encountering a syntactically incorrect clause then an error message will be printed. We now illustrate some common syntax errors together with the error messages generated. You will notice that the error messages can be quite obscure.

```
foo (X).            % space between functor and left bracket

* bracket follows expression **
foo
* here **
( X ) .


fooX).              % missing left bracket

* operator expected after expression **
fooX
* here **


foo(X.              % missing right bracket

* , or ) expected in arguments **
foo ( X
* here **


foo(X Y).           % missing argument separator

* variable follows expression **
foo ( X
* here **


foo([a,b).          % missing right square bracket

* , | or ] expected in list **
foo ( [ a , b
* here **
```

```
foo(a)                  % missing '.'
foo(b).

* atom follows expression **
foo ( a )
* here **
foo ( b ) .

foo(a),                 % used ',' for '.'
foo(b).

{ERROR: (,)/2 - attempt to redefine built_in predicate}
```

This latter error message is caused because the input is equivalent the the logical statement **foo(a)** $\wedge$ **foo(b)** which is not in Horn clause form and therefore not legal **Prolog**. Here is another related error:

```
foo;- baz.              % ; instead of :
{ERROR: (;)/2 - attempt to redefine built_in predicate}
```

We suggest that, if you have made a syntax error and pressed **Return** (so you cannot delete the error) then type in '.' followed by **Return**. This will probably generate a syntax error and you can try again. Of course, there are situations for which this will not work: you cannot use this method to get out of the problem with:

```
| ?- ['foo.pl].
|:
```

or the equivalent problem with double quotes.

Now SICStus does one nice thing: **consult(foo)** will first try to find a file "foo.pl". If it does not find one, it will look for "foo".

Interactive Program Development

We want to be able to develop a program interactively. This suggests that we will edit our program using one of the editors provided (such as vi, ex, gnu emacs or microemacs), enter **Prolog**, load our program, find a bug, exit **Prolog** and repeat.

This is clumsy, so we describe two methods that should aid interactive program development. In both cases, however, we must be aware of a problem in connection with **consult/1**.

A Problem with consult/1

Consider the query:

| ?- **consult([foo1,foo2]).**

where both **foo1** and **foo2** contain clauses for, say, **baz/1**. We get the following:

The procedure baz/2 is being redefined.
Old file: /u/user5/ai/staff/paulb/foo1.pl
New file:/u/user5/ai/staff/paulb/foo2.pl
Do you really want to redefine it? (y, n, p, or ?) ?

Therefore, as far as is possible, avoid splitting your predicate definitions between files.

> The command **reconsult(foo)** is equivalent to **consult(foo)**. The command **reconsult(foo)** can be rewritten as [**-foo**] and the command **reconsult([foo1,foo2])** can be rewritten as [**-foo1,-foo2**].
>
> Some **Prolog** systems distinguish these commands. For these systems, the command **consult([foo1,foo2])** has the consequence of loading the syntactically correct clauses found both in **foo1** and in **foo2** —if they share the definition of **baz/2** then both parts of the definition will be loaded.
>
> Finally, if you really have to distribute your predicate definitions between files with a command like **consult([foo1,foo2])** then there must be a *declaration* that the predicate is a *multifile* predicate *before* SICStus encounters the first clause. So, if **baz/2** is shared between files, we need to place
>
> :- multifile baz/2.
>
> before the first clause for **baz/2**.
>
> Even though mostly you won't need to do this, there are occasions when it does make sense to distribute a predicate across several files.

Two Variations on Program Development

The first variation depends on whether or not you are using a unix shell that allows for job suspension. This can be checked by getting into **Prolog** and issuing the character **^Z** which is the usual default for suspending a job. You then find yourself at the unix level where you can edit your file in the normal way. When you have finished editing, get back into **Prolog** with the command:

unix prompt: **fg**

which stands for bringing a suspended job into the foreground. Now you are back in **Prolog** and you have to reload your program using **consult/1**[2].

The second, more satisfactory variation depends on defining a predicate which can be used to edit a file without explicitly leaving **Prolog**. This can be done because there is a built-in predicate **shell/1** which takes as its argument a unix command as a list of the ASCII codes associated with the characters forming the command. Here is a simple program that, if loaded, can be used to edit a file and automatically reconsult it after the edit is finished.

[2]In SICStus anyway —if you are using a **Prolog** system that distinguishes between **consult/1** and **reconsult/1** then you *must* use **reconsult/1** or you can get into trouble.

Program Database

```
edit(File):-
                        editor(Editor),
                        name(Editor,EditorList),
                        name(File, FileList),
                        append(EditorList,[32|FileList],CommandList),
                        name(Command,CommandList),
                        unix(shell(Command)),
                        reconsult(File).

editor(emacs).

append([],L,L).
append([H|L1],L2,[H|L3]):-
                        append(L1,L2,L3).
```

Now you have to remember to load this each time you enter **Prolog**. One way this can be done is by having a file called **prolog.ini** in your home directory. This file will then be automatically consulted each time you enter **Prolog**. Put the above program in such a file and try it out. Note also that you can change the editor of your choice by redefining **editor/1**. The predicate **append/3** is very useful: it 'glues' two lists together —*e.g.* the goal **append([a,b],[c,d],X)** results in **X=[a,b,c,d]**. It is so useful that you will probably want it around all the time.

Avoiding Various Kinds of Trouble

There is a problem connected with a missing predicate definition. In SICStus Prolog, the default behaviour is to place you into the tracer. This is roughly what happens:

```
{Warning: The predicate foo/1 is undefined}
1        1     Fail: foo(_22) ?
```

Sometimes, however, we simply want to assume that if a call is made to a missing predicate then this is equivalent to not being able to solve the goal and the call therefore fails. This is connected with the *closed world assumption* which is outlined in chapter 7.

One way in which this can be controlled is to declare that the predicate, say **foo/1** is *dynamic* with the declaration:

```
?- dynamic foo/1.
```

This has the effect that, if there are no clauses for a dynamic predicate then the program will quietly fail.

A 'missing' predicate can be caused in a number of ways which will now be listed.

- A file that should have been loaded has not been loaded
- A subgoal has been misspelled —*e.g.* a call to **f00** instead of to **foo**.
- The name of a predicate has been misspelled in all the clauses of the definition. —*e.g.* the call is to **foo** but every definition is for **foo0**.
- A subgoal has the wrong number of arguments —*e.g.* there is a call **foo(1)** when the definition for **foo** has two arguments.

- The definition for a predicate consistently has the wrong number of arguments.

- Finally, you just may have really forgotten to define some predicate.

One way of dealing with all of these —even if it is hard to locate the cause— is to set a system flag to regard every call to an undefined predicate as some sort of error and to invoke the tracer. This is exactly what SICStus does. If you are using some other **Prolog** system that does not have this default behaviour, it may allow for you to use the following (perhapse even in your **prolog.ini** file):

```
?- unknown(X,trace).
```

The call **unknown(X,trace)** will change the behaviour from whatever the current setting is to 'trace' (the only other behaviour is 'fail'). To find the current setting without changing it you can use the query **unknown(X,X)** (SICSTUS can be reset to quietly fail with the command **unknown(X,fail)**).

Another problem can be caused by misspelling variables. For example, the definition:

```
mammal(Animal):-
        dog(Aminal).
```

probably features a misspelled variable. However, SICStus version 0.6 does not report such a definition. Some other **Prolog** systems, such as Edinburgh **Prolog**, provide something akin to:

```
Warning: singleton variable Animal in procedure mammal/1
Warning: singleton variable Aminal in procedure mammal/1
```

A *singleton variable* occurs in a clause if a variable is mentioned once and once only. Such a variable can never contribute a binding to the final result of the computation. Even though there are occasions when this does not matter, a singleton variable is an indication that there might be a misspelling.

> Consider the clause **member(X,[X|Y]).** This has a legitimate singleton variable, **Y**. If you need to mention a singleton variable, then you can use the *anonymous variable*. This is a special symbol for a variable for which you don't want to know any binding made. It is written as an underscore (_). Consequently, the above clause becomes **member(X,[X|_]).**
>
> This is fair enough and there will be no warning given when the clause is read in. It is, however, good practice to give meaningful names to variables —as much for program maintenance as for any other reason. The way round this can be achieved with a variable that begins with an underscore (_). For example, the above clause could be rewritten as **member(X,[X|_Tail]).** The *anonymous variable* is also described in section 10.2.

Input/Output Facilities

We now mention, in passing, some of the I/O facilities built into **Prolog**. We have already met a way of inputting multiple clauses via **consult/1** (and **reconsult/1**). We have already met predicates that produce output —**write/1** and **nl/0**— in chapter 4. Much more information can be found in chapter 10.10.

For now, we will not show how to output to a file —see chapter 10.10 for the details. In passing, we mention that a single **Prolog** term can be read in using **read/1**. Input using this predicate must be terminated by the standard '.' followed by *white space*.

Here are some low level I/O predicates:

get0(X)	unifies **X** with next non blank printable character (in ASCII code) from current input stream
get(X)	unifies **X** with next character (in ASCII) from current input stream
put(X)	puts a character on to the current output stream. **X** must be bound to a legal ASCII code

Note that they do not have a declarative reading. They fit poorly into the theoretical structure underlying **Prolog** —but other languages suffer from this problem (*e.g.* ML).

The Debugging Issue

Once we have loaded our syntactically correct program and tried it out we may realise that things aren't the way we want. We may come to realise that we did not (informally) specify the problem correctly or that we must have coded the specification wrongly.

We may come to realise that we have an error in our code through executing some query which produces an unexpected result. We regard such evidence as a *symptom* description. The kinds of *symptom* description that may result include:

- (apparent) non-termination
- unexpected **Prolog** error message(s)
- unexpected failure (or unexpected success)
- wrong binding of one of the variables in the original query

  There is also the possibility of unexpected side-effects (or an unexpected failure to produce a side-effect).

Different strategies exist for pinning down the cause(s) of these *symptom*s. We will not give a complete account here —just sketch in ways in which the tracer can be used.

The idea of using the tracer is to unpack the program's response to the query which produced a *symptom* description. This is done by examining the program's behaviour in the hope that we can track down subcomponents which

'misbehave'. Hence we search for a *program misbehaviour* description. Once this has been found we then need to track the fault to an error in the code and generate a *program code error* description. Finally, underlying the error in the code may be a number of misunderstandings about the way **Prolog** executes a program, the generality of the code written and so on. Tracking this down would produce a *misconception* description.

The Tracer Outlined

The description of the tracer's features that follows is intentionally brief. A more complete account can be found in appendix B. Note that the tracer uses the Byrd box model.

Full tracing only applies to non-compiled (*i.e.* interpreted) code but some limited tracing can be done for compiled code. The behaviour is similar to the treatment of system predicates.

Activating the Tracer

First, we outline the facilities for altering the behaviour of the system with regard to the tracer.

spy(predicate_name) Mark any clause with the given predicate_name as "spyable". Does not work for built-in predicates.

debug If a spied predicate is encountered, switch on the tracer.

nodebug Remove all spypoints. The tracer will therefore not be invoked.

nospy(predicate_name) Undo the effect of spy —*i.e.* remove the spy point.

debugging Shows which predicates are marked for spying plus some other information.

trace Switches on the tracer.

notrace Switches the tracer off. Does not remove spypoints.

Note that both **spy/1** and **nospy/1** can also take a list of predicates for their argument. The predicates can also be specified as, for example, **foo/1**. This allows for the distinction between (distinct) definitions for two or more predicates all with different arities.

There is also the concept of **leash**ing. The tracer provides for the possibility of various decisions to be made by the user at each of the four ports. There is also a facility for stopping interactions at the ports. This is done via **leash/1**. This predicate can take one of five arguments: **full**, **tight**, **half**, **loose** and **off**.

Argument	Consequence
full	Call, Exit, Redo and Fail interactive
tight	Call, Redo and Fail interactive
half	Call and Redo interactive
loose	Call interactive

The default is **full**.

The system is set up to default to full leashing: to change this, You can set your system up using the **prolog.ini** file by putting a line such as **?-leash([call,exit]).** in it.

Note that the ports of spy-points are always leashed (and cannot be unleashed).

Interacting with the Tracer

Now we outline the actions that the user can take at one of the interactive ports. In all, there are about 22 different actions that can be taken. We will describe a useful subset of 6 commands.

creep This is the single stepping command. Use **Return** to **creep**. The tracer will move on to the next port. If this is interactive then the user is queried —otherwise, the tracer prints out the results for the port and moves on.

skip This moves from the **Call** or **Redo** ports to the **Exit** or **Fail** ports. If one of the subgoals has a spypoint then the tracer will ignore it.

leap Go from the current port to the next port of a spied predicate.

retry Go from the current **Fail** or **Exit** port back to the **Redo** or **Call** port of the current goal —*i.e.* replay the execution over again.

unify This provides for the user giving a solution to the goal from the terminal rather than executing the goal. This is available at the **Call** port. This is of use in running programs which are incomplete (providing a form of "stub" for a predicate that has not yet been written). Enter a term that should unify with the current goal.

(re)set subterm This provides the facility to examine a subterm of a complex term. This provides a means for focussing on the part of the datastructure which is of interest. Consider the display at the **Call** port.

```
1        1    Call:foo(a(1,baz),[q,w,e,r,t])?
```

By selecting the **set subterm** option with **^ 1** we would see

```
1        1    Call:^ 1 a(1,baz)?
```

Then we can further select with **^ 2** :

```
1        1    Call:^ 2 baz?
```

To go back to the parent of a term requires the **reset subterm** command (**^**).

Debugging

We now sketch a simple strategy for using the tracer which copes with several of the *symptom*s described above. First, we handle *(apparent) non-termination.*

> There may be several reasons why a program appears not to terminate. These include factors outside of **Prolog** —*e.g.* the system is down, the terminal screen is frozen and the keyboard is dead. Another factor might be a 'bug' in the **Prolog** system itself. We have four more possibilities: some built-in predicate may not have terminated (*e.g.* you are trying to satisfy a **read/1** goal but not terminated input properly), you may accidently be writing to a file instead of to the terminal, the program might just be extremely inefficient or, finally, the program is never going to terminate —real non-termination— but it is hard to be sure of this!

During the execution of the goal:

^C	Raise an interrupt
t Return	Switch on tracing
^C	If no trace output, raise another interrupt
Return or ...	**creep** (or some other choice)

If the trace reveals a sequence of repeated, identical subgoals then this suggests that the program will not terminate.

Now, we look at a top-down way of debugging a program for terminating programs. The idea is to examine a goal by looking at each of its subgoals in turn until an error is detected. The subgoal containing the error is then explored in the same way. The basic schema is to

trace,goal.	Turn on the tracer and issue the goal
s	**skip** over each subgoal
r	If an incorrect result is detected, **redo** the last subgoal
Return	**creep**
repeat ...	Repeat the process for the new set of subgoals

All we suggest is that you examine whether or not a goal succeeds (or fails) when it should, whether or not it binds (or does not bind) those variables which you expect, and whether the bindings are the ones you intended.

We illustrate with a simple program found in figure 5.6. If the predicate **for_yuppies/1** is taken to mean "a country is suitable for yuppies to live in if it is near Austria and wealthy" then we might intend that the query **for_yuppies(austria)** should succeed —but it does not. We make sure that **leash(full)** (the default), turn on the tracer with **trace** and then issue the goal **for_yuppies(austria)**. Using the box model, we should get (in a simpler form than that produced by most tracers):

Call:	for_yuppies(austria) ? creep
Call:	near_austria(austria) ? skip
Fail:	near_austria(austria) ? retry
Call:	near_austria(austria) ? creep
Call:	country(austria) ? skip
Exit:	country(austria) ? creep
Call:	neighbouring(austria,austria) ? skip
Fail:	neighbouring(austria,austria) ?

At this point we know that there is no clause for **neighbouring(austria,austria)** and we can change the program.

Program Database	
for_yuppies(X):-	
	near_austria(X),
	rich(X).
near_austria(X):-	
	country(X),
	neighbouring(X,austria).
country(austria).	
country(switzerland).	
country(england).	
country(france).	
country(west_germany).	
neighbouring(switzerland,austria).	
neighbouring(west_germany,austria).	
neighbouring(leichtenstein,austria).	
neighbouring(czechoslovakia,austria).	
rich(X):-	
	average_income(X,Y),
	loadsamoney(Y).
average_income(austria,10000).	
average_income(switzerland,20000).	
average_income(czechoslovakia,5000).	
loadsamoney(X):-	
	X>8000.

Figure 5.6: Yuppies on the Move

Note that this is what the Byrd box model predicts, what SICSTUS does but *not* what Edinburgh **Prolog** produces. Consequently this strategy, although eminently sensible, will not work well for Edinburgh **Prolog**.

Chapter 6

Programming Techniques and List Processing

We introduce the idea of *calling patterns* —the ways in which a predicate may be used.
We then present some standard schemata for list processing.
We then apply these ideas to the construction of a simple-minded dialogue handler.

6.1 The 'Reversibility' of Prolog Programs

Consider the program:

Program Database
square(1,1).
square(2,4).
square(3,9).
square(4,16).
square(5,25).
square(6,36).

This has the reading that the second argument is the square of the first argument. There are four kinds of query: we can ask what is the square of a specific number, what number has a specific square and what entities are related by the square relation. We can also ask whether two specific numbers are in the relation to each other of one being the square of the other. The queries would look like this:

?- **square(2,X).**
?- **square(X,5).**
?- **square(X,Y).**
?- **square(2,3).**

Unlike many other programming languages, we do not need different procedures to calculate each of these results. This is a consequence of the declarative reading of **Prolog**. Sometimes we say that the program for **square/2** is *reversible*.

This is a very desirable property for programs. For example, if we could write a program to determine that a given string of words was a legitimate sentence then

we could use the same program to generate arbitrary grammatical sentences. Unfortunately, it not always possible to give a declarative reading to a **Prolog** program.

6.1.1 Evaluation in Prolog

Unlike many programming languages, **Prolog** does not automatically evaluate 'expressions'. For example, in Pascal,

```
Y := 2 + 1;
```

the term **2 + 1** is automatically evaluated and **Y** is assigned the value 3. Here is an attempt to do 'the same thing' in **Prolog** using **=/2**:

```
Y = 2 + 1.
```

with the consequence that the term **2+1** is unevaluated and the term **Y** is unified with the term **2+1** with the result that **Y** is bound to **2+1**.

> Similar problems arise in relation to LISP. LISP will generally seek to evaluate expressions. For example, in
>
> ```
> (foo (+ 1 2) 3)
> ```
>
> LISP evaluates the term (s-expression) **(foo (+ 1 2) 3)** by evaluating **(+ 1 2)** to **3** and then evaluating **(foo 3 3)**. A naive attempt to construct a similar expression in **Prolog** might look like:
>
> ```
> foo(1+2,3)
> ```
>
> but **Prolog** does *not* try to evaluate the term **1+2**.

Of course, there are times when evaluation is exactly what is wanted. Sometimes, particularly with arithmetic expressions, we want to evaluate them. A special predicate **is/2** is provided. This predicate can be used as in:

```
Y is 2 + 1.
```

In this case, the term **2+1** is evaluated to **3** and **Y** is unified with this term resulting in **Y** being bound to **3**.

We can use **is/2** to implement a successor relation:

```
successor(X,Y):-
                Y is X + 1.
```

where it is intended that **successor/2** takes the first argument as input and outputs the second argument which is to be the next largest integer.

In the above, note that **X + 1** is intended to be *evaluated*.

This means that you must use the stated calling pattern as to try to solve the goal **successor(X,7)** will lead to trying to evaluate **X + 1** with **X** *unbound*. This cannot be done. The result is an error message and the goal fails.

Consider the query

?- **3 is X+1.**

This results in a failure and an error message.

*** Error: uninstantiated variable in arithmetic expression:

Yet the logical statement that we might associate with the query is

∃ **X 3 is_one_more_than X**

This requires that we can search for the integer that, when added to 1 gives 3. Quite reasonable, but the arithmetic evaluator used is non-reversible. So the evaluation of arithmetic expressions is a one-way process.

Therefore **is/2** must always be called with its second argument as an arithmetic expression which has any variables already bound. So **successor/2** is not 'reversible'. For these queries,

1. **successor(3,X).**
2. **successor(X,4).**
3. **successor(X,Y).**
4. **successor(3,5).**

The 1st and 4th goals result in correct results (success and failure respectively) while the 2nd and 3rd goals produce error messages and fail.

6.2 Calling Patterns

For any given predicate with arity greater than 0, each argument may be intended to have one of three calling patterns:

- Input —indicated with a +
- Output —indicated with a -
- Indeterminate —indicated with a ? (+ or -)

For example, **successor/2** above requires a calling pattern of

1st argument must be +
2nd argument can be + or - and is therefore ?

We write this as

mode successor(+,?).

The notation used here is consistent with the *mode declarations* found in many **Prolog** libraries. For a further example, the mode declaration of **is/2** is **mode is(?,+).**

Because of the discrepancy between the declarative and the procedural aspects of **Prolog** we often need to think carefully about the intended usage of a predicate. It is good practice to comment your code to indicate a predicate's intended usage.

6.3 List Processing

Many programs will be easiest to write if lists are used as the basic data structure. Therefore, we will need to process lists in a number of different ways. We are going to look at four different kinds of task and then loosely describe the schemata which can be utilised.

6.3.1 Program Patterns

One way in which experienced **Prolog** programmers differ from beginners is that they have picked up a wide variety of implementation techniques from their previous programming experience and are able to bring this to bear on new problems. Here, we consider four schemata for handling a large number of list processing tasks. This not intended to cover all possible list processing programs. Rather, the intention is to give some guidance about how to think about the problem of constructing a program.

Test for Existence

We want to determine that some collection of objects has at least one object with a desired property. For example, that a list of terms has at least one term which is also a list. Here is the general schema:

list_existence_test(**Info**,[Head|Tail]):-
    element_has_property(**Info**,Head).
list_existence_test(**Info**,[Head|Tail]):-
    list_existence_test(**Info**,Tail).

The expression **Info** stands for a specific number of arguments (including zero) that carry information needed for the determination that a single element has the desired property. The arguments represented by **Info** are *parameters* while the remaining argument is the *recursion argument*. The functors in *italics* are in *italics* to indicate that these can be replaced by 'real' functors.

We outline two examples. The first has 0 parameters. We test whether a list contains lists using **nested_list/1**—*e.g.* we want the goal **nested_list([a,[b],c])** to succeed.

```
nested_list([Head|Tail]):-
        sublist(Head).
nested_list([Head|Tail]):-
        nested_list(Tail).
sublist([]).
sublist([Head|Tail]).
```

Note that, for any non-empty list, a goal involving **nested_list/1** can be matched using either the first or the second clause. This produces the possibility that, if the goal is **redo**ne then it may once again succeed (if there is more than one occurrence of a sublist). This may not be what is wanted. You can test this with the query:

?- **nested_list([a,[b],c,[],[d],e]),write(y),fail.**

which produces the output **yyyno** because the first subgoal succeeds, the second writes **y** and the third fails (**fail/0** always fails!). Then backtracking occurs to **write/1** which fails.

We then backtrack into **nested_list/1** which *can* be resatisfied. Basically, the first success had terminated with the subgoal **sublist([b])** succeeding for the goal **nested_list([[b],c,[],[d],e])**. We can resatisfy this goal using the second clause which then sets up the goal **nested_list([c,[],[d],e])** which will eventually succeed. This will result in another **y** being written and, after a while, another attempt to resatisfy **nested_list/1** etc.

The point is that you are safe when no goal can be satisfied via different clauses. We could repair the above using an extralogical feature which is described in chapter 9 (the cut).

The program for **member/2** fits into this pattern when used with **mode member(+,+)**.

```
member(Element,[Element|Tail]).
member(Element,[Head|Tail]):-
                member(Element,Tail).
```

where there is one parameter —*viz* the first argument.

In case you are wondering where the **element_has_property** item has gone then we can rewrite **member/2** to the logically equivalent:

```
member(Element,[Head|Tail]):-
                Element = Head.
member(Element,[Head|Tail]):-
                member(Element,Tail).
```

Now we can see how this definition fits the above schema.

Test All Elements

In this situation we require that the elements of a list all satisfy some property. Here is the general schema:

```
test_all_have_property(Info,[]).
test_all_have_property(Info,[Head|Tail]):-
                element_has_property(Info,Head),
                test_all_have_property(Info,Tail).
```

Again, the expression **Info** stands for a specific number of parameters that carry information needed for the determination that an individual element has the desired property. The remaining argument is the *recursion argument*. We illustrate with a predicate **digits/1** for testing that a list of elements consists of digits only. We assume that we have **mode all_digits(+)**.

```
all_digits([]).
all_digits([Head|Tail]):-
                member(Head,[0,1,2,3,4,5,6,7,8,9]),
                all_digits(Tail).
```

plus definition of **member/2**.

This predicate has a declarative reading that a list has the property of consisting of digits if the first element is a digit and the tail of the list has the property of consisting of digits.

> Note that we can make this fit the schema better if the term **[0,1,2,3,4,5,6,7,8,9]** is passed in as a parameter.

Return a Result —Having Processed One Element

Now we turn to the idea that we can return a result. This requires an extra argument to be carried around —termed the *result* argument. We will now outline two further schemata that can be seen as developments of the two above. The first is intended to work through a list until an element satisfies some condition whereupon we stop and return some result. The schema is:

```
return_after_event(Info,[H|T],Result):-
                property(Info,H),
                result(Info,H,T,Result).
return_after_event(Info,[Head|Tail],Ans):-
                return_after_event(Info,Tail,Ans).
```

We will illustrate this with a predicate **everything_after_a/2** that takes a list and returns that part of the list after any occurrence of the element **a**. We assume that the mode is **mode everything_after_a(+,-)**.

```
everything_after_a([Head|Tail],Result):-
                Head = a,
                Result = Tail.
everything_after_a([Head|Tail],Ans):-
                everything_after_a(Tail,Ans).
```

Again, there are no parameters. There is one input (also the recursion argument) and one output argument (also the result argument).

> The first clause can be rewritten to:
>
> ```
> everything_after_a([a|Tail],Tail).
> ```

> Again, there is the same problem with this program as with the *test for existence* schema. The goal **everything_after_a([d,a,s,a,f],X)** will succeed with **X=[s,a,f]**. On **redo**ing, the goal can be resatisfied with **X=[f]**. This suggest that we have to be very careful about the meaning of this predicate.

Return a Result —Having Processed All Elements

We now deal with a very common task: taking a list of elements and transforming each element into a new element (this can be seen as a mapping). The schema for this is:

process_all(**Info**,[],[]).
process_all(**Info**,[H1|T1],[H2|T2]):-
process_one(**Info**,H1,H2),
process_all(**Info**,T1,T2).

where **process_one/1** takes **Info** and **H1** as input and outputs **H2**

The reading for this is that the result of transforming all the elements in the empty list is the empty list otherwise, transform the head of the list and then transform the rest of the list.

The second clause can be rewritten to:

process_all(**Info**,[H1|T1],Ans):-
process_one(**Info**,H1,H2),
process_all(**Info**,T1,T2),
Ans = [H2|T2].

Understanding the way in which this program works is quite difficult.

An example program is one that takes a list of integers and 'triples' each of them. The goal **triple([1,12,7],X** would result in **X=[3,36,21]**. We assume the mode of **mode triple(+,-)**.

triple([],[]).
triple([H1|T1],[H2|T2]):-
H2 is 3*H1,
triple(T1,T2).

This has the reading that the two arguments lie in the relation that the head of the second argument is 3 times that of the head of the first argument and the tails lie in the same relation. The declarative reading is easier to construct than exploring the way in which a goal is executed.

6.3.2 Reconstructing Lists

We now elaborate on a feature of the schema for *return a result —having processed all elements.* Looking at the structure of the head of the 2nd clause for **triple/2**, we see that the recursive call is structurally simpler than the head of the clause —*viz* **triple(T1,T2)** is 'simpler' than **triple([H1|T1],[H2|T2])**. The input variable for the recursive call, a list, is structurally smaller and so is the output variable.

Many students try to write **triple/2** as:

```
triple([],[]).
triple([H1|T1],T2):-
                H2 is 3*H1,
                triple(T1,[H2|T2]).
```

This does not work at all. Looking at the trace output, it is tempting to think the program is *nearly* right. Consider this trace output from SICStus **Prolog** for the goal **triple([1,2],X)**.

```
1   1 Call: triple([1,2],_95) ?
2   2 Call: _229 is 3*1 ?
2   2 Exit: 3 is 3*1 ?
3   2 Call: triple([2],[3|_95]) ?
4   3 Call: _520 is 3*2 ?
4   3 Exit: 6 is 3*2 ?
5   3 Call: triple([],[6,3|_95]) ?
5   3 Fail: triple([],[6,3|_95]) ?
4   3 Redo: 6 is 3*2 ?
4   3 Fail: _520 is 3*2 ?
3   2 Fail: triple([2],[3|_95]) ?
2   2 Redo: 3 is 3*1 ?
2   2 Fail: _229 is 3*1 ?
1   1 Fail: triple([1,2],_95) ?
```

At one point, we have a term **triple([],[6,3|_95])** which, if only it succeeded, might provide the result we want (even though it seems to be back to front). The first observation is that, since its first argument is [] it can only match the first clause for **triple** and this has a second argument of [] —so, this call must fail. The second observation is that each recursive call is called with an increasingly complex second argument —but, when the call is over, there is no way in which this complex argument can be passed back to the original query. For example, we start by trying to show that

```
triple([1,2],X) is true if triple([2],[3|X]) is true
```

Even if **triple([2],[3|X])** were true, that only means that **triple([1,2],X)** is true —where has the **3** gone?

We now describe the original schema for *return a result —having processed all elements* and an alternative way.

Building Structure in the Clause Head

This is the same as the previous *return a result —having processed all elements.* The following version of predicate **triple/2** is described as *building structure in the clause head*:

```
triple([],[]).
triple([H1|T1],[H2|T2]):-
                H2 is 3*H1,
                triple(T1,T2).
```

We can see this if we think of the output argument as a structure which is to be constructed out of two parts: a bit we can calculate easily (**H2**) and another bit which requires a recursive call to determine its structure (**T2**). The term **[H2|T2]** just shows how the result is constructed out of these bits.

Building Structure in the Clause Body

Now we produce a variant which achieves a similar (but not identical) effect. We introduce a new kind of variable: the *accumulator*. Consider the example:

```
triple([],Y,Y).
triple([H1|T1],X,Y):-
                H2 is 3*H1,
                triple(T1,[H2|X],Y).
```

We still have the first argument as the recursion argument but now the third argument is the result argument and the second argument is the accumulator. Now, we can see that the recursive call **reverse(T,[H|X],Y)** is simpler in the first argument than the head **reverse([H|T],X,Y)** and more complex in the second argument (the accumulator).

Note that the third argument is unchanged. If this is so, how can it take a value at all? Well, the recursion stops once we reach a call with its first argument as the empty list. This means that we will need to unify the goal with the head of the first clause. This will result in the second argument (the accumulator) being unified with the third argument (the result) which, at this point, is an unbound variable. We establish that this up-to-now unchanged variable is bound to the term in the accumulator. Following back along the path of recursive calls, we see that (more or less) the result we want is returned.

The goal **triple([1,2,3],[],X)** will result in **X=[9,6,3]**. Note two things: the expected order is reversed and that the accumulator has to be initialised in the original call. Sometimes, however, the order is not too important.

Here is the schema:

```
process_all(Info,[],Acc,Acc).
process_all(Info,[H1|T1],Acc,Ans):-
                process_one(Info,H1,H2),
                process_all(Info,T1,[H2|Acc],Ans).
```

where **process_one/1** takes **Info** and **H1** as input and outputs **H2**

6.4 Proof Trees

For an illustration of the difference between *building structure in the clause head* and *building structure in the clause body*, we construct an AND/OR proof tree for the goal **triple([1,2],Y)** using the code described previously for the *building structure in the clause head* case in figure 6.1 and, in figure 6.2, an AND/OR proof tree for the goal **triple([1,2],[],Y)** for the case of *building structure in the clause body*.

The method used to rename the variables is to use an superscript to indicate different instances of a variable.

> There is a slight cheat because the different instances of **Y** have not been distinguished. Really, there should be a succession of instances —$\mathbf{Y}^1$,$\mathbf{Y}^2$ and so on. They are, however, all established as equivalent (via unification).

You will notice that they are extremely similar in shape. The difference lies in the order of the construction of the variable bindings. Note that, in figure 6.1, the binding for **Y** is achieved after computing $\mathbf{T2}^1$ and the binding for $\mathbf{T2}^1$ is achieved after computing $\mathbf{T2}^2$ which is done through the clause **triple([],[]).** In the other case, in figure 6.2, the binding for **Y** is achieved through the clause **triple([],L,L).**

The main point is that one computation leaves incomplete structure around (which is eventually completed) while the other does not do so.

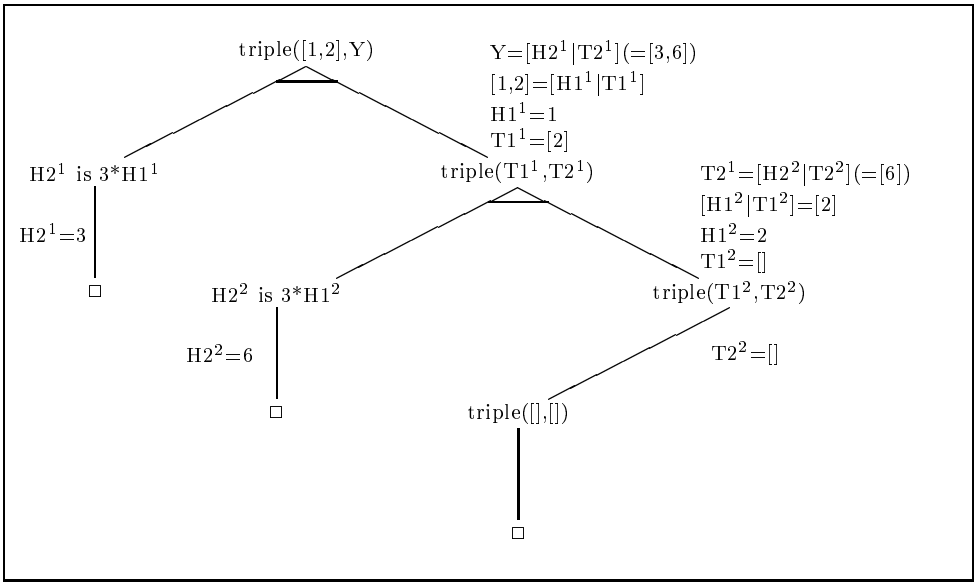


Figure 6.1: The Proof Tree for **triple([1,2],Y)**

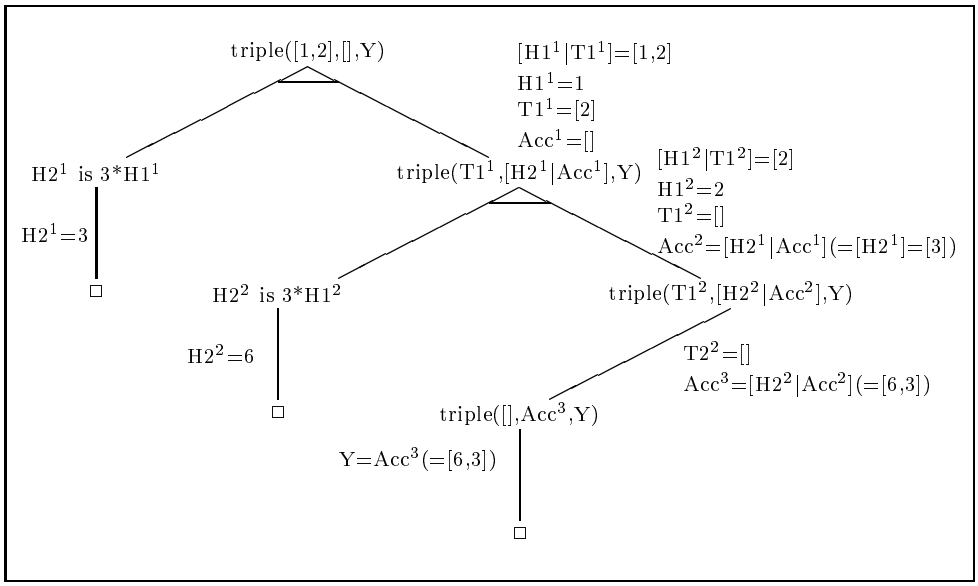


Figure 6.2: The Proof Tree for **triple([1,2],[],Y)**

6.5 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be aware of some standard techniques for processing lists and be able to identify programs that use these techniques.

Exercise 6.1 *We will now work on the four basic list schemata that we have suggested:*

1. *The Schema* test for existence
2. *The Schema* test all elements
3. *The Schema* return a result —having processed one element
4. *The Schema* return a result —having processed all elements

1. *The Schema* test for existence

   (a) *Define a predicate* **an_integer/1** *which checks to see if a list has at least one integer in it. Use the built-in predicate* **integer/1**.

   ?- an_integer([a,fred,5,X]).
   yes

   (b) *Define a predicate* **has_embedded_lists/1** *which checks to see if a list is an element which is itself a list. Assume that the input list contains no variables and that the empty list is not a member of this input list.*

   ?- has_embedded_lists([a,[b],c,d,e]).
   yes

2. *The Schema* test all elements

   (a) *Define a predicate* **all_integers/1** *that succeeds if and only if the (one) argument contains a list of integers.*

   ?- all_integers([1,fred,23]).
   no

   (b) *Define a predicate* **no_consonants/1** *which checks to see if a list of lower-case alphabetic characters has no consonants in it. Make up your own predicate to check whether an atom is a consonant.*

   ?- no_consonants([a,e,i,t]).
   no
   ?- no_consonants([a,e,e,i]).
   yes

3. *The Schema* return a result —having processed one element

   (a) *Write a predicate* **nth/3** *which takes two inputs: the first a positive integer and the second a list. The* output *(initially, an uninstantiated variable) will be the element that occurs at the nth position in the list. So*

   ?- nth(3,[this,is,[an,embedded,list]],X).
   X=[an,embedded,list]

   (b) *Define a predicate* **next/3** *which again takes two inputs: a possible member of a list and the list. The* output *should be the element of the list that immediately follows the named list element (if it exists —if not, the predicate should fail).*

   ?- next(a,[b,r,a,m,b,l,e],X).
   X=m

   (c) *define del_1st/3 which takes a possible element of a list and a list as inputs and "returns" the list with the first occurence of the named element removed. (If the named element is not in the list then the predicate is to fail)*

   ?- del_1st(a,[b,a,n,a,n,a],X).
   X=[b,n,a,n,a]

   This one can also be solved using an accumulator with the help of **append/3**.

4. *The Schema* return a result —having processed all elements *All these can be done in two ways. One uses the idea of* building structure in the clause head *and the other* building structure in the clause body.

   Remember that the latter requires one more argument than the former — the accumulator. *As this usually needs initialising it is customary to do this by such as:*

```
foo(a,X):-
                    foo(a,[],X).
```

Do each problem both ways.

(a) *Define* **nple/3** *to take two inputs —an integer and a list of integers. The* result *is to be a list of integers formed by multiplying each integer in the list by the input integer.*

```
?- nple(5,[1,2,3],X).
X=[5,10,15]
```

(b) *Define* **del_all/3** *which takes a possible element of a list and a list as inputs and* returns *the list with all occurences of the named element removed. (If the named element is not in the list then the* result *is, of course, the whole list with no deletions)*

```
?- del_all(a,[b,a,n,a,n,a],X).
X=[b,n,n]
```

(c) *Define* **sum/2** *to take a list of integers as input and* return *the output as their sum. This one is slightly unusual with regard to the base case.*

```
?- sum([1,32,3],X).
X=36
```

Chapter 7

Control and Negation

We introduce a number of facilities for controlling the execution of **Prolog**. We outline the problem of trying to represent logical negation and one solution. We introduce some more programming techniques.

7.1 Some Useful Predicates for Control

true/0

Always succeeds.

```
father(jim,fred).
```

is logically equivalent to

```
father(jim,fred):-
                true.
```

That is, any *unit clause* is equivalent to a non-unit clause with a single subgoal **true** in the body.

fail/0

Always fails.

```
lives_forever(X):-
                fail.
```

is intended to mean that any attempt to solve the goal **lives_forever(X)** will fail.

repeat/0

If it is asked to **Redo** then it will keep on succeeding.

```
test:-
        repeat,
        write(hello),
        fail.
```

The goal **test** produces the output:

```
hellohellohellohellohellohellohellohellohello...
```

repeat/0 behaves as if it were defined in **Prolog** as:

```
repeat.
repeat:-
        repeat.
```

call/1

The goal **call(X)** will call the interpreter as if the system were given the goal **X**. Therefore X must be bound to a legal **Prolog** goal.

```
?- call(write(hello)).

hello
yes
```

To handle a query which has multiple goals then:

```
?- call((write(hello),nl)).

hello
yes
```

Note that we cannot write **call(write(hello),nl)** as this would be taken to be a usage of **call/2** with the first argument **write(hello)** and the second argument **nl)** —and most systems do not have a **call/2**.

Note that **call/1** is unusual in that its argument must be a legitimate **Prolog** goal. Also note that **call(X)** will be legal if and only if **X** is bound to a legal goal.

7.2 The Problem of Negation

To maintain the connection with predicate logic, we would like to be able to represent the negation of a statement. This, however, proves to be problematic.

Consider

```
man(jim).
man(fred).

?- man(bert).

no
```

To say that **man(bert)** is not true we have to assume that we known all that there is to know about **man/1**. The alternative is to say the the **no** indicates **don't know** and this is not a possible truth value!

Turning to **Prolog**, If we try to solve a goal for which there is no clause (as in the case above) then we assume that we have provided **Prolog** with all the necessary data to solve the problem. This is known as the *Closed World Assumption*.

This enables us to stick to the desirable property that a goal can have only two outcomes.

\+/1

This strangely named predicate is **Prolog**'s equivalent to the **not** (often written as ¬ which stands for negation) of predicate logic. It is not named **not/1** because we it turns out that we cannot easily implement classical negation in **Prolog**.

The predicate **\+/1** takes a **Prolog** goal as its argument. For example:

```
?- \+( man(jim) ).
```

will succeed if **man(jim)** fails and will fail if **man(jim)** succeeds.

7.2.1 Negation as Failure

Negation as failure is the term used to describe how we use the *closed world assumption* to implement a form of negation in **Prolog**. We now give an example which uses a rule to define women in terms of them not being men. Logically, $\forall$ x $\in$ people ($\neg$ man(x) $\Longrightarrow$ woman(x)).

```
man(jim).
man(fred).
woman(X):-
                \+( man(X) ).

?- woman(jim).

no
```

The strategy is: to solve the goal **woman(jim)** try solving **man(jim)**. This succeeds —therefore **woman(jim)** fails. Similarly, **woman(jane)** succeeds. But there is a problem. Consider:

```
?- woman(X).
```

It succeeds if **man(X)** fails —but **man(X)** succeeds with **X** bound to **jim**. So **woman(X)** fails and, because it fails, **X** *cannot* be bound to anything.

> We can read **?- woman(X)** as a query "is there a woman?" and this query failed. Yet we know that **woman(jane)** succeeds. Therefore, this form of negation is not at all like logical negation.
>
> The problem can be highlighted using predicate logic. The query **woman(X)** is interpreted as
>
> $\exists$ x $\neg$ man(x)
>
> which, logically, is equivalent to
>
> $\neg$ $\forall$ x man(x)
>
> Now **Prolog** solves this goal in a manner roughly equivalent to
>
> $\neg$ $\exists$ x man(x)
>
> The only time we get something like the desired result if there is no existentially quantified variable in the goal. That is, whenever **\+/1** is used then make sure that its argument is bound at the time it is called.

Also, note that **\+(\+(man(X)))** is not identical to **man(X)** since the former will succeed with **X** *unbound* while the latter will succeed with **X** bound, in the first instance, to **jim**.

> This is the basis of a well known **Prolog** programming 'trick' —*i.e.* it is a technique which is frowned upon by purists. The idea is to test whether, for example, two terms will unify without the effect of binding any variables. The goal **\+(\+(X=2))** will succeed without binding **X** to **2**. The meaning is roughly **X** *would unify* with **2**.

7.2.2 Using Negation in Case Selection

We can use **\+/1** to define relations more carefully than previously. To illustrate, consider

```
parity(X,odd):-
                odd(X).
parity(X,even).
```

together with the set of facts defining **odd/1**.

The goal **parity(7,X)** is intended to succeed using the first clause. Suppose that some later goal fails forcing backtracking to take place in such a way that we try to **redo parity(7,X)**. This goal unifies with the rest of the second clause! This is not desirable behaviour. We can fix this using **\+/1**.

```
parity(X,odd):-
                odd(X).
parity(X,even):-
                \+(odd(X)).
```

Thus **\+/1** provides extra expressivity as we do not need a set of facts to define **even/1**.

If we go back to a previous example found in section 6.3.1 then we can now resolve the problem about how to deal with unwanted backtracking in programs like:

```
nested_list([Head|Tail]):-
                    sublist(Head).
nested_list([Head|Tail]):-
                    nested_list(Tail).
sublist([]).
sublist([Head|Tail]).
```

The problem is caused by the fact that a goal like **nested_list([a,[b],c,[d]])** will succeed once and then, on **redo**ing, will succeed once more. This happens because the goal unifies with the heads of both clauses —*i.e.* with **nested_list([Head|Tail])** (the heads are the same). We can now stop this with the aid of **\+/1**:

```
nested_list([Head|Tail]):-
                    sublist(Head).
nested_list([Head|Tail]):-
                    \+(sublist(Head)),
                    nested_list(Tail).
sublist([]).
sublist([Head|Tail]).
```

Note that this is at the price of often solving the identical subgoal twice —the repeated goal is **sublist(Head)**. Note also that there is never more than one solution for **sublist(X)**.

Finally, we can define **\+/1** using **call/1** and the cut (**!/0**:

```
\+(X):-
                    call(X),
                    !,
                    fail.
\+(X).
```

This is a definition which essentially states that "if **X**, interpreted as a goal, succeeds then **\+(X)** fails. If the goal **X** fails, then **\+(X)** succeeds. To see this is the case, you have to know the effect of the *cut — fail* combination (**!,fail**). See later on in this chapter for more details of this.

7.3 Some General Program Schemata

We have already introduced some list processing schemata. Now we discuss some further, very general, program schemata.

Generate — Test

One of the most common techniques in **Prolog** is to use the backtracking in first generating a possible solution, then testing the possible solution to see if it is acceptable. If not, backtracking takes place so that another possible solution can be generated.

generate_and_test(**Info**,X):-
    ...
    generate(**Info**,X),
    test(**Info**,X),
    ...

In the above schema, the ellipsis (...) indicates a number of subgoals (0 or more).

We can distinguish two kinds of generator: a finite generator and an infinite generator. We will illustrate with two different versions of a non-negative integer generator which we will call **int/1** —we cannot name this **integer/1** since this is already defined (as a built-in predicate) and it only works with **mode integer(+)** and we want **int/1** to work with **mode int(-)**.

Finite and Infinite Generators

We define a predicate **integer_with_two_digit_square/1** to produce a positive integer that has a square which is greater than or equal to 10 and less than 100.

```
integer_with_two_digit_square(X):-
        int(X),
        test_square(X).

test_square(X):-
        Y is X*X,
        Y >= 10,
        Y < 100.
```

Here is the definition of **int/1** which is a finite generator —because there are only a finite number of unit clauses (containing no variables) used to define **int/1**.

```
int(1).
int(2).
int(3).
int(4).
int(5).
```

The goal **integer_with_two_digit_square(X)** eventually fails because the generator runs out of potential solutions. Now we define a version of **int/1** which is an infinite generator (verifying this is left as an 'exercise for the reader'!).

```
int(1).
int(N):-
        int(N1),
        N is N1 +1.
```

On backtracking, this will generate a new solution for **integer_with_two_digit_square(X)** until we test **10**. From then on, we will keep generating with **int/1** and failing with **test_square/1**. We are trapped in a *generate—test* cycle with no way out.

The usual way out is to ensure that once we have found the solution we want then we *commit* ourselves to that solution and forbid backtracking from ever seeking another solution. Again, the usual solution is to place a cut (**!/0**) after the test. This results in:

```
integer_with_two_digit_square(X):-
                    int(X),
                    test_square(X),!.
```

and the example demonstrates the (usually necessary) fix to stop a program using the *generate — test* schema from *overgenerating*. However, our solution now provides for only one solution to be generated!

Test — Process

Now we look at another fundamental schema. The idea with *test — process* is to guarantee that some inputs will only be 'processed' if the input passes a test.

```
test_process(Info,X,Y):-
                    test(Info,X),
                    process(Info,X,Y).
```

where we assume that the **Info** is 0 or more arguments which are all input arguments, the last but one argument is an input argument and the last argument is a output argument. Although this gives a very procedural view it is often possible to give a declarative reading.

We usually want to make sure that

1. *test* does not have alternative ways of confirming that the generated element is ok
2. *process* does not have alternative ways of 'processing' the input

In short, we often want only one way of finding an output.

We have already met a program that satisfies this schema —one for **parity/2** (which is slightly rewritten here).

```
parity(X,Y):-
                    odd(X),
                    Y=odd.
parity(X,Y).
                    \+(odd(X)),
                    Y=even.
```

plus set of facts defining **odd/1**

This example illustrates that if the input argument is an integer then we see two cases: either the integer is even or it is odd. There is no third case. Nor can any integer be both **even** and **odd**.

As in the above example, the usage of *test — process* is closely coupled with the idea of writing all the clauses for a predicate in this form —each clause is designed to handle one 'class' of input. The whole scheme falls down if we do not design the 'classes' of input to be disjoint –*i.e.* no input falls into more than one category. We also require that each input falls in at least one category —to summarise, each input falls in one and only one class.

We can show a previous example which does not properly use the *test — process* schema (for good reasons). Modifying the code using this schema results in a different *and* useful program.

```
member(Element,[Element|Tail]).
member(Element,[Head|Tail]):-
                member(Element,Tail).
```

Now **member/2** can be used as a generator if the first argument is a variable and its second argument is a list —as in the goal **member(X,[a,b,c,d,e,f]**. The first solution for **X** is the first element of the list **[a,b,c,d,e,f]**. On **redo**ing, we get, in succession, **X** bound to the different elements in the list.

We now rewrite using the *test — process* schema. We also rename the predicate to the standard name of **memberchk/2** (this is its usual name in libraries of **Prolog** code).

```
memberchk(Element,[Head|Tail]):-
                Element = Head.
memberchk(Element,[Head|Tail]):-
                \+(Element = Head),
                memberchk(Element,Tail).
```

This will no longer generate alternative solutions on backtracking for the goal **memberchk(X,[a,b,c,d,e,f])** (because there are no alternative ways of resatisfying it). If the mode of use is **mode memberchk(+,+)** then the meaning is that we check that the first argument is an element of the list (which is the second argument).

Failure-Driven Loop

We now introduce an extremely procedural programming technique for simulating a kind of iteration. The idea is deliberately *generate* a term and then *fail*. This suggests the useless schema

```
failure_driven_loop(Info):-
                generate(Info,Term),
                fail.
failure_driven_loop(Info).
```

Provided that the *generator* eventually fails any version of this schema will always succeed —*i.e.* it will be equivalent to **true**.

We now use *side effecting* predicates to do something useful with the generated term.

A *side-effecting* predicate is one that is (often) logically equivalent to **true** but also does something else that is non-logical. For example, **write/1** and **nl/0** have the side-effect of writing material onto the terminal screen (usually). Also, **consult/1** and **reconsult/1** have the side-effect of changing the program. The predicate **read/1** has the side-effect of destructively reading input from the terminal (or whatever).

To illustrate the problem: if we query **Prolog** with the goal **(write(hello),fail)** then **write/1** will be used to write **hello** on (we assume) the terminal screen and the call to **fail/0** will fail. Now, logically, we have a statement with the truth value of **false** —so we have proved that the goal cannot succeed and therefore there should be no message (**hello**) on the screen.

Here is another example: if we try the goal **(read(X),fail)** then **read/1** will be used to read some input from the user (we assume) and the call to **fail/0** will fail. Again, we have a statement with the truth value of **false** —so the input should still be available for consideration. Yet we taken input from the keyboard (or somewhere) and we do not put that input back so that it can be reconsidered. The input has been consumed.

We can see that any predicate succeeds generating an effect that cannot be undone on backtracking must be a *side-effecting* predicate.

The complete *failure-driven loop* schema can be taken as:

```
failure_driven_loop(Info):-
                    generate(Info,Term),
                    side_effect(Term),
                    fail.
failure_driven_loop(Info).
```

This can be elaborated by having several *side-effecting* predicates, replacing the **fail/0** with some other predicate that fails and so on.

We illustrate with a simple example. We will use **int/1** as a finite generator and then print out the valid arguments for this relation on the screen.

```
int(1).
int(2).
int(3).
int(4).
int(5).

print_int:-
                    int(X),
                    write(X),nl,
                    fail.
print_int.
```

This programming technique can be very useful. In the early days, it was overused because it was space-efficient.

Some Practical Problems

We now come to some needs that cannot easily be satisfied and still retain a clean declarative reading. We look at three problems that are interconnected.

Commit

We have outlined the use of *test* — *process* to do case analysis but it was necessary to have one clause for each case. If we have a goal which can be satisfied via two different clauses then, on **redo**ing, the same goal may generate a different solution.

In reality, this situation can arise quite often —*i.e.* the tests we do on the input do not divide the input into non-overlapping classes. Essentially, we have two problems. We often want to make sure that only one clause is legitimate —once it has been determined that the input passes some test. We think of this as a statement of *commitment* to the solution(s) derived through 'processing' the input.

test_process(**Info**,X,Y):-
test(**Info**,X),
commit,
process(**Info**,X,Y).

When we backtrack and try to find another way of satisfying some program that makes use of the *test* — *process* schema then we first try to find another way of satisfying the *process* part. If that fails, then we try to resatisfy the *test* part. We do not want this to happen.

Then, assuming that we cannot resatisfy the *test* part, we try to resatisfy the goal making use of this program by trying different clauses.

Therefore there are two senses in which we may want to be 'committed': we want to *commit* to using a single clause and we want to *commit* to the result of a test —we do not want to run the risk that the test can be successful (with the same input) twice.

Satisfy Once Only

Sometimes, we would like a way of stopping **Prolog** looking for other solutions. That is, we want some predicate to have only one solution (if it has one at all). This is the requirement that the predicate be **determinate**.

Naturally, predicates which do not have this property are *indeterminate*. This is a desirable property sometimes —*e.g.* the *generate* — *test* schema makes use of the *generator* being *indeterminate*. On the other hand, it can cause major problems when a program has many predicates which are unintentionally *indeterminate*. Our aim is to make sure that those predicates which should be *determinate* actually are *determinate*.

We have already met an example of a predicate (**memberchk/2**) that might have been written with this situation in mind. We recall that **member/2** used with **mode member(-,+)** behaves as a generator. Perhaps it is worth pointing out that **member/2** with **mode member(+,+)** is also, under certain

circumstances, resatisfiable —precisely when there are repetitions of the sought element in the list which constitutes the second argument.

Of course, if we are dealing with *lists–as–sets*, we should have arranged it so that the second argument does not have repeated elements. Anyway, it is very desirable to have a *determinate* version of **member/2** available.

```
memberchk(X,[X|Y]):-
                make_determinate.
memberchk(X,[Y|Z]):-
                memberchk(X,Z).
```

Note this isn't quite what we had before. Previously, we arranged for **memberchk/2** to be determinate with the help of **\+/1**. Stating our requirement as above, we seem to be going outside of logic in order to tell the **Prolog** interpreter that, once we have found the element sought, we never want to consider this predicate as resatisfiable.

Fail Goal Now

We often search for the solution to a goal using several clauses for some predicate. For example, we might have a social security calculation which tries to assign how much money to give a claimant. Here is a fragment of program:

```
calculate_benefit(Claim_Number,Nationality,Age,Other_Details):-
                Nationality = british,
                calculate_british_entitlement(Age,Other_Details).
calculate_benefit(Claim_Number,Nationality,Age,Other_Details):-
                Nationality = martian,
                give_up.
calculate_benefit(Claim_Number,Nationality,Age,Other_Details):-
                Nationality = french,
                calculate_french_entitlement(Age,Other_Details).
```

If we reach the situation where we realise that the whole search is doomed then we may want to say something informally like 'stop this line of approach to the solution and any other corresponding line'. In the above, if we find we are trying to assign benefit to a martian then we make the decision that **calculate_benefit/4** should fail and therefore that there is no point in trying to use any remaining clauses to find a solution.

In practice, we need to make use of this kind of action. Again, we are potentially asking **Prolog** to behave abnormally.

In fact, in all these situations, we are asking **Prolog** to behave in a non-standard way. Whatever the complications, it is hard top make do without ways to:

- *Commit*
- *Make Determinate*
- *Fail Goal Now*

7.4 What You Should Be Able To Do

You should be able to make use of the 'predicates' **true/0**, **fail/0**, **repeat/0** and **call/1**.
You should be able to describe the difference between the open and closed world assumptions.
You should be able to describe the difference between classical negation and negation as failure.
You should be able to distinguish *side-effecting* from *non side-effecting* predicates.
You should be able to use **Prolog** negation to achieve the effect of case selection.
You should be able to use the techniques of *generate — test*, *test — process* and *failure-driven loop*. You should also be aware of the needs for the techniques *commit — process*, *satisfy–only–once* and *fail–goal–now*.

Chapter 8

Parsing in Prolog

We introduce the facilities that **Prolog** provides for parsing. This is done through the idea of a parse tree as applied to a simple model for the construction of English sentences. Three ways of parsing **Prolog** are described: the first illustrates the ideas, the second is more efficient and the third provides an easy way of coding a parser via *Grammar Rules*. We then explain how to extract the parse tree and show how to extend a parser using arbitrary **Prolog** code.

Later on in the course, you will be involved in trying to face up to the problem of parsing ordinary english language sentences. For this lecture, we shall also be interested in parsing sentences but we will look at the very simplest examples.

First, what do we want the parser to do? We would like to know that a sentence is correct according to the (recognised) laws of english grammar.

The ball runs fast

is syntactically correct while

The man goes pub

is not as the verb "go" (usually) does not take a direct object.

Secondly, we may want to build up some structure which describes the sentence —so it would be worth returning, as a result of the parse, an expression which represents the syntactic structure of the successfully parsed sentence.

Of course, we are not going to try to extract the *meaning* of the sentence so we will not consider attempting to build any *semantic* structures.

8.1 Simple English Syntax

The components of this simple syntax will be such categories as sentences, nouns, verbs etc. Here is a (top down) description:

Unit: sentence
Constructed from: noun phrase followed by a verb phrase

Unit: noun phrase
Constructed from: proper noun or determiner followed by a noun

Unit: verb phrase
Constructed from: verb or verb followed by noun phrase

Unit: determiner
Examples: a, the

Unit: noun
Examples: man, cake

Unit verb:
Examples: ate

8.2 The Parse Tree

Figure 8.1 shows the parse tree for the sentence:

the man ate the cake

with some common abbreviations in brackets. We must remember that many

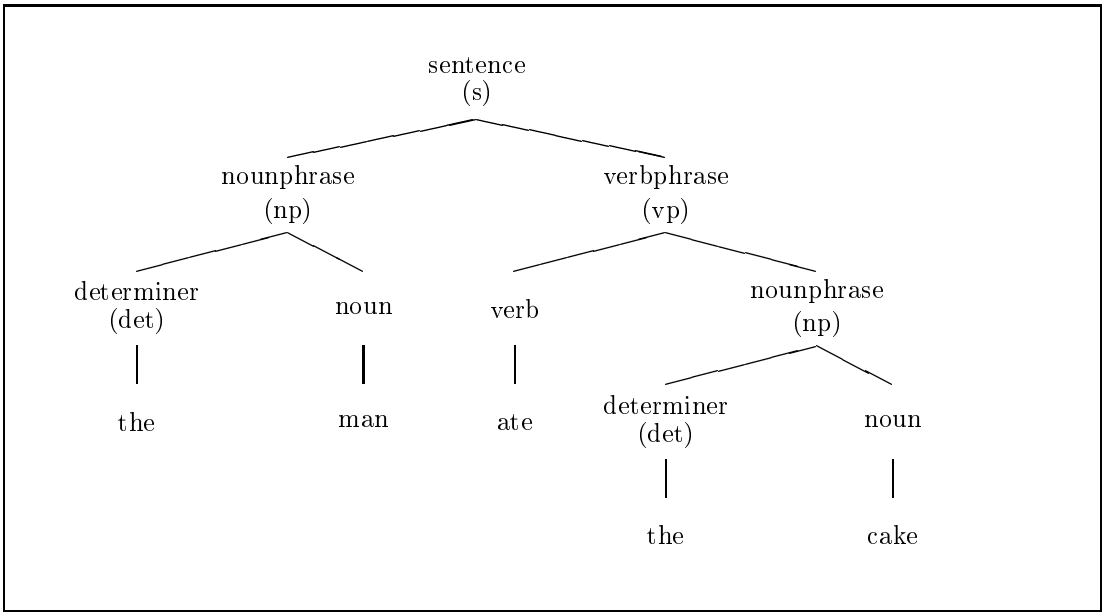


Figure 8.1: A Parse Tree

sentences are ambiguous —*i.e.* they result in different parse trees.

8.3 First Attempt at Parsing

We assume that we will parse sentences converted to list format. That is, the sentence "the man ate the cake" will be represented by the list **[the,man,ate,the,cake]**.

We use **append/3** to glue two lists together. The idea is that **append/3** returns the result of gluing takes input as lists in the first and second argument positions and returns the result in the third position.

```
sentence(S):-
                append(NP,VP,S),
                noun_phrase(NP),
                verb_phrase(VP).

noun_phrase(NP):-
                append(Det,Noun,NP),
                determiner(Det),
                noun(Noun).

verb_phrase(VP):-
                append(Verb,NP,VP),
                verb(Verb),
                noun_phrase(NP).

determiner([a]).
determiner([the]).
noun([man]).
noun([cake]).
verb([ate]).
```

Here is what happens to the query:

```
?- sentence([the,man,ate,the cake]).

append/3 succeeds with NP=[], VP=[the,man,ate,the,cake]
noun_phrase/1 fails
append/3 succeeds with NP=[the], VP=[man,ate,the,cake]
noun_phrase/1 fails
append/3 succeeds with NP=[the,man], VP=[ate,the,cake]
noun_phrase/1 succeeds
...
verb_phrase/1 succeeds
```

This is all very well but the process of parsing with this method is heavily *non deterministic*.

Also, it suffers from not being a very flexible way of expressing some situations. For example, the problem of adjectives:

```
the quick fox
```

is also a *noun phrase*.

We might try to parse this kind of noun phrase with the extra clause:

```
noun_phrase(NP):-
                append(Det,Bit,NP),
                determiner(Det),
                append(Adj,Noun,Bit),
                adjective(Adj),
                noun(Noun).
```

A little ungainly.

8.4 A Second Approach

We now try an approach which is less non-deterministic. We will start by looking at:

```
sentence(In,Out)
```

The idea is that **sentence/2** takes in a list of words as input, finds a legal sentence and returns a result consisting of the input list minus all the words that formed the legal sentence.

We can define it:

```
sentence(S,S0):-
                noun_phrase(S,S1),
                verb_phrase(S1,S0).
```

Here is a rough semantics for **sentence/2**.

> A sentence can be found at the front of a list of words if there is a noun phrase at the front of the list and a verb phrase immediately following.

This declarative reading should help to bridge the gap between what we want to be a sentence and the procedure for finding a sentence.

Here is the rest of the parser:

```
noun_phrase(NP,NP0):-
                determiner(NP,NP1),
                noun(NP1,NP0).
verb_phrase(VP,VP0):-
                verb(VP,VP1),
                noun_phrase(VP1,VP0).
determiner([a|Rest],Rest).
determiner([the|Rest],Rest).
noun([man|Rest],Rest).
noun([cake|Rest],Rest).
verb([ate|Rest],Rest).
```

As you can see, there is a remarkable sameness about each rule which, once you see what is going on, is fairly tedious to type in every time. So we turn to a facility that is built in to **Prolog**.

8.5 Prolog Grammar Rules

Prolog, as a convenience, will do most of the tedious work for you. What follows, is the way you can take advantage of **Prolog**.

This is how we can define the simple grammar which is accepted 'as is' by **Prolog**.

```
sentence      -->    noun_phrase, verb_phrase.
noun_phrase   -->    determiner, noun.
verb_phrase   -->    verb, noun_phrase.
determiner    -->    [a].
determiner    -->    [the].
noun          -->    [man].
noun          -->    [cake].
verb          -->    [ate].
```

It is very easy to extend if we want to include adjectives.

```
noun_phrase   -->    determiner, adjectives, noun.
adjectives    -->    adjective.
adjectives    -->    adjective, adjectives.
adjective     -->    [young].
```

This formulation is sometimes known as a *Definite Clause Grammar* (DCG).

We might later think about the ordering of these rules and whether they really capture the way we use adjectives in general conversation but not now.

Essentially, the **Prolog** Grammar Rule formulation is *syntactic sugaring*. This means that **Prolog** enables you to write in:

```
sentence      -->    noun_phrase, verb_phrase.
```

and **Prolog** turns this into:

```
sentence(S,S0):-
                noun_phrase(S,S1),
                verb_phrase(S1,S0).
```

and

```
adjective     -->    [young].
```

into

```
adjective(A,A0):-
                'C'(A,young,A0).
```

where **'C'/3** is a built in **Prolog** Predicate which is defined as if:

```
'C'([H|T],H,T).
```

8.6 To Use the Grammar Rules

Set a goal of the form

```
sentence([the,man,ate,a,cake],[])
```

and not as

```
sentence.
```

or

```
sentence([the,man,ate,a,cake])
```

8.7 How to Extract a Parse Tree

We can add an extra argument which can be used to return a result.

```
sentence([[np,NP],[vp,VP]])              -->    noun_phrase(NP), verb_phrase(VP).
noun_phrase([[det,Det],[noun,Noun]])     -->    determiner(Det), noun(Noun).
determiner(the)                          -->    [the].
and so on
```

What we have done above is declare predicates **sentence/3**, **noun_phrase/3**, **verb_phrase/3**, **determiner/3** and so on. The explicit argument is the first and the two others are added when the clause is read in by **Prolog**. Basically, **Prolog** expands a grammar rule with n arguments into a corresponding clause with n+2 arguments.

So what structure is returned from solving the goal:

```
sentence(Structure,[the,man,ate,a,cake],[])
```

The result is:

```
[[np,[[det,the],[noun,man]]],[vp,[...
```

Not too easy to read!

We can improve on this representation if we are allowed to use **Prolog** terms as arguments. For example, in **foo(happy(fred),12)** the term **happy(fred)** is one of the arguments of **foo/2**. Such a term is known as a *compound term*. We discuss this at greater length in chapter 10.

With the help of compound terms, we could tidy up our representation of sentence structure to something akin to:

```
sentence([np([det(the),noun(man)]),vp([...
```

8.8 Adding Arbitrary Prolog Goals

Grammar rules are simply expanded to **Prolog** goals. We can also insert arbitrary **Prolog** subgoals on the right hand side of a grammar rule but we must tell **Prolog** that we do not want them expanded. This is done with the help of *braces —i.e.* { }. For example, here is a grammar rule which parses a single character input as an ASCII code and succeeds if the character represents a digit. It also returns the digit found.

```
digit(D) -->
                [X],
                { X >= 48,
                  X =< 57,
                  D is X-48 }.
```

The grammar rule looks for a character at the head of a list of input characters and succeeds if the **Prolog** subgoals

```
{ X >= 48,
  X =< 57,
  D is X-48 }.
```

succeed. Note that we assume we are working with ASCII codes for the characters and that the ASCII code for "0" is 48 and for "9" is 57. Also note the strange way of signifying "equal to or less than" as "=<".

8.9 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to write a simple parser for a small subset of English.
You should be able to use **Prolog**'s *grammar rules* to define the grammar of a simple language.
You should be able to describe how **Prolog** rewrites the grammar rules into 'standard' **Prolog**.
You should be able to use the *grammar rules* to extract a parse tree.

Exercise 8.1 *Here is a Definite Clause Grammar:*

```
s       -->  np,vp.
np      -->  det,noun.
np      -->  det,adjs,noun.
vp      -->  verb,np.
det     -->  [a].
det     -->  [the].
adjs    -->  adj.
adjs    -->  adj,adjs.
adj     -->  [clever].
noun    -->  [boy].
noun    -->  [sweet].
verb    -->  [buys].
```

1. *Give some examples of sentences that this grammar could parse.*

2. *Modify this DCG so that the parse returns information about the structure of the sentence.*

3. *Suppose that the DCG is given a sentence to parse containing a misspelled word -say "boy". Modify the DCG so that the information about the structure of the sentence will include some information about any unrecognised component.*

4. *Suppose now that the DCG is given a sentence to parse missing a word or two. Modify the DCG so that it will identify the missing component.*

The last two parts of this exercise are hard —it is essentially the problem of robust parsing. *We try to do the best we can to identify gaps, misspellings and redundant information.*

Chapter 9

Modifying the Search Space

We describe solutions to various problems of control raised in chapter 7.
We detail other useful **Prolog** built-in predicates that are non-logical.

9.1 A Special Control Predicate

We now present a solution to the practical problems posed in chapter 7 about how to control **Prolog**'s search strategy. We summarised these issues as ones of:

- *Commit*
- *Make Determinate*
- *Fail Goal Now*

In each of these cases the solution is to make use of a built-in predicate which always succeeds —but with a very unpleasant *side-effect*. This notorious predicate is known as the *cut* and written **!/0**.

The reason why cut (**!/0**) is so unpleasant are that it effects **Prolog**'s search tree. Consequently, by adding a cut, the program's meaning *may* change radically. We sometimes say that a cut that does this is a *red cut*. On the other hand, the placing of a cut may not change the intended meaning but simply junk a part of the search tree where it is known that there is no legal solution. Such a cut is termed a *green cut*. The *Art of Prolog* by Sterling and Shapiro has a nice section on the cut [Sterling & Shapiro, 1986].

We now go over how to solve the three control problems.

9.1.1 Commit

Assume we want to make Social Security payments. That is, **pay(X,Y)** means "pay the sum X to Y". Assume that we also have this code fragment.

```
pay(X,Y):-
                british(X),
                entitled(X,Details,Y).
```

```
pay(X,Y):-
                european(X),
                entitled(X,Details,Y).
```

In each clause, the first subgoal in the body is acting as a *test* in a program using the *test — process* schema. We also assume that, for some reason, we have not been able to apply the disjoint (and exhaustive) case analysis technique.

Consequently, if we have successfully checked that a person is British and, for some reason, the subgoal **entitled(X,Details,Y)** fails (or some later computation forces backtracking back to **redo** the call to **pay/2** that we are considering) then there may be *no point* in

- checking if they are "european" (assuming that there are no regulations under which British people can qualify for payment as being European when they fail to qualify as British citizens).
- checking to see if there is more than one entry for the person in some database accessed by **british/1**.

In the immediate situation, we want to be committed to telling **Prolog** not to **redo** the **british/1** subgoal *and* not to consider other clauses for **pay/2** that *might* contribute an alternative.

The truth is, of course, that we may want these two consequences whether or not **entitled/3** fails.

If this is so, then we insert a cut as shown below and highlighted by a box.

```
pay(X,Y):-
                british(X),
                !,
                entitled(X,Details,Y).
pay(X,Y):-
                european(X),
                !,
                entitled(X,Details,Y).
```

We want to be committed to the choice for the **pay/2** predicate. We can see the use of **!/0** as a *guard* that has two effects.

- On backtracking through the list of subgoals: a **cut** can be thought of as indicating that all attempts to **redo** a subgoal to the left of the **cut** results in the subgoal immediately **fail**ing. We sometimes say that any unifications taking place *prior* to the cut have been *frozen* and cannot be remade.
- On backtracking into the predicate once the call had **exit**ed: if one of the clauses defining the predicate had previously contained a **cut** that *had been executed* then no other clauses for that predicate may be used to resatisfy the goal being **redo**ne. We sometimes say that, once a cut is executed, later clauses have been *chopped* out of the *search space*.

Note that, with the cut in the position it is above, it is still possible that **entitled/3** could be resatisfied. We have to guarantee that we have made **entitled/3** determinate before we can guarantee that **pay/2** is determinate. We have to do some more on this issue.

Also note that the effect of cut (**!/0**) prunes the search space only until the parent goal of the cut fails. If we leave the **Fail** port of **pay/2** and some previous goal leads to another call to **pay/2** then the cut (**!/0**) has no effect until it is executed.

We also have to remember that cut (**!/0**) has two distinct effects: backtracking cannot **redo** any subgoals to the left of the cut and clauses in the program database for the same predicate that are textually after the current clause are unreachable. See figure 9.1 for a graphic representation of these effects on a rather artificial program.

```
a(X):-  b(X),c(X).
b(1).
b(4).
c(X):-  d(X),!,e(X).
c(X):-  f(X).
d(X):-  g(X).
d(X):-  h(X).
e(3).
f(4).
g(2).
h(1).
```

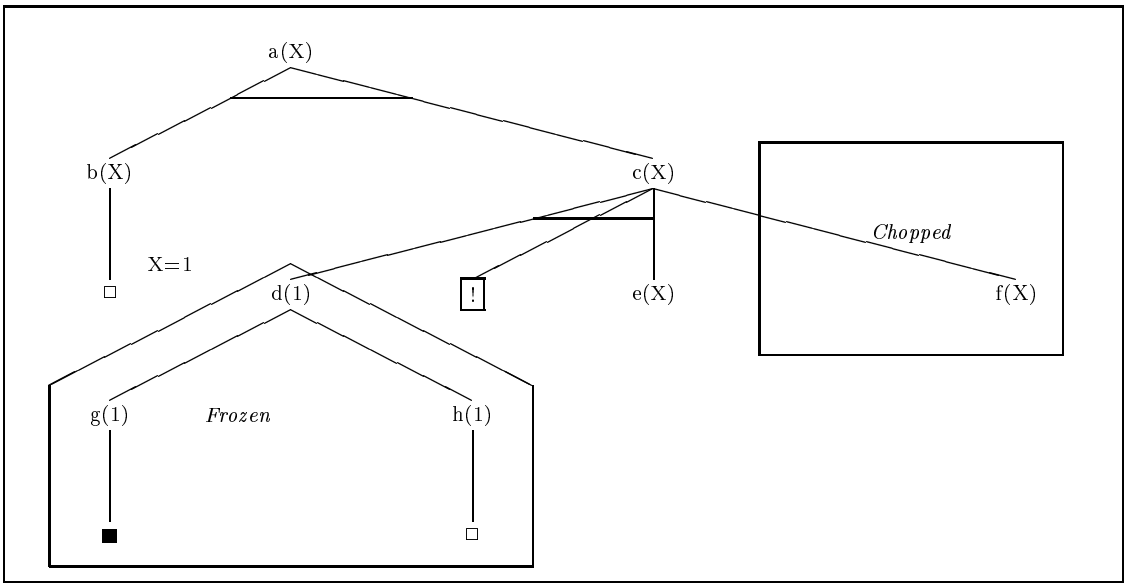


Figure 9.1: The Effect of **cut** on the AND/OR Tree

9.1.2 Make Determinate

We now go onto the key problem of making our programs determinate. That is, if they succeed, then they succeed precisely once unless we *really* want them to generate alternative solutions. Many programmers find taming backtracking to be a major problem.

Consider the problem raised by this program:

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

together with the goal

```
?- sum(2,X).
```

The meaning of **sum/2** is that, for the first argument N (a positive integer), there is some integer, the second argument, which is the sum of the first N positive integers.

We *know* that, for the **mode sum(+,-)**, there is only one such result. Therefore, if we try to **redo** a goal such as **sum(2,Ans)** it should fail. We could test that this is so with:

```
?- sum(2,Ans),write(Ans),nl,fail.
```

We would like the result:

```
3
no
```

Alas, here is the result using Edinburgh **Prolog**.

```
3
(a very very long wait)
```

We have a runaway recursion. Figure 9.2 shows the execution tree for the goal **sum(2,Ans)**. Now look at the goal:

```
?- sum(2,X),fail.
```

and the resulting fragment of the execution tree which is shown in figure 9.3. **Prolog** goes into a non terminating computation. We want to make sure that, having found a solution, **Prolog** never looks for another solution via **Redo**ing the goal. Figure 9.4 shows the consequence when the cut (**!/0**) is used.

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

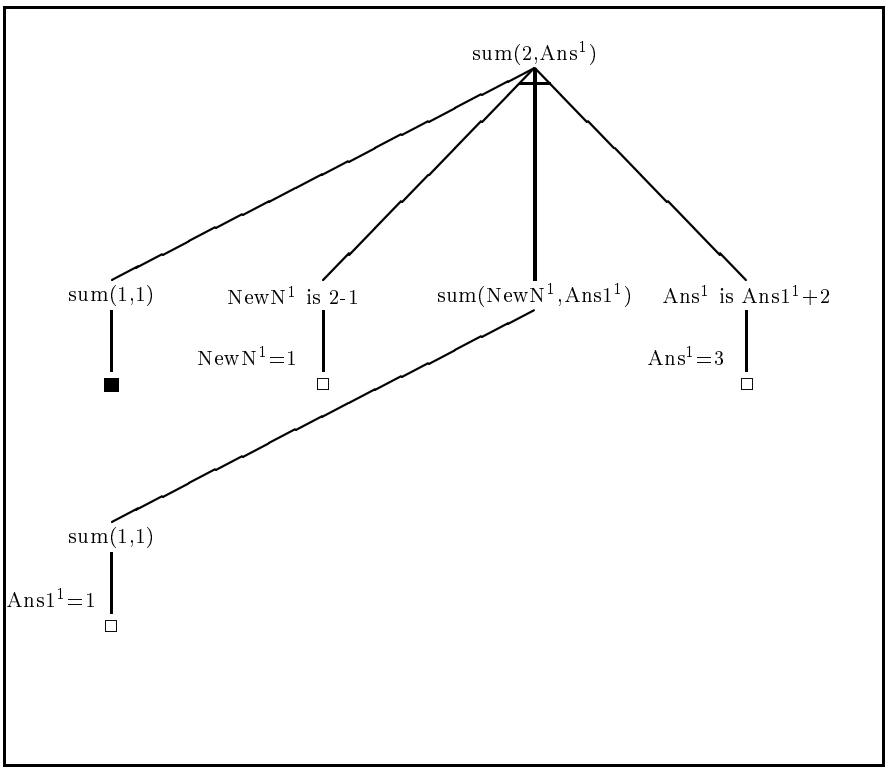


Figure 9.2: The First Solution to the Goal **sum(2,Ans)**

9.1.3 Fail Goal Now

We are trying to solve the problem that arises when we realise, in the middle of satisfying subgoals for some goal, that the goal will never succeed —even if we try other clauses which have heads that unify with the goal.

Here is a way of defining **woman/1** in terms of **man/1** where we base the idea that, in trying to establish that someone is a "woman", we prove that they are actually a "man" and there is therefore no point in trying to find some other proof that this person is a woman.

```
woman(X):-
                man(X),
                !,
                fail.
woman(X).
```

Putting it a slightly different way, to solve for **woman(jim)** we try **man(jim)**. If that succeeds then we want to abandon the attempt to prove **woman(jim)** without trying any other clauses for **woman/1**.

Note that the use of the **cut** (**!/0**) stops any attempt to resatisfy **man/1** once backtracking is forced through **fail/1** failing. Note also that the second clause for **woman/1** will not be used after the *cut—fail* combination has been met.

We call this use of **cut** in conjunction with **fail/0** the *cut—fail* technique.

The above code for **woman/1** is a special case of **Prolog**'s implementation of

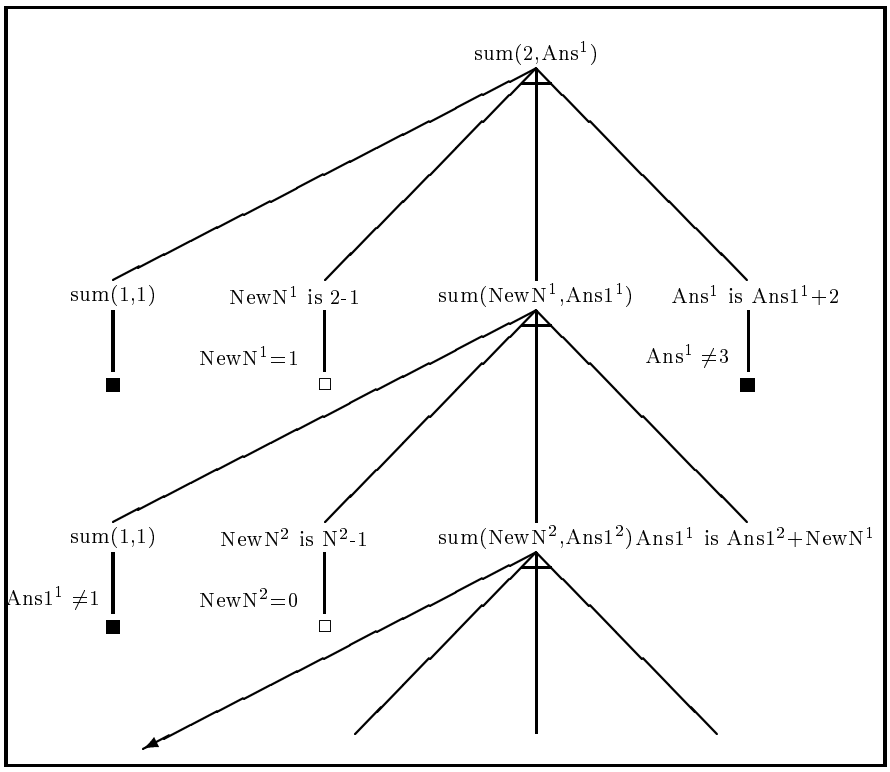


Figure 9.3: Resatisfying the Goal **sum(2,Ans)**

negation as failure. Here is a possible definition of **\+/1** using cut (**!/0**) and **call/1**.

```
\+(Goal):-
                call(Goal),
                !,
                fail.
\+(Goal).
```

9.2 Changing the Program

The use of cut (**!/0**) changes the search space while the program is running. We now introduce a family of predicates that can be used to change the search space during program execution. We do this with the strongest request:

> Never use these predicates unless you really have to do so

9.2.1 Do Not Do It!

The **Prolog** database is the set of clauses loaded into **Prolog** via **consult/1** or **reconsult/1** (these predicates can also be used at run-time so they are subject to the same strictures as the rest described below).

If, during run-time, a new clause is introduced into the **Prolog** database then this can change the behaviour of the program as, often, the program's meaning changes.

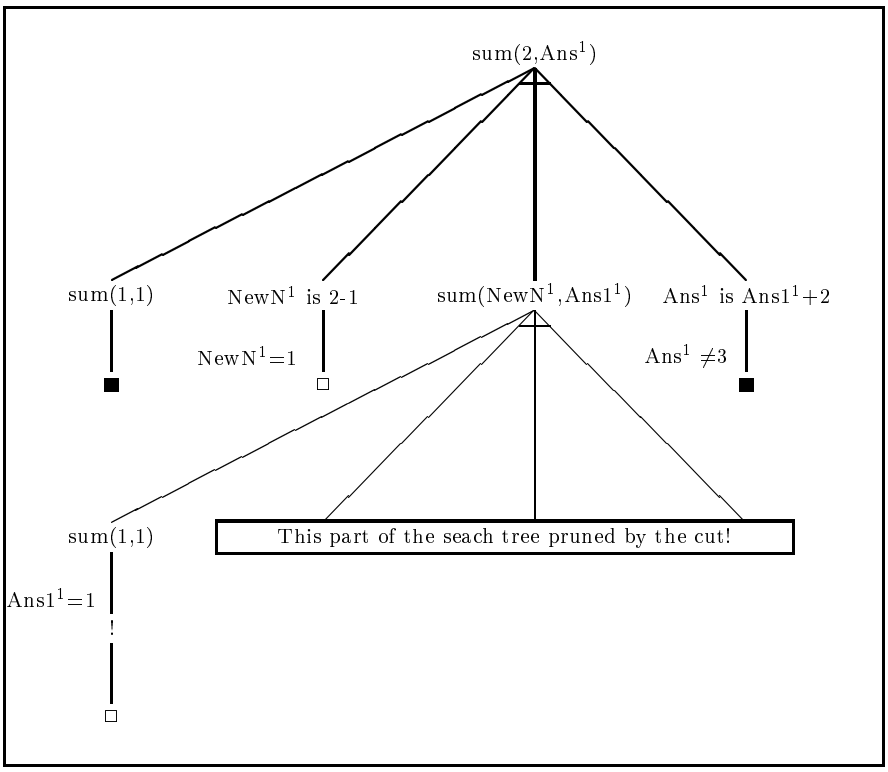


Figure 9.4: The Effect of the cut on the Goal **sum(2,Ans)**

The predicates that we refer to are as follows:

Program Modifying Predicates	
assert(C)	Assert clause C
asserta(C)	Assert C as first clause
assertz(C)	Assert C as last clause
retract(C)	Erase the first clause of form C
abolish(Name,Arity)	Abolish the procedure named F with arity N

Note that all the predicates except **retract/1** are *determinate*. They are not resatisfiable. The predicate **abolish/2** has **mode abolish(+,+)** while the predicate **retract/1** can be used with **mode retract(-)**. This latter predicate can therefore be used to 'wipe out' a complete program as in:

```
?- retract(X),fail.
```

This will fail with the side-effect of removing all the clauses loaded. We can remove just some clauses as in:

```
?- retract(foo(1,X)).
```

will remove all clauses whose heads unify with **foo(1,X)**.

Note that to add a clause which is also a *rule* you will need to write **assert((a:-b))** and not **assert(a:-b)**. See chapter 10 for an explanation.

Together, these predicates can be used to implement global flags and a form of global variable. This almost always makes it harder to understand individual parts of the program —let alone the disastrous effect such changes have on the declarative reading of programs.

All these predicates are *side-effect*ing. Therefore, backtracking will not undo these side-effects. For example, if **assert/1** is used to maintain a database of results found so far then, on backtracking, **Prolog** will not remove these results.

Further, the program becomes sensitive to interupts. It has been known for someone to abort a program (using **^C** and then **a** for abort) between the **assert**ing of a new clause and the **retract**ing of an old clause —leaving an unexpected old clause around which interfered badly with the subsequent execution of the program.

If a problem seems to require the use of **assert/1** then, usually, there is another way of doing things.

9.2.2 Sometimes You have To!

There are one or two occasions when you might want to use these predicates. The main one is when you have *definitely* proved that something is the case. That is, there is no way in which some statement (added to the program as a clause) can be false. Sometimes, of course, a program is supposed to modify the **Prolog** database. For example, **consult/1** and **reconsult/1**.

Often, we do not want to modify the program *itself* —rather, we want to change the data the program accesses. There is a facility in Edinburgh **Prolog** known as the *recorded database*. This is a way of storing **Prolog** terms under a *key*. Such terms are hidden from the **listing/0** program. The predicates that access this recorded database are:

Program Modifying Predicates	
erase(R)	Erase the record with reference R.
record(K,T,R)	Record term T under key K, reference R.
recorda(K,T,R)	Make term T the first record under key K, reference R.
recorded(K,T,R)	Term T is recorded under key K, reference R.
recordz(K,T,R)	Make term T the last record under key K, reference R.

These can be used to squirrel away information to be used by the program itself. An example is the predicate **random/2**:

```
random(Range,Num):-                              % to choose random number in range
        recorded(seed,Seed,Ref),                 % get seed from database
        erase(Ref),                              % delete old value of seed
        Num is (Seed mod Range) + 1,             % fit seed into range
        NewSeed is (125*Seed+1) mod 4093,        % calculate new value
        record(seed,NewSeed,_Ref).               % and assert it into database
```

This shows how we can maintain information about the seed used to generate the next pseudo-random number. Note that, unless we want to delete an entry (using **erase/1**) we usually use an *anonymous variable* for the record reference.

Using this family of predicates is more elegant (and sometimes more efficient) but suffers from the same problems as the **assert** family.

9.3 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

> You should be able to use the cut (!) to implement the techniques of *commit*, *make–determinate* and *fail–goal–now* (*cut—fail*).
> You should know how to change the program at run-time and understand the dangers in doing so. You should know some of the circumstances when it is acceptable to do so.
> You should know how to use the *recorded* database.

Exercise 9.1 *1. Given the following clauses, it is required to place cut(s) in the program to achieve the given outputs: First, determine what the output will be without placing any cuts in the program.*

```
female_author:-
                author(X),
                write(X),
                write(' is an author'),
                nl,
                female(X),
                write(' and female'),
                nl.
female_author:-
                write('no luck!'),
                nl.

author(X):-
                name(X).
author(X):-
                write('no more found!'),
                nl,
                fail.

name(sartre).
name(calvino).
name(joyce).

female(murdoch).
female(bembridge).
```

and here are the desired outputs. Make sure that you use only one cut *to get the desired output.*

(a)
```
sartre is an author
no more found!
no luck!
```

(b)
```
sartre is an author
calvino is an author
```

joyce is an author
no more found!

(c) *sartre is an author*
no luck!

(d) *sartre is an author*

(e) *sartre is an author*
calvino is an author
joyce is an author
no luck!

2. *Here is an example of code taken from one of the standard* **Prolog** *libraries —only all the cuts have been removed! Try to put them back.*

```
delete([], _, []).
delete([Kill|Tail], Kill, Rest) :-
                    delete(Tail, Kill, Rest).
delete([Head|Tail], Kill, [Head|Rest]):-
                    delete(Tail, Kill, Rest).
```

   The semantics is roughly "remove the element named in the second argument from the list in the first argument to produce the list in the third argument (which does not contain any copies of the element to be removed)".

   Therefore, the first two arguments are supposed to be inputs and the third an output. Note that the predicate must be determinate so that, if asked to **Redo**, *it will* **fail**.

3. *Define a predicate* **disjoint/1** *which is true only when the argument to* **disjoint/1** *contains no repeated elements. Make sure that the predicate is* determinate.

   Now use the cut—fail *method to define the same predicate.*

4. *Try writing* **plus/3** *which declares that "the first two arguments add up to the third argument provided all the instantiated arguments are integers". If, however, less than two argument are not integers then the predicate should fail and print out some pleasing error message.*

   Note that this is not equivalent to "Z is X + Y" and get the cuts in!

Chapter 10

Prolog Syntax

We describe **Prolog** syntax more formally.
We introduce the concept of a **Prolog** term, a variation of the logical variable and arbitrarily nested terms.
We explain how two **Prolog** terms are unified and demonstrate the need for a special check to ensure that we do not get infinite datastructures.
We show that lists are also terms and illustrate how to concatenate two lists together.
We also show that the structure of every **Prolog** clause is also a **Prolog** term.

Prolog Terms are one of:

- Constant
- Variable
- Compound Term

10.1 Constants

A Constant is one of:

- Atom
- Integer
- Real Number

Atoms are made up of:

- letters and digits: AB...Zab...z01...9 and _ (underscore)
- symbol: any number of +, -, *, /, \, ^, <, >, =, ~, :, ., ?, @, #, $ &
- quoted strings: 'any old character' —but the single quote character is handled specially

Normally, *atoms* start with a lower case letter. Note that, in a quoted atom, you can include a "'" by prefixing it with another " ' ". So, to print a "'" on the screen you will need a goal like **write('''')**.

10.2 Variables

Variables usually start with a capital letter. The only interesting exception is the special *anonymous variable* written _ and pronounced "underscore". In the rule

```
process(X,Y):-
                generate(_,Z),
                test(_,Z),
                evaluate(Z,Y).
```

the underscores refer to *different* unnamed variables. For example, here are two versions of member/2.

```
member(X,[X|Y]).
member(X,[Y|Z]):-
                member(X,Z).

member(X,[X|_]).
member(X,[_|Z]):-
                member(X,Z).
```

Note that, in the clause,

```
know_both_parents(X):-
                mother(_,X),
                father(_,X).
```

the underscores do not refer to the same object. The reading is roughly that "we know both the parents of **X** if someone(name unimportant) is the mother of **X** and someone else (unimportant) is the father". Note that **Prolog** regards the two occurrences of the *anonymous variable* in the above as different variables.

10.3 Compound Terms

A *Compound Term* is a *functor* with a (fixed) number of *arguments* each of which may be a **Prolog** *term*.

This means that we can arbitrarily nest compound terms. For some examples:

```
happy(fred)
                principal functor =    happy
                1st argument      =    a constant (atom)

sum(5,X)
                principal functor =    sum
                1st argument      =    constant (integer)
                2nd argument      =    variable
```

```
not(happy(woman))
                principal functor =    not
                1st argument      =    compound term
```

Nesting compound terms may be of use to the programmer. For example, the clause

```
fact(fred,10000).
```

is not as informative as

```
fact(name(fred),salary(10000)).
```

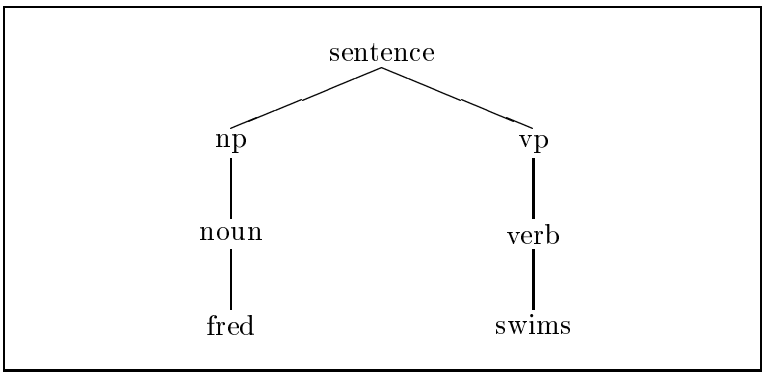
which can be thought of as defining a PASCAL-type record structure.

10.4 (Compound) Terms as Trees

Take the compound term

```
sentence(np(noun(fred)),vp(verb(swims)))
```

and construct a tree. Start by marking the root of the tree with the principal functor and draw as many arcs as the principle functor has arguments. For each of the arguments, repeat the above procedure.



10.5 Compound Terms and Unification

Consider

```
?- happy(X)=sad(jim).
```

—fails, because we know that it is necessary that the principal functors and their arities are the same for unification to succeed.

```
?- data(X,salary(10000))=data(name(fred),Y).
```

—succeeds, because, having matched the principal functors (and checked that the arities are the same) we recursively try to match corresponding arguments. This generates two subgoals:

```
X = name(fred)
salary(10000) = Y
```

which both succeed.

10.6 The Occurs Check

This is an aside. If we try to unify two expressions we must generally avoid situations where the unification process tries to build infinite structures. Consider:

```
data(X,name(X)).
```

and try:

```
?- data(Y,Y).
```

First we successfully match the first arguments and Y is bound to X. Now we try to match Y with name(X). This involves trying to unify name(X) with X. What happens is an attempt to identify X with name(X) which yields a new problem —to match name(X) against name(name(X)) and so on. We get a form of circularity which most **Prolog** systems cannot handle.

To avoid this it is necessary, that, whenever an attempt is made to unify a variable with a compound term, we check to see if the variable is contained within the structure of the compound term.

This check is known as the *occurs check*. If we try to unify two terms and we end up trying to unify a variable against a term containing that variable then the unification should fail.

Most **Prolog** implementations have deliberately missed out the *occurs check* —mostly because it is computationally very expensive.

Consequently, the goal **X=f(X)** will usually succeed where it should really fail. The most common way in which this error might manifest itself is when the system tries to print out the binding for **X**. This usually results in an attempt to print an infinite term.

```
?- X=f(X).
X=f(f(f(f(f(f(f(f(f(f(f(f(f...
```

10.7 Lists Are Terms Too

If a list is a term then it must be a compound term. What, then is its principal functor? Predicates have a fixed arity but lists can be any length —so what is the arity of the principle functor?

For the moment only, let us suppose we have a gluing agent which glues an element onto the front of a list. We know this is a reasonable supposition because we already have a list destructor/constructor that works like this.

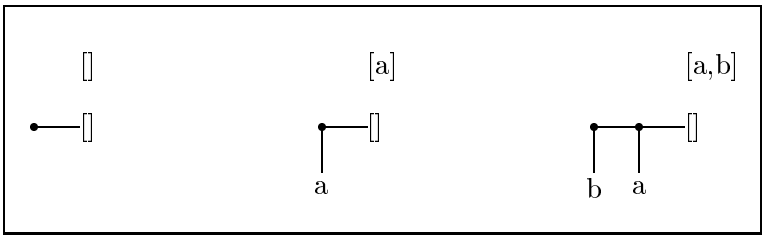
```
[a,b,c,d] = [Head|Tail]
—results in Head=a, Tail=[b,c,d]
```

We might think of this constructor as a predicate **cons/2**. We have to build lists like this. Note, however, that there is no built-in predicate named **cons/2** —the real name for the list constructor function is **./2**!

In **Prolog**, the empty list is represented as []. In some implementations, the empty list is named "nil" —but the **Prolog** you will use does not use this name.

Familiar List Notation	Intermediate Form	Compound Term Form
[ ]		[]
[a]		cons(a,[])
[b,a]	cons(b,[a])	cons(b,cons(a,[]))
[c,b,a]	cons(c,[b,a])	cons(c,cons(b,cons(a,[])))

Now to represent the lists as trees —but we will distort them a little:

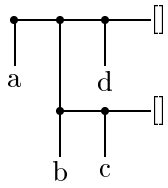


You will have noticed that we could have written **cons** where we have written **.** —well, remember that **Prolog** doesn't use a meaningful name for the constructor **cons/2**. Really, the constructor is **./2**. For (textual) explanation purposes, we shall stick to using **cons/2**.

Now we will show how to unpack the structure of a non-flat list. We do this by building up the structure from left to right.

```
[a,[b,c],d]
      goes to
cons(a,[[b,c],d])
      goes to
cons(a,cons([b,c],[d])
      goes to
         now [b,c] is              cons(b,[c])
         that is,                  cons(b,cons(c,[]))
cons(a,cons(cons(b,cons(c,[])),[d])
      goes to
cons(a,cons(cons(b,cons(c,[])),cons(d,[])))
```

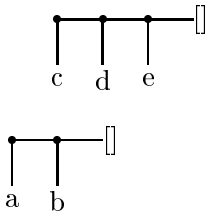
As this is difficult to read, we construct a tree using the method for drawing trees of compound terms.



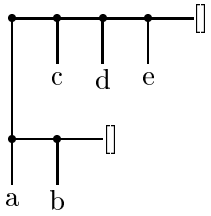
10.8 How To Glue Two Lists Together

We want to 'glue', say, **[a,b]** to **[c,d,e]** to give the result **[a,b,c,d,e]**. That is, we want a predicate **append/3** taking two lists as input and returning the third argument as the required result.

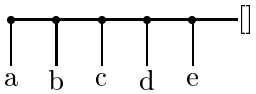
Here are the two lists as trees:



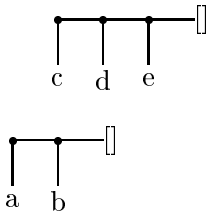
You might think of checking to see whether **cons([a,b],[c,d,e])** correctly represents the list **[a,b,c,d,e]**. Look at this 'solution' as a tree.



It is not the required



Let's try again:



We could solve our problem in a procedural manner using our list deconstructor as follows:

> Lop off the head **a** of the first list [**a**,**b**]
> Solve the subproblem of gluing [**b**] to [**c**,**d**,**e**]
> Put the head **a** back at the front of the result
>
> But we have a subproblem to solve:
>
> Lop off the head **b** of the first list [**b**]
> Solve the subproblem of gluing [] to [**c**,**d**,**e**]
> Put the head **a** back at the front of the result
>
> But we have a subproblem to solve:
> Gluing [] to [**c**,**d**,**e**] is easy..the result is [**c**,**d**,**e**]

First thing to note is that there is a recursive process going on. It can be read as:

> Take the head off the first list and keep it until we have solved the subproblem of gluing the rest of the first list to the second list. To solve the subproblem simply apply the same method.
>
> Once we are reduced to adding the empty list to the second list, return the solution —which is the second list. Now, as the recursion unwinds, the lopped off heads are stuck back on in the correct order.

Here is the code:

```
append([],List2,List2).
append([Head|List1],List2,[Head|List3]):-
                append(List1,List2,List3).
```

10.9 Rules as Terms

Consider:

```
happy(X):-
                rich(X).
```

If this is a term then it is a compound term. Again, what is its principal functor and its arity?

1 Principal Functor is

:-

Usually, the functor is written in infix form rather than the more usual prefix form.

2 Arity is

2

3 The above rule in prefix form

:-(happy(X),rich(X)).

But what about

```
happy(X):-
                healthy(X),
                wealthy(X),
                wise(X).
```

Trying to rewrite in prefix form:

:-(happy(X),whatgoeshere?).

Note that the comma ',' in this expression is an argument separator. In the definition of **happy/1** above, the commas are read as "and".

Yes,

healthy(X),wealthy(X),wise(X).

is also a compound term with principal functor

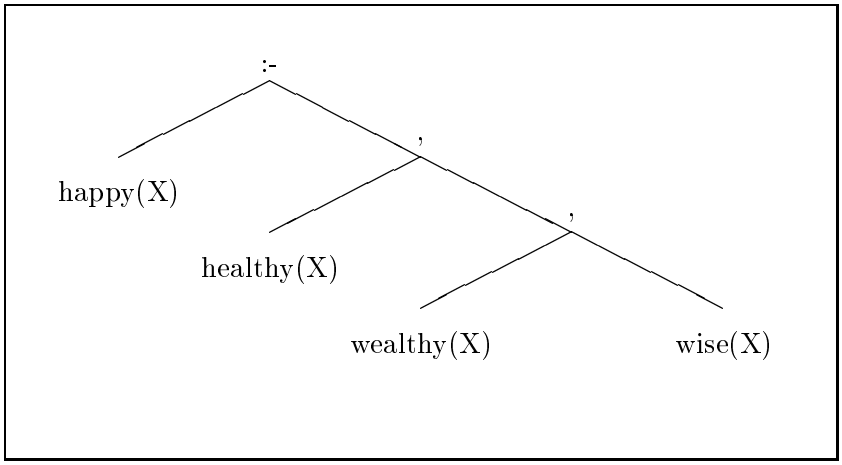
,

and arity 2. Since we have to represent three subgoals and the arity of ',' is 2 we again have a nested compound term. The correct prefix form for the example is:

',' (healthy(X),',' (wealthy(X),wise(X))).

Note: try the goal **display((healthy(X),wealthy(X),wise(X)))** to see the "truth". Also, note that, for a reason as yet unexplained, you need an extra pair of brackets around the goal you want printed via **display/1**.

Here is the tree:



10.10 What You Should Be Able To Do

You should be able to use the anonymous variable correctly.
You should know how to form **Prolog** atoms.
You should be able to construct a tree to represent any compound term —including lists and rules.
You should be able to determine whether or not two **Prolog** terms unify.
You should know what the occurs check is for and when it should be used.

Another Interlude: Input/Output

We describe how to make use of input and output streams.
We show how to read from files and write to files.
We describe how to read individual **Prolog** terms and how to build a 'consult' predicate.
We illustrate the development of several example programs to demonstrate how to write **Prolog** programs.

We discuss a number of practical issues.

Testing a Predicate

Suppose that we want to test the predicate **double/2** to see if it works for its intended inputs.

```
double(X,Y):-
                Y is 2*X.
```

To do this, we write a *test* predicate:

```
test:-
                read(X),
                double(X,Y),
                write(Y),
                nl.
```

Here is a transcription of executing the query **test**:

```
?- test.
|: 2.
4
yes
```

Note that, since we are using **read/1** which only accepts valid **Prolog** terms terminated by a "." followed by **Return** (in this case), we have to enter input integers as **2.**!

Now to make this into a loop. The easy way is to recursively call **test/0**. We would prefer, however, to put in a test so that we can abort the loop. This requires an *end-of-input* marker.

```
test:-
        read(X),
        \+(X = -1),
        double(X,Y),
        write(Y),
        nl,
        test.
```

When we input the *end-of-input* marker (-1) we backtrack to **read/1** which fails (for this **Prolog** implementation!) and **test/0** fails as there are no other clauses. We could always add a second clause (after —not before) which guaranteed that the goal **test** succeeded once the *end-of-input* marker was met.

> Note that it is up to us to make sure that **read/1** is never asked to process non-integer inputs. We could always define and use our own **read_integer/1** to catch non-integer input.

Input/ Output Channels

The **standard input** stream is taken from the keyboard and is known as "user".

Think of the stream of characters typed in as issuing from a file called "user".

The **standard output** stream is directed to the terminal screen and is known as "user" too.

Think of the stream of characters issuing from **Prolog** as going to a file called "user".

Input/ Output and Files

Let us take our input data from a file called "in".

```
go:-
        see(in),
        test,
        seen.
```

We *wrap* the **test/0** predicate into a predicate **go/0** which takes input from the specified file "in". This file should contain legal **Prolog** terms —for the predicate **double/2** we want something like:

```
2.
23.
-1.
```

Facilities for Redirecting Input	
see/1	Take input from the named file
seen/0	Close the current input stream and take input from user
How do you find out what the current input stream is?	
seeing/1	Returns name of current input stream

Now to redirect output to a file named "out":

```
go:-
        tell(out),
        see(in),
        test,
        seen,
        told.
```

Using the same file "in" as previously, "out" will contain:

```
4
46
```

Facilities for Redirecting Output	
tell/1	Send output to the named file
told/0	Close the current output stream and send output to user
How do you find out what the current output stream is?	
telling/1	Returns name of current output stream

The End of File Marker

When **read/1** encounters the end of a file it returns the **Prolog** atom

```
end_of_file
```

So we can rewrite **test/0**:

```
test:-
        read(X),
        \+(X = end_of_file),
        double(X,Y),
        write(Y),
        nl,
        test.
```

and now we have our *end-of-input* marker as the atom **end_of_file**.

Input of Prolog Terms

Both **consult/1** and **reconsult/1** have been described in chapter 5.5. **Prolog** will try to read a clause at a time from the named file. So any error message only refers to the current term being parsed.

Of course, if **Prolog** cannot find the end properly then we have problems. The **Prolog** you are using will load all clauses that parse as correct and throw away any ones that do not parse.

Some example problems: the first is where we have typed a ',' instead of a '.'.

```
a:-                          a:-
        b,                           b,
        c,      is read as           c,
d:-                          d:-e.
        e.
```

There are problems with this reading which will be reported by **Prolog**. Here is another problem caused by typing a '.' for a ','.

```
a:-                          a:-
        b.                           b.
        c,      is read as   c,d:-e.
d:-
        e.
```

This is basically illegal as we are seen to be trying to insert a clause defining **,/2** into the **Prolog** database.

Defining Your Own Consult

For this, we need some additional information about the *side-effecting* predicate **assert/1**. Note that you should make use of this predicate as little as possible. If tempted to use it, **think again**.

The predicate **assert/1** takes a legal **Prolog** clause as its argument. A call with a legal argument will always succeed with the *side-effect* of inserting the clause in the database —usually, at the end of any clauses with the same principle functor and arity (there is a variant, **asserta/1**, which can be used to position a new clause for a predicate at the beginning).

Essentially, we redirect input to a named file, read a clause, assert it and recurse.

```
my_consult(File):-see(File),
                  my_read(X),
                  my_process(X),
                  seen.
my_process(X):-
                  \+(X=end_of_file),
                  my_assert(X),!,
                  my_read(Y),
                  my_process(Y).
my_process(X):-
```

```
                    \+(X=end_of_file),
                    my_read(Y),!,
                    my_process(Y).
my_process(end_of_file).
my_read(X):-
                    read(X),!.
my_read(X):-
                    my_read(X).
my_assert(X):-
                    assert(X).
```

There are some subtleties here. We have to consider various problems with, inevitably, different treatments.

The first problem is that of syntactically incorrect input. To handle this, we have defined a resatisfiable form of **read/1**. The predicate **my_read/1** is designed so that, if **read/1** fails, we just try again. Since **read/1** has the *side-effect* of throwing away the offending input, we can have a go with another chunk of input. This mimics the behaviour of **consult/1**.

The second problem is to make sure that **end_of_file** is treated properly —we do not want to insert it into our database nor do we want to force backtracking to take place back into **my_read/1**! The simplest solution is to realise that we only want to keep resatisfying **my_read/1** if **read/1** fails owing to a syntactic error. Once **read/1** succeeds we would like to be *committed*. Hence we use case selection in **my_process/1** making use of **\+/1**. This means that, on encountering **end_of_file**, we will use the third clause of **my_process/1**.

There is a third problem which this procedure can handle. There are syntactically correct **Prolog** terms which are not legal **Prolog** clauses. For example, **a,b:-c.** is a legal term but not a legal clause. The predicate **my_assert/1** will fail and we will then try the second clause of **my_process/1** which will pick up some more input and try to handle that. The cut (**!/0**) is needed in the first and second clauses of **my_process/1** because we are certain that if we have successfully 'processed' a clause then we are *committed* from there on.

There is a fourth problem. If there is a query (or directive) in the file consulted such as **?- write(hello)** then we do *not* want to assert this clause —we want to issue some goal to the **Prolog** interpreter. This could be handled by two extra clauses for **my_assert/1**. One of these would be **my_assert((?- X)):- !,call(X).** Fixing this program to deal with this fourth difficulty can be left as an exercise for the reader (again).

The fifth problem is to write your own version of **reconsult/1**. This is a little trickier.

The sixth problem is not immediately obvious —but remember that **Prolog** converts a grammer rule like **s --> np,vp** into something like **s(S,S0):- np(S,S1), vp(S1,S0)**. Therefore, we ought to arrange to handle this.

In reality there is one further problem. It is possible to write one's own transformation rule to turn some legal **Prolog** clause into another one using **term_expansion/2**. This, however, can be hidden inside the call to the predicate that transforms grammar rules.

What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to write a program to read input from one file and write output to another file.
You should also understand something of how the **Prolog** *consult-loop* works and (possibly) be able to write your own version.

Chapter 11

Operators

We describe some familiar *operator*s.
We define the three forms which they may take.
We introduce and describe the notions of *operator precedence* and *operator associativity*.
We then describe how to define new operators and then how to parse complex terms containing several user-defined operators.

An operator is a predicate which has some special properties.

Here is a list of ones we have met already:

$+$	$-$	$*$	$/$
$<$	$=<$	$>$	$>=$
$=$	is	\+	
,	$-->$	$:-$	$?-$

Note that **\+/1** is an operator. So we can write **\+(man(jim))** as **\+ man(jim)**.

11.1 The Three Forms

11.1.1 Infix

Here are some examples of arithmetic expressions that use infix operators:

$$3+2 \quad 23-2 \quad 8*2 \quad 30/2 \quad 2<7 \quad 6>2 \quad Yis\ 23$$

All the infix operators used in the above are necessarily *binary* operators —*i.e.* they have an arity of 2. Each of the above terms can be rewritten in 'regular' **Prolog** syntax as

$$+(3,2) \quad -(23,2) \quad *(8,2) \quad /(30,2) \quad <(2,7) \quad >(6,2) \quad is(Y,23)$$

Remember that the use of the inequality operators requires that both arguments are evaluated before unification is applied. For **is/2**, only the second argument is evaluated before unification is applied.

Here are some examples of infix operators used in the basic syntax of **Prolog** clauses.

healthy(jim), wealthy(fred) adjective --> [clever] a:- b

These infix operators are also binary. Here are their regular forms.

','(healthy(jim), wealthy(fred)) -->(adjective, [clever]) :-(a,b)

Note how the functor **,/2** has to be 'protected' with single quotes as in ','.

11.1.2 Prefix

Some expressions using prefix operators:

\+ man(jane) + 23 - 12

and here are the equivalent regular expressions:

\+(man(jane)) +(23) -(12)

Inevitably, prefix operators are associated with unary predicates —-*i.e.* they have an arity of 1.

11.1.3 Postfix

There are no predefined postfix operators but this one might have existed!

X is_a_factorial

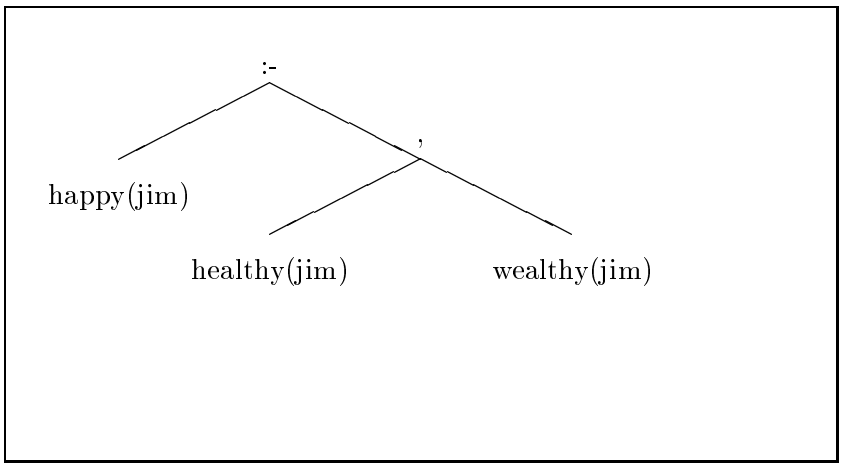
If it had then it would have been writable in the regular form **is_a_factorial(X)**. As with prefix operators, postfix operators have an arity of 1.

11.2 Precedence

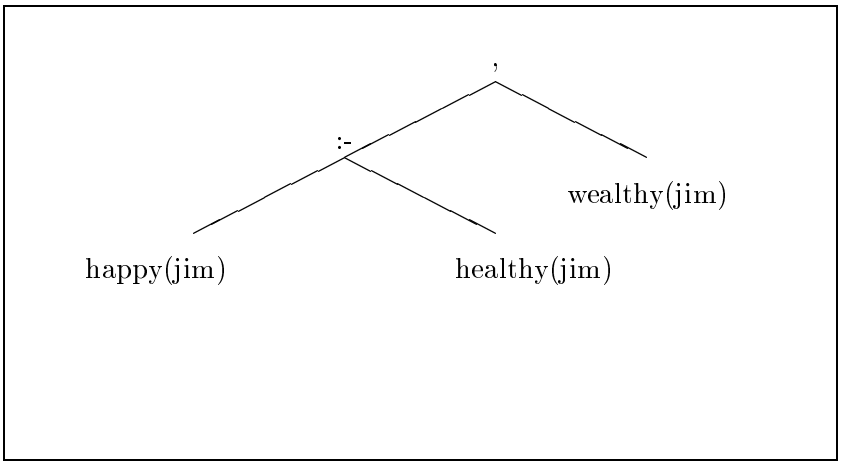
We will now look at the structure of some **Prolog** expressions:

```
happy(jim):-
                healthy(jim),
                wealthy(jim).
```

We assume that it is always possible to represent a **Prolog** expression as a tree in an unambiguous way. Is this



which corresponds to **happy(jim):- (healthy(jim),wealthy(jim))** or



which corresponds to **(happy(jim):- healthy(jim)),wealthy(jim)**. We can see that the first version is the one we have taken for granted. We describe this situation by saying that **,/2** binds *tighter* than **:-/2**.

This relates to the way we are taught to calculate arithmetical expressions in that we are told that we do multiplication before addition (unless brackets are used to override this). But there is another way to think of things: how to construct the expression tree. In this case, we choose the root to be the operator that is 'loosest' (in opposition to 'tightest' for computational purposes).

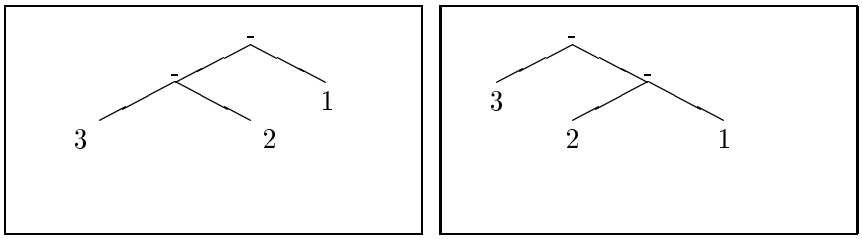
The issue is decided by *operator precedence*.

To construct a tree which describes a **Prolog** expression we first look for the operator with the highest precedence (this is in some sense the opposite of the way we compute a function). If this operator is an infix one, we can divide the expression into a left hand one and a right hand one. The process is then repeated, generating left and right subtrees.

Operator	Precedence
:-	1200
`-->`	1200
,	1000
`\+`	900
is	700
<	700
=	700
=<	700
>	700
>=	700
+	500
-	500
*	400
/	400

We still need to decide what to do with two operators of the same precedence. Should we regard

as one or the other of:



and, remember, that we are not yet talking about arithmetic evaluation!

We can use brackets to distinguish

(3 - 2) -1

from

3 - (2 - 1)

but we have a special way of distinguishing which interpretation we wish **Prolog** to make. In the above arithmetic example, the left hand tree has two subtrees hanging from the root "-". The left hand one has "-" as its root while the right hand one is not so allowed. We say that this interpretation of "-" is *left associative*.

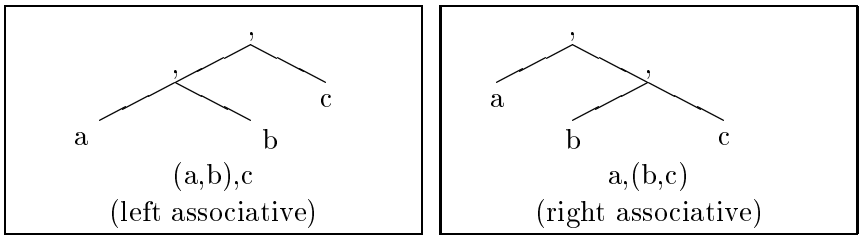
The normal interpretation of "-" is left associative. The common left associative operators are:

* / + - div [1]

Are there any *right associative* operators? Yes —consider how we are to disambiguate

a,b,c

where "a", "b" and "c" are all legal **Prolog** subgoals.



The answer is that **,/2** is right associative. Usually, we do not have to concern ourselves with the details of this.

In all the previous cases we have allowed exactly one subtree to have, as its root, the same operator as the "principal" root. We can extend this to permit operators of the same precedence. Thus, since "+" and "-" have the same precedence, we know that both operators in

[1] **div/2** is integer *div*ision. It is a synonym for **///2** —read this as an infix operator of arity 2 written //.

```
3 - 2 + 1
```

are left associative (and legal) and therefore the expression represents

```
(3 - 2) +1.
```

Sometimes, we do not wish to permit left or right associativity. For example, obvious interpretations of:

```
a:- b :- c
Y is Z+1 is 3
a --> b --> c
```

do not readily spring to mind. Therefore we make it possible to forbid the building of expressions of this sort.

11.3 Associativity Notation

11.3.1 Infix Operators

Left Associative	yfx
Right Associative	xfy
Not Associative	xfx

Note that "x" indicates that the indicated subtree must have, as its root, an operator of lower precedence than that of the root.

The "y" indicates that the root of the subtree may have the same precedence as the operator that is the root of the tree.

The "f" indicates the operator itself.

11.3.2 The Prefix Case

Here are a number of unary, prefix operators:

Operator	Precedence
:-	1200
?-	1200
\+	900
(unary) +	500
(unary) -	500

We regard a prefix operator as having only a right hand subtree. We must decide which of the above may be right associative. That is, which of the following make sense:

```
+ + 1
\+ \+ happy(jim)
:- :- a
```

Of these possibilities, we only accept **\+/1** as right associative.

11.3.3 Prefix Operators

Right Associative	fy
Not Associative	fx

11.3.4 Postfix Operators

As we have no examples here at the moment, here is the table:

Left Associative	yf
Not Associative	xf

11.4 How to Find Operator Definitions

It is possible to find out the associativity and precedence of any operator —whether it is a built-in one or a user-defined one— with the help of **current_op/3**. For example, here is how to find out about +:

```
?- current_op(X,Y,+).
X=500
Y=fx ;

X=500
Y=yfx
```

produces two solutions (if we ask for a further solution after the first one is found). The first solution is the precedence and associativity for unary + (in that order) and the second is for binary +. Note that you can get all the operators currently known with the help of a *failure-driven loop*:

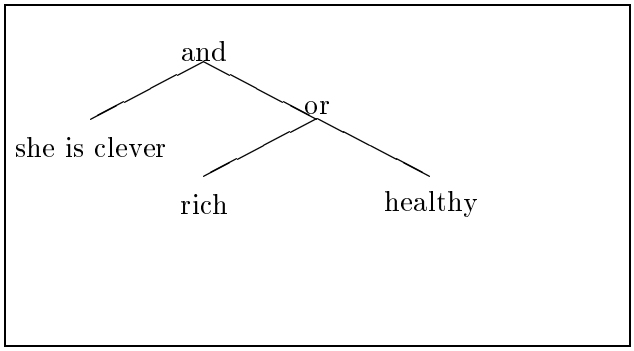
```
?- current_op(X,Y,Z),write_op(X,Y,Z),fail.
write_op(Precedence,Associativity,Operator):-
                write('Operator '),write(Operator),
                write(' has precedence '),write(Precedence),
                write(' and associativity '),write(Associativity),
                nl.
```

You will find some strange things amongst the 45 different operator declarations.

11.5 How to Change Operator Definitions

We will illustrate with an infix operator **and/2** and another **or/2**. We will choose the precedence of **and/2** to be greater than that of **or/2**. This means that we interpret:

```
she is clever and rich or healthy
```



Since **and/2** reminds us of **,/2** we will give it the same precedence and associativity:

Precedence	Associativity
1000	xfy

The required command is

```
op(1000,xfy,and).
```

The predicate **op/3** takes a precedence as its first argument, a legal associativity for its second argument and an operator name for its third argument. If given legal arguments, it succeeds with the *side-effect* of adding or changing an operator definition. You can even change the existing definitions —but, be warned, this can be dangerous.

We could also make it like **,/2** by *defining* and/2 as in:

```
X and Y :-
        call(X),
        call(Y).
```

Note that we have to have defined **and/2** as an operator *before* we can write the head of this clause as **X and Y**.

For **or/2** we choose *precedence* of 950 (less than **and/2**) and *associativity* of xfy (the same as **and/2**) with:

```
op(950,xfy,or)
```

and define it as equivalent to:

```
X or Y :-
        call(X).
X or Y :-
        call(Y).
```

11.6 A More Complex Example

We now try to represent data structures that look like:

if a and b or c then d

As we already have a representation for "a and b or c", this reduces to representing

if a then b

We will make "then" an infix operator of arity 2. Because both subtrees might contain **and/2** we will need to make **then/2** of higher precedence than **and/2** —say,1050 and not associative. Hence:

op(1050,xfx,then).

This means that "if" must be a prefix operator. As we do not wish expressions of the form

if if a

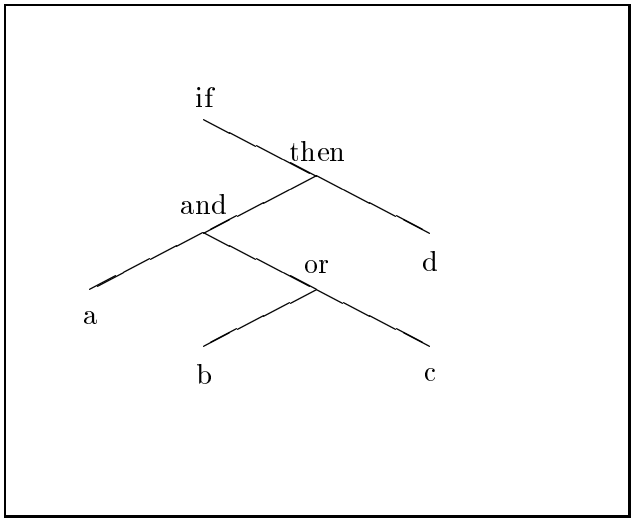
we must make **if/1** of higher precedence than **then/2** (say, 1075) and **if/1** must be non associative:

op(1075,fx,if).

We can now represent

if a and b or c then d

as the tree



or, as the **Prolog** term

```
if(then(and(a,or(b,c)),d))
```

This **Prolog** term is difficult to read but unambiguous while the representation using operators is easy to read but depends heavily on your understanding the precedences and associativities involved. All right if you wrote the code but the code is harder for someone else to read.

11.7 What You Should Be Able To Do

After finishing the exercises at the end of the chapter:

You should be able to parse a complex **Prolog** term that includes several built-in operators. You should be able to do the same thing with user-defined operators. You should be able to define your own infix, prefix and postfix operators.

Exercise 11.1 *Given the following declarations of precedence and associativity, express this clause as a tree.*

```
rule31: if      colour of wine          =white
           and body of wine             =light or body of wine=medium
           and sweetness of wine        =sweet or sweetness of wine=medium
        then wine                       =riesling confidence_factor 1000.
```

Operator	*Precedence*	*Associativity*
:	*975*	*xfy*
if	*950*	*fx*
then	*949*	*xfy*
and	*800*	*xfy*
or	*750*	*xfy*
confidence_factor	*725*	*xfy*
=	*700*	*xfx*
of	*595*	*xfy*

Chapter 12

Advanced Features

We describe predicates provided for examining terms. We show how to find all the solutions to a goal. We describe difference lists and illustrate their use We describe some aspects of good **Prolog** programming style. We summarise the extent to which programming **Prolog** is logic programming and mention some interesting developments.

We discuss some powerful features that **Prolog** offers then the important subject of programming style. Finally, some aspects of **Prolog** are mentioned that demonstrate that the development of Logic Programming is by no means over.

12.1 Powerful Features

12.1.1 Powerful Features —Typing

Prolog is a very weakly typed language. In some sense, the only type is the *term*.

Not all these features are first order predicate logic. Nevertheless they give great power into the hands of the programmer.

predicate/arity	succeeds	if the argument is
atom/1		atom
integer/1		integer
number/1		integer or real
atomic/1		atom or integer or real
var/1		uninstantiated variable
nonvar/1		not an uninstantiated variable

We demonstrate their use first by defining **type/2** which has **mode type(+,-)**. It takes a term as its first argument and returns a type for the term. On **redo**ing, it will attempt to find another type. To complicate the matter, we have specially distinguished *lists* —which are compound terms.

```
type(X,variable):-
                var(X),!.
type(X,atom):-
```

```
                atom(X).
type(X,integer):-
                integer(X).
type(X,real):-
                number(X),
                \+(integer(X)).
type(X,list):-
                nonvar(X),
                X=[_|_].
type(X,compound_term):-
                \+(atomic(X)),
                nonvar(X).
```

We have to use cut **!/0** in the first clause because, otherwise, we can generate spurious solutions for the goal **type(X,Y)**. There is one bug in the above — the goal **type(X,X)** succeeds with **X=atom**! This is not really wanted. How would you guard against this?

12.1.2 Powerful Features —Splitting Up Clauses

The first predicate we look at is good for 'picking up' clauses from the current **Prolog** database. The remainder are useful for destructing and constructing arbitrary **Prolog** terms.

clause/2

```
happy(X):-
                healthy(X),
                wealthy(X).
happy(jim).
```

The goal **clause(happy(X),Y)** produces

```
Y = healthy(X), wealthy(X)
```

on **redo**ing,

```
Y = true
```

Note the second answer returns a body of **true** for the clause **happy(jim)**.

For SICStus (and Quintus), the first argument of **clause/2** must specify at least the principle functor. That is, a call such as **clause(X,Y)** will fail.

However, for many **Prolog** systems, any calling pattern can be used: this means that we can also extract all the clauses which are facts with the goal **clause(X,true)**.

Before we show how to get round this limitation in SICSTUS, we illustrate with a simplified version of **listing/0** which we name **list/0**:

```
list:-
                clause(X,Y),
                write_clause(X,Y),
                fail.
list.

write_clause(X,Y):-
                write((X:-Y)),
                nl.
```

Now this can be made to work for SICStus by using **predicate_property/2**. This predicate can be called as in:

```
?- predicate_property(X,interpreted).
```

and **X** will be bound to the head of the first clause found that is "interpreted"[1]. So the amended code for **list/0** is:

```
list:-
                predicate_property(X,interpreted),
                clause(X,Y),
                write_clause(X,Y),
                fail.
list.
```

Note however that this fails to print the final '.' of a clause and that it also prints facts as if they were rules with their body equal to **true**. We can improve on this a little by changing **write_clause/2**.

```
write_clause(X,true):-
                write(X),
                write('.'),nl.
write_clause(X,Y):-
                \+(Y=true),
                write(X),
                write((:-)),nl,
                write_body(Y).

write_body(Y):-
                write(' '),
                write(Y),
                write('.'),nl.
```

Note that we have used **\+/1** to make the code *determinate*. If we wanted to put each subgoal on a separate line then we could rewrite **write_body/1**.

[1] If you have compiled your program then you now have a problem!

functor/3

```
?- functor(fact(male(fred),23),F,N).
F=fact
N = 2
```

The predicate **functor/3** can be used to find the principal functor of a compound term together with its arity. It can also be used to generate structures:

```
?- functor(X,example,2).
X = example(A,B)
```

except that the variables will be shown differently.

arg/3

```
?- arg(1,fact(male(fred),23),F).
F = male(fred)
```

The predicate **arg/3** is used to access a specified argument for some **Prolog** term.

As an example we will provide a predicate that uses side-effects, while taking apart an arbitrary **Prolog** term, to print some information about the term. It uses **type/2** as defined previously.

```
analyse(Term):-
                type(Term,Type),
                \+(Type=compound_term),
                \+(Type=list),
                write(Term,Type).
analyse(Term):-
                type(Term,compound_term),
                write(Term,compound_term),
                functor(Term,N,A),
                analyse_bit(0,A,Term).
analyse_bit(Counter,Counter,_):-
                !.
analyse_bit(Counter,Terminator,Term):-
                NewCounter is Counter +1,
                arg(NewCounter,Term,SubTerm),
                analyse(SubTerm),
                analyse_bit(NewCounter,Terminator,Term).

write(Term,Type):-
                write(Term),
                write(' is of type '),
                write(Type),nl.
```

The predicate **analyse/1** uses both **functor/3** to find the arity of a term and then uses **arg/3** to work through the various argument of the term one at a time. Note how we dive down into the substructure of a term before finishing the description of each of the arguments in the term. Lists, by the way, are not treated specially by **analyse/1**.

=../2

Now **=..** is pronounced "univ". It can be used to map a term onto a list in this way:

Term $\rightarrow$ [Functor, Arg_1, Arg_2, ... Arg_n]

For example, **=../2** can be used with **mode =..(+,+)** and **mode =..(+,-)**:

```
?- foo(12,fred)=.. [foo,12,fred].
yes

?- fact(male(fred),23)=.. X
X= [fact,male(fred),23]
```

The predicate can also be used with **mode =..(-,+)**.

```
?- X=.. [fact,male(fred),23].
X = fact(male(fred),23)
```

Here are some more examples:

```
?- (a + b) =.. X.
X = [+, a, b]

?- [a,b,c] =.. X.
X = ['.',a,[b,c]]
```

We demonstrate a real application where we have a predicate **triple_one/2** which takes as input an integer (first argument) and outputs (second argument) its triple. We are going to use **=../2** to triple each element of an input list. This will mimic the behaviour of a predicate **triple/2** previously used as an example. We define a predicate **map/3** which takes a predicate name as its first argument, the input list as the second argument and returns the output list as the third argument as in:

```
?- map(triple,[1,2,3],X).
X=[3,6,9]
```

We give the special case with the first argument as triple and then generalise it.

```
map(triple,[],[]).
map(triple,[H1|T1],H2|T2]):-
                X=.. [triple,H1,H2],
                call(X),
                map(triple,T1,T2).
```

The main trick is to assemble a term looking like **triple(H1,H2)** using **=../2** and then use **call/1** to execute the goal.

Now we replace the specific reference to **triple** and provide a more general version that can handle the task for arbitrary predicates of arity 2 —provided that they are defined to work with **mode** *predname*(+,-).

```
map(Functor,[],[]).
map(Functor,[H1|T1],H2|T2]):-
                X=.. [Functor,H1,H2],
                call(X),
                map(Functor,T1,T2).
```

The next task is to allow for an even more general version that can do the same sort of thing for predicates with an arity of more than two!

For example, define a predicate **npl/3** that takes a positive integer as first argument and a number as its second argument, returning the third argument as the second argument 'npled' as in:

```
?- nple(7,5,X).
X=35
```

We define **nple/3**:

```
nple(Multiplier,In,Out):- Out is Multiplier*In.
```

Now to look at the code. Now, we need to give the new version of **map/3** a first argument which contains the necessary info — *viz* the name of the predicate and the constant multiplier.

We can do this as the term **nple(N)** where **N** is the multiplier. We transform the term **nple(N)** into a list **[nple,N]** and then append the two arguments **H1** and **H2** using the standard **append/3**. This list is then rebuilt as the term **nple(N,H1,H2)** and then executed via **call/1**.

```
map(nple(N),[],[]).
map(nple(N),[H1|T1],[H2|T2]):-
                nple(N)=.. List,
                append(List,[H1,H2],NewList),
                X=.. NewList,
                call(X),
                map(nple(N),T1,T2).
```

Nowhere does this really depend on the arity of **nple(N)** —so we just replace the term **nple(N)** by **Term**.

```
map(Term,[],[]).
map(Term,[H1|T1],[H2|T2]):-
                    Term=.. List,
                    append(List,[H1,H2],NewList),
                    X=.. NewList,
                    call(X),
                    map(Term,T1,T2).
```

12.1.3 Powerful Features —Comparisons of Terms

There is a standard order defined on **Prolog** terms —*i.e.* one **Prolog** term can be compared with another and we can reach a decision about which comes before which. The predicates that achieve this are not part of the first order predicate logic. We only list them briefly here.

==/2

If you do *not* want to unify two **Prolog** terms but you want to know if the terms are strictly identical:

```
?- X == Y.
no

?- X=Y, X == Y.
yes
```

\==/2

This is equivalent to the **Prolog** definition

```
X \== Y:-
                    \+ (X == Y).
```

@>/2, @>=/2, @</2 and @=</2

These are the predicates that can be used to decide on the ordering of terms.

12.1.4 Powerful Features —Finding All Solutions

Remember that a *query* **foo(X)** is really asking something akin to whether (in predicate logic) $\exists$ **X foo(X)**. How do we ask $\forall$ **X foo(X)**? The answer, for situations where there are (obviously) a finite set of solutions is to use one of two special built-in predicates.

setof/3

The semantics for **setof/3** are unpleasant. It has to be used with care. We take this in three stages.

Stage 1

Suppose that we have these (and only these) facts about **knows/2**.

```
knows(jim,fred).
knows(alf,bert).
```

How do we find all the solutions of the form **knows(X,Y)**? Now the goal **knows(X,Y)** is equivalent to asking "does there exist some **X** and some **Y** such that **knows(X,Y)**". For all solutions we want to ask something like "for what set of values of **X** and set of values of **Y** is it true that for all **X** and all **Y** then **knows(X,Y)**".

```
setof([X,Y],knows(X,Y),Z).
Z = [[jim,fred],[alf,bert]]
```

where **Z** is the set of all solution pairs **[X,Y]** such that **knows(X,Y)**.

Stage 2

Now suppose we only want to gather the first element of the pairs.

```
?- setof(X,Y^knows(X,Y),Z).
Z = [jim, alf]
```

Wait a minute ... what is that Y^ bit? You have to *existentially quantify* any variables in which you are not interested if you are to get the set of all solutions and a reasonably clean semantics.

You have to read this as "find the set **Z** consisting of all values of **X** for which there exists a value **Y** for which **knows(X,Y)**". The **Y^** is interpreted as "there exists a **Y**" and is vital.

Stage 3

If you leave off the existential quantification the semantics of **setof/3** becomes conditional on the status of **Y** *at the time* the predicate is called.

```
foo(2,3).
foo(3,4).
foo(4,3).

?- setof(X,foo(X,Y),Set).
Set = [2,4]
```

In this case, **Set** is the set of all **X** for which there is a specific (but somewhat arbitrary) **Y** such that **foo(X,Y)**.

Note that the first argument is really a *variable pattern* which specifies which variables get put into the list of solutions and *how they are to appear*. For example:

```
?- setof(firstbit(X),Y^foo(X,Y),Set).
Set = [firstbit(2),firstbit(3),firstbit(4)]
```

Note also that any repeated solutions are removed and all the solutions are placed in a standard ordering.

bagof/3

The only difference between **bagof/3** and **setof/3** is that **bagof/3** leaves repeated solutions in the answer. Note that **bagof/3** is less expensive than **setof/3**.

Also note that, if there are no solutions then both **bagof/3** and **setof/3** fail! If you want a predicate that behaves like **setof/3** (or **bagof/3**) but succeeds with an empty list if there are no solutions then write something like:

```
all(X,Y,Z):-
                setof(X,Y,Z),
                !.
all(X,Y,[]).
```

which will behave in the desired way.

12.1.5 Powerful Features —Find Out about Known Terms

It is occasionally useful to find out various aspects of the system's knowledge —*e.g.* the known atoms that are not used by the system, the predicates defined by the user or the predicates defined by the system. We only mention these facilities in passing.

current_atom/1

```
?- current_atom([]).
yes
```

current_functor/2

Many **Prolog** systems implement this feature —but not SICStus.

```
?- current_functor(atom,atom(fred)).
yes
```

current_predicate/2

```
knows(fred).

?- current_predicate(knows,knows(fred)).
yes
```

current_op/3

```
?- current_op(1200,xfx,(:-)).
yes
```

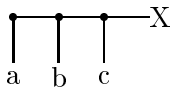
Note the use of brackets around **:-** to allow the term to be parsed correctly.

All the above can be used to generate information as well!

12.2 Open Lists and Difference Lists

We now briefly describe a valuable technique for programming in **Prolog**.

Consider the list **[a,b,c|X]**. We know the structure of the list *up to a point*.



If, at some point, we know that **X** is unbound then we say that we have an *open list*. We also say (informally) that **X** is a 'hole'.

Note that we are already familiar with what happens if we unify **X** with, say, **[d]**:

```
?- List=[a,b,c|X], X=[d].
List=[a,b,c,d]
```

Here, we started with an *open list* and 'filled' in the hole with the structure:

This results in a *proper* list (say) —the normal representation for a list. We generally think of a list processing procedure as taking a proper list as input and returning a proper list as output.

Now suppose that we realise that we do not have to represent the idea of a list as a proper list. There is nothing to stop us saying that we will represent a list of things as an open list. That is, we do this instead:

```
?- List=[a,b,c|X], X=[d|X1].
List=[a,b,c,d|X1]
```

and *partially* 'fill in' the 'hole' at the end of the list.

Now we can think of *open* list processing where we take an open list as input and return an open list as output.

Of course, if we have an open list as output we can always convert it into a proper list by 'filling in' the hole with the empty list (note that, in this case, we could fill in the hole with any proper list) —as in:

```
?- List=[a,b,c|X], X=[d,e,f].
List=[a,b,c,d,e,f]
```

Hang on a minute! We seem to be doing what **append/3** does here (with **mode append(+,+,-)**)! There is a difference, however, as the first argument is 'input' *partially* instantiated and is 'output' wholly instantiated!

If we had the first list expressed as an open list then all we have to do is to define a predicate that fills in the hole with the second list. Here is a very naive (and limited) definition of this sort of **append/3** —we shall call it **open_append/2**.

```
open_append([H1,H2,H3|Hole],L2):-
                    Hole=L2.

?- X=[a,b,c|Ho],open_append(X,[d,e,f]).
X=[a,b,c,d,e,f]
```

We have turned an open list into a proper list alright but in a limited way because our definition of **open_append/2** assumes that we have a list with three elements and the hole. We must improve on this.

If we want to reason about open lists then we often want to say something like "take the open list and fill in the hole with ...". Consequently, we would like to say that a certain term is an open list with such-and-such a hole. This suggests a new representation for the idea of a list —we represent a list of terms as an open list together with the hole.

This representation is known as a *difference* list —for a reason that will become apparent. Such a representation might be that the list of the terms **a**, **b** and **c** taken in order are represented by two terms —**[a,b,c|Hole]** and **Hole**. Now let us rewrite **open_append/2** as **difference_append/3**. We input the open list, the hole and the list to be appended.

```
difference_append(OpenList,Hole,L2):-
                    Hole=L2.
```

```
?- X=[a,b,c|Ho],difference_append(X,Ho,[d,e,f]).
X=[a,b,c,d,e,f]
```

This is better but we now will introduce a notation for difference lists. Since the list we are really interested in is always the open list without the hole we will represent difference lists like this:

```
[a,b,c,d|Hole] - Hole
```

Do not worry about the use of the minus operator —it carries connotations of subtraction but it is just a convenient uninterpreted (in this context) infix operator. We could easily define an operator of our own. Now the above can be rewritten as:

```
difference_append(OpenList-Hole,L2):-
                Hole=L2.

?- X=[a,b,c|Ho]-Ho,difference_append(X,[d,e,f]).
X=[a,b,c,d,e,f]-[d,e,f]
```

Whoops! Now we have returned a difference list but we are only really interested in the open list part —we want to lop off the hole. We redefine **difference_append/2** to be **difference_append/3**.

```
difference_append(OpenList-Hole,L2,OpenList):-
                Hole=L2.

?- X=[a,b,c|Ho]-Ho,difference_append(X,[d,e,f],Ans).
Ans=[a,b,c,d,e,f]
```

We are nearly there now. We have a strange version of **append/3** which takes a difference list as its first argument, a proper list as its second argument and returns a proper list.

We could live with this but let us be systematic and produce a version that appends a difference list to a difference list to return a difference list. Here is the first attempt to return a proper list given two difference lists:

```
difference_append(OpenList1-Hole1,OpenList2-Hole2,OpenList1):-
                Hole1=OpenList2.

?- X=[a,b,c|Ho]-Ho,difference_append(X,[d,e,f|Hole2]-Hole2,Ans).
Ans=[a,b,c,d,e,f|Hole2]
```

Note that we had to change the form of the second argument in order to represent the proper list **[d,e,f]** as a difference list.

We have returned an open list but we want a difference list. The first list has gained the hole of the second list. All we need to ensure is that we return the hole of the second list. Here we go again!

```
difference_append(OpenList1-Hole1,OpenList2-Hole2,OpenList1-Hole2):-
                Hole1=OpenList2.

?- X=[a,b,c|Ho]-Ho,difference_append(X,[d,e,f|Hole2]-Hole2,Ans).
Ans=[a,b,c,d,e,f|Hole2] - Hole2
```

Now we can recover the proper list we want this way:

```
?- X=[a,b,c|Ho]-Ho,difference_append(X,[d,e,f|Hole2]-Hole2,Ans-[]).
Ans=[a,b,c,d,e,f]
```

One more transformation can be made: you will note that all we are saying in the body of **difference_append/3** is that the hole of the first difference list has to be the open list of the second difference list.

```
difference_append(OpenList1-Hole1,Hole1-Hole2,OpenList1-Hole2).
```

We now have an extremely neat way of appending two difference lists together to get a difference list. Now, why bother?

Consider the question about how to add an element to the front of a list. This is easy because you can, for example, add **X=a** to the list **Y=[b,c,d]** as in **[X|Y]**. Now try to write a predicate **add_to_back/3** to take an element and add it to the end of a list. This does not work.

```
add_to_back(El,List,Ans):-
                Ans=[List|El].

?- add_to_back(a,[b,c,d],X).
X=[[b,c,d]|a]
```

Not only is this not even a proper list (it does not end in []) but it is not equal to **[b,c,d,a]**! What we have to do is something like:

```
add_to_back(El,[],[El]).
add_to_back(El,[Head|Tail],[Head|NewTail];-
                add_to_back(El,Tail,NewTail).
```

This is an expensive procedure. We have to do many computations before getting to the back of the list. We can, however, use difference lists to do this:

```
?- difference_append([b,c,d|Hole1]-Hole1,[a|Hole2]-Hole2,Ans-[]).
Ans=[b,c,d,a]
```

This is a cheap computation. Now we could define a version of **add_to_back/3** for difference lists:

```
add_to_back(El,OpenList-Hole,Ans):-
                difference_append(OpenList-Hole,[El|ElHole]-ElHole,Ans-[]).

?- add_to_back(a,[b,c,d|Hole]-Hole,Ans).
Ans=[b,c,d,a]
```

Exercise 12.1 *This is a set of exercises on difference lists. The first two exercises should be a rehearsal of examples from the previous notes.*

1. *Just for practice, define* **diff_append/3** *which takes two difference lists and returns a third difference list which is the second appended to the third. That is, you should get:*

   ?- diff_append([a,b|X]-X,[c,d,e|Y]-Y,Answer).
   Answer = [a,b,c,d,e|Y] - Y

2. *Again, just for practice, define* **add_at_end/3** *which adds the first argument to the end of a difference list (the second argument) and returns the result (the third argument). That is, you should get:*

   ?- add_at_end(e,[a,b,c,d|X]-X,Answer).
   Answer = [a,b,c,d,e|Y] - Y

3. *Now define a predicate* **diff_reverse/2** *which reverses the first list (1st argument) to produce the second argument. That is:*

   ?- diff_reverse([a,b,c|X]-X,Answer).
   Answer = [c,b,a|Y] - Y

   The idea is that a difference list is both input and output.

4. *Now write* **diff_flatten/2** *to flatten a proper list which consists of integers or constants or lists of these.*

   ?- diff_flatten([1,2,[3,4,[5,4,[3],2],1],7],Ans).
   Ans=[1,2,3,4,5,4,3,2,1,7|Z]-Z

5. *Now write* **diff_quicksort/2**. *This should take a difference list as its first argument basically consisting of integers and return a difference list as its second argument with the integers in order smallest to largest.*

   ?- diff_quicksort([3,1,2|X]-X,Ans).
   Ans=[1,2,3|Y]-Y

Here is a version of quicksort in **Prolog**.

```
quicksort([],[]).
quicksort([Head|Tail],Sorted):-
        split(Head,Tail,Small,Big),
        quicksort(Small,SortedSmall),
        quicksort(Big,SortedBig),
        append(SortedSmall,[Head|SortedBig],Sorted).

split(_,[],[],[]).
split(X,[Y|Tail],[Y|Small],Big):-
        X > Y,
        split(X,Tail,Small,Big).
split(X,[Y|Tail],Small,[Y|Big]):-
        X =< Y,
        split(X,Tail,Small,Big).
```

6. *Now try to use a difference list to simulate a queue. The queue is represented by a difference list. Arrivals are stuck on the back of the list using* **add_at_end/3** *and departures are removed from the front of the list in the obvious way. You will have to write other predicates to control the number of arrivals and the number of departures in some suitable way. You might use the previously defined random number generator.*

   Try to think about what happens if you try to remove the first element of an empty queue!

12.3 Prolog Layout

We now make some comments on some aspects of good programming practice with particular reference to program layout.

12.3.1 Comments

All programs should be carefully commented. This is for the standard reasons of making program maintenance easier. As **Prolog** has such a regular underlying syntax, superficially similar programs can behave very differently. Consequently, program *comments* can be very helpful in aiding *program comprehension.*

Program Headers It is sensible that a large program is divided up into self contained chunks —or, at least, chunks with explicit references made to the other chunks necessary for the program to run.

The main program is then built out of the various chunks. In SICStus **Prolog**, the programmer has to use files to represent 'program chunks'. Indeed, it is quite common for a large program to be described by a single file which loads all the necessary files in the right order.

Therefore, it is sensible to provide headers for each file as in:

```
% Program:    pract2.pl
% Author:     aidai
% Updated:    27 October 1988
```

```
% Purpose:     2nd AI2 Practical
% Uses:        append/3 from utils.pl

% Defines:
%              foo(+,-)
%              baz(+,+)
```

This header has several advantages which need no elaboration.

Section Comments There is a special form of comment which should be used with great care. Here is an illustration:

```
\*
we now define append/3 so that
it can be used as a generator
*\
```

Everything between the "/* ... */" will be ignored by **Prolog**. It is best to put this just before the code discussed.

The danger is that the programmer might forget to close off an opened comment which normally has disastrous consequences. On the positive side, it can be used to comment out chunks of code during program development.

End of Line Comments The use of the % sign to indicate a comment is generally safer because the comment is terminated automatically by an *end of line*. Consequently, this form of comment is preferred.

There are two forms of usage: as a descriptor for a predicate and as comments on individual clauses and subgoals in a clause. As an illustration of comments on predicate definitions:

Now for the use of % for clauses and subgoals:

```
append([],L,L).                    % the base case
append([H|L1],L2,[H|L3]) :-
                append(L1,L2,L3).  % recurse on the first argument
```

Everything on the line after "%" will be ignored by **Prolog**.

Code Layout Generally, separate different predicate definitions by at least one blank line. The general structure is:

```
File Header
(space)
Predicate Header
Head of Clause1 :-
                Indented Subgoal1
```

```
                    ...
                    Indented Subgoal_n
Head of Clause_2 :-
...
Head of Clause_n :-
                    Indented Subgoal_1
                    ...
                    Indented Subgoal_n
(space)
Predicate Header
...
```

12.4 Prolog Style

Now for some very short comments on improving your style. For more detail, read [Bratko, 1986, pp184–186].

12.4.1 Side Effect Programming

Avoid (where possible). Most of the time it is possible to avoid the worst offences. If forced to use *side-effect*ing predicates then try to limit their distribution throughout the code. It is a good idea to have *one* user-defined predicate within which a clause is **assert**ed, one in which a clause is **retract**ed, one to write out a term on the screen etc.

Modifying the Program at Runtime

Prolog permits this but it is bad programming style —unless you are intending to write programs to modify themselves.

It is usually better to consider carrying around the wanted information as an extra argument in all the relevant clauses.

The cut (!/0)

Where possible (and reasonable), use **\+/1** instead.

Use cuts with *great care*.

Think about every cut (**!/0**) you want to place in terms of the effect you are trying to achieve.

Always try to put them "as low as possible" in the structure of the program.

;/2

The predicate **;/2** is defined as an infix operator. It is used to express disjunctive subgoals. For example, **member/2** can be rewritten as:

```
member(X,[H|T]):-
                    (
                       X=H
```

```
        ;
            member(X,T)
        ).
```

The semantics of **;/2** are roughly equivalent to logical *or*. Best to avoid its use.

The predicate definition

```
a:- b ; c.
```

is better written as:

```
a:- b.
a:- c.
```

If you do use this construct then avoid nesting it deeply as this makes code *very* hard to read and understand.

if . . . then & if . . . then . . . else

Prolog can be made to obey control structures of this form.

The **if . . . then** form makes use of the infix operator **->/2**.

The extension to **if . . . then . . . else** is achieved with the help of the **;/2** predicate.

You may be comfortable with such constructs but it is usually better, if more cumbersome, to avoid them. Here is how one might define **Prolog**'s "if ...then ...else".

```
(A -> B ; C) :-
                call(A),
                !,
                call(B).
(A -> B ; C) :-
                call(C).
```

There are great dangers in using this construction in conjunction with the cut (**!/0**)

Just to illustrate its application, we can rewrite the predicate **analyse/1** used earlier.

```
analyse(Term):-
                type(Term,Type),
                ( (Type=compound_term ; Type = list) ->
                        (write(Term,Type),
                        functor(Term,N,A),
                        analyse_bit(0,A,Term))
                ;
                        write(Term,Type)).
```

To repeat, it can be *very difficult* to understand programs using nested **;/2** or the **if ... then (... else)** construct.

It is almost always preferable to use auxillary predicates to tidy up the 'mess'.

```
analyse(Term):-
                type(Term,Type),
                ( non_simple(Type) ->
                        analyse_non_simple(Term,Type)
                ;
                        write(Term,Type)).

non_simple(compound_term):-
                !.
non_simple(list).

analyse_non_simple(Term,Type):-
                write(Term,Type),
                functor(Term,N,A),
                analyse_bit(0,A,Term).
```

12.5 Prolog and Logic Programming

12.5.1 Prolog and Resolution

There are many different **Prolog**s but they are all based on a technique from theorem proving known as SLD Resolution.

SLD resolution can be guaranteed to be *complete* in that if a solution exists then it can be found using some search strategy.

SLD resolution can be guaranteed to be *sound* in that if an answer is obtained then it is a solution to the original problem for some search strategy.

It is a research goal to study **Prolog** implementations and check that their search strategy preserves the completeness and soundness of the underlying method of SLD resolution.

Note that the cut can affect completeness but not soundness.

Note also that there is no theoretical way of determining whether or not an attempt to solve a problem will terminate. If there is a solution then it can be shown that it can be found in a finite number of steps.

12.5.2 Prolog and Parallelism

Various people are working on strategies for parallel execution of **Prolog**.

This includes Clarke and Gregory at Imperial College, London where much work has been done in developing PARLOG.

Ehud Shapiro of the Weizmann Institute, Israel has produced Concurrent **Prolog** (CP).

12.5.3 Prolog and Execution Strategies

John Lloyd and others have produced MU-**Prolog** at the University of Melbourne in an attempt, inter alia, to replace the standard **Prolog** left-right execution strategy for subgoals with a strategy which can reorder the execution sequence depending on which subgoals have enough information to proceed with their execution. Their new implementation is called NU-**Prolog**.

12.5.4 Prolog and Functional Programming

Many attempts are being made to combine **Prolog** with functional programming features.

12.5.5 Other Logic Programming Languages

Prolog is not a pure logic programming language. It may be the best we have but there is some interest in building better languages.

As **Prolog** is less expressive than first order predicate calculus, a fair amount of work is going on to produce systems that permit the user to exploit the expressivity of full first order predicate logic —and other logics too!

12.6 What You Should Be Able To Do

You should be able to determine the type of a term (using the basic types provided by **Prolog**).
You should be able to take a **Prolog** term and transform it.
You should be able to construct arbitrary terms.
You should be able to determine whether two terms have identical bindings.
You should be able to find all the solutions of a goal.
You should understand the rudiments of good programming style.
You should know something about the notion of logic programming and the ways in which progress has been made with **Prolog** towards meeting the goal of using logic to develop programs.
You should be ready to read [Sterling & Shapiro, 1986].

Appendix A

A Short Prolog Bibliography

At the moment of writing, the most suitable books to use in conjunction with these notes are [Clocksin & Mellish, 1984] and [Bratko, 1986] (both now exist in new versions). The manual for the version of **Prolog** actually used is [SICStus, 1988] which is very similar to [Bowen, 1981].

For those with a more ambitious turn of mind then [Sterling & Shapiro, 1986] must be very highly recommended. The book by Richard O'Keefe is also highly recommended but quite hard work [O'Keefe, 1990]. Slightly less useful but worth a read is [Kluźniak & Szpakowicz, 1985].

A simpler approach can be found in [Burnham & Hall, 1985].

A number of books exist outlining the Imperial College variant of **Prolog** known as micro-Prolog. Generally, it is wiser to stay with the DEC-10 family of **Prolog**s until you are more confident. The useful books are [Ennals, 1982] for a very simple introduction and [Clark & McCabe, 1984] for a more ambitious and determined student. The best book on the market is probably [Conlon, 1985].

For reading further afield then [Kowalski, 1979] is probably the classic. Also, [Hogger, 1984] is a very worthwhile introduction to logic programming. The work of Lloyd provides those interested in theory with a very thorough analysis of the foundations of logic programming [Lloyd, 1987]. Further ideas for reading can be gleaned from [Balbin & Lecot, 1985].

Bibliography

[Balbin & Lecot, 1985] Balbin, I. and Lecot, K. (1985). *Logic Programming: A Classified Bibliography.* WildGrass Books, Australia.

[Bowen, 1981] Bowen, D.L., (ed.). (1981). *DECSystem-10 Prolog User's Manual.* Department of Artificial Intelligence, Edinburgh. Available as Occasional Paper No 27.

[Bratko, 1986] Bratko, I. (1986). *Prolog Programming for Artificial Intelligence.* Addison Wesley, Wokingham.

[Burnham & Hall, 1985] Burnham, W.D. and Hall, A.R. (1985). *Prolog Programming and Applications.* Macmillan.

[Clark & McCabe, 1984] Clark, K.L. and McCabe, F.G. (1984). *micro-Prolog: Programming in Logic.* Prentice Hall.

[Clocksin & Mellish, 1984] Clocksin, W.F. and Mellish, C.S. (1984). *Programming in Prolog.* Springer Verlag.

[Conlon, 1985] Conlon, T. (1985). *Start Problem Solving with Prolog.* Addison Wesley.

[Ennals, 1982] Ennals, J.R. (1982). *Beginning micro-Prolog.* Ellis Horwood, Chichester.

[Hogger, 1984] Hogger, C. (1984). *Introduction to Logic Programming.* Academic Press.

[Kluźniak & Szpakowicz, 1985] Kluźniak, F. and Szpakowicz, S. (1985). *Prolog for Programmers.* Academic Press.

[Kowalski, 1979] Kowalski, R. (1979). *Logic for Problem Solving. Artificial Intelligence Series*, North Holland.

[Lloyd, 1987] Lloyd, J.W. (1987). *Foundations of Logic Programming.* Springer-Verlag, 2 edition.

[O'Keefe, 1990] O'Keefe, R.A. (1990). *The Craft of Prolog.* MIT Press.

[SICStus, 1988] SICStus. (1988). *SICStus Prolog User's Guide and Reference Manual.* Swedish Institute of Computer Science, Sweden.

[Sterling & Shapiro, 1986] Sterling, L. and Shapiro, E.Y. (1986). *The Art of Prolog.* MIT Press, Cambridge, MA.

Appendix B

Details of the SICStus Prolog Tracer

The description of the SICStus Prolog tracer follows closely the description of the Quintus Prolog tracer since these two Prolog systems are very similar. We consider version 0.6 here as defined in [SICStus, 1988].

The SICStus debugger is a development of the DEC-10 debugger. It is described in terms of the so-called four port model of **Prolog** execution. The four ports are **call**, **exit**, **redo** and **fail**. Full tracing only applies to non-compiled code but some limited tracing can be done for compiled code. The behaviour is similar to the treatment of system predicates.

Monitor Execution: Different kinds of control are provided. The difference between **debug** and **trace** is that **trace** goes into **creep** mode directly whereas **debug** waits for some decision from the user to start offering the standard range of debugging options. Both otherwise cause the system to save relevant information.

The predicate **nodebug/0** switches off debugging and the predicate **debugging/0** shows the action on finding an unknown predicate, whether debugging is in action or not, which predicates are spied and what mode of leashing is in force.

Control of Information Shown:

Controlling Amount of Execution Information:
Spypoints can be set for any number of relations via the predicate **spy/1**. The argument of **spy** might be a predicate name or a name/arity or a list of such. Undefined predicates can be spied by using the name/arity form of argument.

Controlling Amount of Interaction: The **leash/1** predicate is provided to control the amount of interaction with the programmer. The options are: full (prompt on **call**, **exit redo** and **fail**), tight (prompt on **call**, **redo** and **fail**), half (prompt on **call** and **redo**), and loose (prompt on **call**)[1].

Controlling Amount of Term Visible: Representing a complex term by ellipsis is done automatically in the debugger but the user can control the complexity of displayed terms. There do not

[1] The off (no prompt) choice provided by Quintus does not seem to be supported.

appear to be any supplied procedures to manage the depth when using **write/1** etc. There is also a way of examining subterms via the **set subterm** option within the debugger.

Going Forwards: There are several different versions of the 'next interesting event'.

A Step at a Time: The user is able to single step through the code using the **creep** option.

On to Next Spy Point: The user is able to jump to the next predicate that is spied using the **leap** option.

Skip: The **skip** option jumps to the **exit/fail** port of the procedure or the first procedure encountered that has a spy-point. Only available at the **call/redo** ports. It does not stop at spy points.

Going Backwards: Single stepping backwards versus jumping back to the last choice point.

Retry a Previous Goal: The **retry** command transfers control back to the **call** port of the current box. Everything is as it was unless any **assert/retract**s have taken place. It is possible to give an argument to **retry** the computation from much further back —this gets messed up usually by cut (!). Side effects are, inevitably, not undone. This includes clauses asserted etc.

Interfering with Execution: Different ways of handling this.

Forcing Failure: While tracing, the programmer is able to force a failure even though a goal would succeed. This can be done via the **unify** option. Just try to unify the current goal with some impossible term.

Forcing Success: This feature is provided via the **unify** choise at the **Call** port for a goal. This could be badly abused.

Find Another Solution: This does not seem to be possible.

Examining a Goal: Different ways of looking at **Prolog** terms.

Writing a Goal: Printing the goal with the syntax obtained by applying any operator declarations in force.

Pretty Printing a Goal: Printing the goal possibly using the user defined **portray/1**, if possible.

Displaying Underlying Syntax: Showing the regular syntax of some goal using **display/1**

Showing Context: Details of the execution in terms of what has happened, what has yet to be done and the source code.

Ancestors: Looking at some possibly user-defined number of ancestor goals. Equivalent to the **ancestors/0** command.

Breaking Out: Providing the facility to access **Prolog** while tracing —with sometimes irreversible consequences.

Single Command: A single command can be executed.

New Interpreter: A new incarnation of **Prolog** is initiated via a call to **break** which will be in force until either another **break** command is given, or an **abort** or an **end_of_file** character.

Aborting Execution Calls the command **abort/0** which aborts the execution back to top level, throwing all incarnations of the interpreter away.

Appendix C

Solutions and Comments on Exercises for Chapter 2

C.1 Exercise 2.1

1. **likes(bill,ice_cream).**

   The predicate **likes** has the declarative reading that the first named object (first argument) 'likes' the second named object.

   We could, of course, have defined **likes** in the reverse way. This would lead to the representation **likes(ice_cream,bill)** and the reading, in this case, that 'ice_cream' is 'liked' by 'bill'.

   Note that we have renamed *ice-cream* systematically to **ice_cream**.

   The term **ice-cream** is a syntactically correct **Prolog** term but it is not an atom since the rules for atoms do not allow for the use of the - character.

   This could be got round in other ways than the above —*e.g.* **'ice-cream'**.

   Also note that we could get away with a one argument 'relation' — *viz* **likes_ice_cream(bill)**. Or even a zero argument 'relation' — **bill_likes_ice_cream**.

   We could try the representation that **bill(likes,ice_cream)**. Usually, predicates are associated with verbs.

2. **is(bill,tall).**

   This might be chosen but there is are problems: first, with the word 'is'. Here, it is associated with the idea that bill possesses an attribute which has the value 'tall'.

   We could represent this as **height(bill,tall)**.

   Another reason for not using **is(bill,tall)** is that there may be many such statements in a database. **Prolog** would then have to sort through a large number of **is/2** clauses such as **is(bill,rich)**. If we choose **height(bill,tall)** then we only search through the clauses that deal with height.

   Finally, the predicate **is/2** is a system predicate and cannot be redefined by the user!

By the way, after all this, note that **tall(bill)** is quite acceptable and probably the one most people will prefer. However, we should note that this representation will make it harder to pick up any relation between **tall(bill)** and, for example, **short(fred)** whereas this is easy for **heigh(bill,tall)** and **height(fred,short)**.

3. **hits(jane,jimmy,cricket_bat)**.

   The declarative reading for **hits/3** is that the 'hit' action is carried out by the first named object on the second named object with the aid of the third object.

   Note that nowhere in this reading is there any sense in which there is insistence on the first (or second) object being the identifier for a person, nor that the third object should be a 'blunt instrument' etc. We could be much more rigid about the meaning.

   As before, there are many variants as to how this could be done. The main point is to stick to one reading throughout the development of a program.

   And again, we have mapped 'cricket-bat' to 'cricket_bat'.

4. **travels(john,london,train)**.

   The declarative reading is that the 'travels' relation holds for the first object travelling to the second object with the means of transport described by the third object.

5. **takes(bill,jane,cheese,edam)**.

   This is a fairly unattractive way to do things —but is easiest for now.

   The reading is that the 'takes' relation holds for the first object transporting a specimen of the third object (which is or sort described by the fourth object) to the second object.

   (Later we will see that we can tidy this up by writing **takes(bill,jane,cheese(edam))** where **cheese(edam)** is a legitimate **Prolog** term.)

6. **lives_at(freddy,16,throgmorton,street,london)**.

   Again, the reading is that the 'lives_at' relation holds for the first object 'living' at an address described by the second, third, fourth and fifth objects.

   This is ugly but is needed if we want to access bits of the address. If we don't want to access bits of the address then we can get away with **lives_at(freddy,'16 throgmorton street, london')**. Now we have a simpler relation and the second argument stands for the whole address.

   The first representation of **lives_at/5** has its own problems. For example, what happens if someone doesn't require a descriptor such as street or road? This has not been specified.

C.2 Execise 2.2

1. **eats(bill,chocolate).**
   eats(bill,bananas).
   eats(bill,cheese).

Here, **eats/2** is read as being a relation that holds when the first named object is willing to consume samples of the second named object.

2. **square_root(16,-4).**
   square_root(16,4).

3. **country(wales).**
   country(ireland).
   country(scotland).

   The reading is that the one-place 'relation' holds when the named object has the status of a country. Here, in our informal description of the semantics of the predicate, we have said nothing about the meaning of what it is to be a country.

C.3 Exercise 2.3

1. **eat(X,custard):- animal(X).**

   This can be paraphrased as 'if X is an animal then X eats custard'.

   We ought also to provide the informal semantics for **eat/2** and **animal/1**. Let us assume that this can be done.

   By the way, we could also, but less satisfactorily, write **custard_eater(X):- animal(X)**.

2. **loves(X,Y):- directed_by(bergman,Y).**

   The relation 'loves' holds between any two objects if the second object is related to bergman via the **directed_by** relation.

   Note that nowhere have we said that the first argument of the **loves/2** relation should be a person. This is implicit in the original statement and, strictly, ought to be enforced.

3. **likes(jim,X):- belongs_to(X,fred)**.

   The relation 'likes' holds between 'jim' and some other object if this object is related to 'fred' through the 'belongs_to' relation.

   Again note that the declarative readings for both **likes/2** and **belongs_to/2** are not provided by this statement.

4. **may_borrow(X,bike,jane):- need(X,bike).**

   The relation 'may_borrow' holds between the first argument and the second (where the third is the owner of the second) if these two arguments are related via the 'need' relation.

C.4 Exercise 2.4

1. **liable_for_fine(X):- owns_car(X,Y), untaxed(Y).**

   We assume that **liable_for_fine/1** holds when its argument is (a person) liable for a fine, that **owns_car/2** holds when the first argument possesses the object named in the second argument (and this object is a car), and that **untaxed/1** holds for all those objects that are required by law to be taxed and are not!

2. **same_house(X,Y):- address(X,Z), address(Y,Z).**

   The **same_house/2** relation holds between two arguments (people) if the **address/2** relation holds between one of these arguments and a third object and between the other and the same third object.

   Note that this makes **same_house(fred,fred)** true.

3. **siblings(X,Y):- mother(X,M), mother(Y,M), father(X,P), father(Y,P).**

   The **siblings/2** relation holds between the two arguments when each is related via the **mother/2** relation to a common object and via the **father/2** relation to a (different) common object.

   This is not correct if the intended meaning is to prevent one person being their own sibling. We would revise this by adding a subgoal such as **not_same(X,Y)**.

   Note that we could have designed a **parents/3** predicate (relation) such that, for example, the second argument is the mother and the third is the father of the first argument. This would result in **siblings(X,Y):- parents(X,M,P), parents(Y,M,P)**.

C.5 Exercise 2.5

1. **british(X):- welsh(X).**
   british(X):- english(X).
   british(X):- scottish(X).
   british(X):- northern_irish(X).

   Note that we have preserved the order of nationalities as described in the statement. This has no logical significance.

2. **eligible_social_security(X):- earnings(X,Y), less_than(Y,28).**
   eligible_social_security(X):- oap(X).

   In the first part of the disjunction, we have introduced an additional predicate **less_than/2** which has the reading that the relation holds when the first argument is less than the second.

   Also, note that the original statement does not make it clear whether or not someone could qualify both as an old age pensioner (oap) and as someone earning very little. This could become an important issue.

3. **sportsperson(X):- plays(X,football).**
   sportsperson(X):- plays(X,rugger).
   sportsperson(X):- plays(X,hockey).

C.6 Exercise 2.6

1. **b:- a.**

   Note that b is true if a is true.

2. **c:- a.**
   c:- b.

   Here we have a straight use of disjunction.

3. **c:- a, b.**

   Here is a straightforward example of a conjunction.

4. **d:- a, b.**
   d:- a, c.

   This is a hard one. We cannot (yet) write what we want to write: that is, **d:- a, (b or c).** Here, we can use de Morgan's law: this is the equivalence: $\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) \Rightarrow \mathbf{d} \Leftrightarrow (\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c}) \Rightarrow \mathbf{d}$.

5. **b:- a**.

   This is hard too. The obvious solution is:

   not a.
   b.

   but this is not allowed. Consequently, we have to transform the expression using the logical equivalence $\neg\ \mathbf{a} \vee \mathbf{b} \Leftrightarrow \mathbf{b} \Rightarrow \mathbf{a}$.

C.7 Exercise 2.7

1. **studies(bill,ai2).**

   We have revised 'AI2' to 'ai2'. We could have simply put quotes around as in **studies(bill,'AI2').**

2. **population(france,50)**.

   where the reading is that the population of the first object in the relation **population/2** is the second object expressed in millions of people.

   Note we have changed 'France' to 'france'.

3. **rich_country(italy).**

   Here, the statement has been expressed as a unary 'relation' of something being a rich country.

4. **height(jane,tall).**

   We have covered a similar example previously.

5. **prime(2).**

   We have asserted that the attribute of primeness belongs to the number 2.

6. **british(X):- welsh(X).**

   The statement has been turned into the equivalent 'everybody who is welsh is british'. This is an alternative to the statement **subset(welsh,british)**. We read this as meaning that the **subset/2** relation holds between the set of welsh people and the set of british people.

   As usual, we have lower-cased the words 'Welsh' and 'British'.

7. **author(hamlet,someone).**

   This is a trick question. You cannot answer this one from the notes. Why not? Well, let me give the meaning of the above: the **author/2** relation holds between 'hamlet' (which stands for the famous play called "Hamlet: Prince of Denmark") and the unique atom **'someone'** which has been conjured from thin air.

   The problem lies in expressing existential statements such as "someone likes ice-cream" and so on. This is informally recast as there exists some person such that this person likes ice-cream. In first order predicate logic, we would formalise this as $\exists$ **x likes(x,ice_cream)**. This can be turned into **likes(whatshisname,ice_cream)** (this is known as *Skolemisation).* Without going into technicalities, we give a legitimate context when this 'trick' can be done —whenever we have no universal quantifiers (*i.e.* indicated by words such as all, everyone, etc) then we may introduce a unique atom (we should be able to guarantee its uniqueness) to stand for the 'someone'.

8. **mortal(X):- human(X).**

   This is an example of a universally quantified statement. It is equivalent to $\forall$ **x human(x)** $\Rightarrow$ **mortal(x)**.

   Note that, in the **Prolog** version, this 'universal quantification' is implicit.

9. **pays_taxes(X):- person(X), rich(X).**

   Again, the universal quantification is implicit in the **Prolog** version.

   Here, we have a body with a conjunction of two goals. This could be avoided with **pays_taxes(X):- rich_person(X).** Which you prefer depends on the way other relevant information is to be used or, how it is provided.

10. **takes(bill,umbrella):- raining.**

    This is a version where it is true that 'Bill' takes his umbrella whenever it is raining.

    Note that in many of these examples, there is no mention of how the truth of various statements change with time.

11. **no_supper(X):- naughty(X).**

    Here, we might have tried to write $\neg$ **supper(X):- naughty(X)**. This is, however, illegal in **Prolog** but not for syntactic reasons.

    Another way of doing this might be **eats_supper(X,false):- naughty(X)**. This allows for a more uniform treatment of both those who are 'naughty' and those who aren't.

12. **employs(firebrigade,X):- man(X), height(X,Y), more_than(Y,6.0).**

    Again, we have gone for the representation 'most likely' to be useful.

    We could hide much of this as **firebrigade_employs(X):- over_six_foot(X)**.

Appendix D

Solutions and Comments on Exercises for Chapter 3

D.1 Exercise 3.1

In this set of exercises, the solutions are slightly abbreviated. Even so, it is likely that your solution is a subset of the solutions proposed here. The most difficult issue is what happens on failure.

1.

Subgoals	Comment	Result
a(1)	uses 1st clause	new subgoals
b(1,Y)	uses 1st clause	succeeds with Y=2
c(2)	uses 1st clause	succeeds
a(1)	using 1st clause	succeeds

The goal is solved in a very straightforward way. There is no backtracking.

2.

Subgoals	Comment	Result
a(2)	uses 1st clause	new subgoals
b(2,Y)	uses 1st clause	fails
b(2,Y)	uses 2nd clause	succeeds with Y=2
c(2)	uses 1st clause	succeeds
a(2)	using 1st clause	succeeds

Here, we have a simple case where a failure forces the use of the second half of the definition for **b/2**.

3.

Subgoals	Comment	Result
a(3)	uses 1st clause	new subgoals
b(3,Y)	tries 1st clause	fails
b(3,Y)	tries 2nd clause	fails
b(3,Y)	tries 3rd clause	succeeds with Y=3
c(3)	tries 1st clause	fails
c(3)	tries 2nd clause	fails
b(3,Y)	tries 4th clause	succeeds with Y=4
c(4)	tries 1st clause	fails
c(4)	tries 2nd clause	fails
b(3,Y)	no more clauses	fails
a(3)	uses 2nd clause	new subgoal
c(3)	tries 1st clause	fails
c(3)	tries 2nd clause	fails
a(3)	no more clauses	fails

This is much harder because in trying to solve **b(3,Y),c(Y)** we have the first subgoal succeed then the second fail twice over before running out of options. We then backtrack to try the remaining option for solving the top level goal which is the subgoal **c(3)** but this also fails.

4.

Subgoals	Comment	Result
a(4)	uses 1st clause	new subgoals
b(4,Y)	tries 1st clause	fails
b(4,Y)	tries 2nd clause	fails
b(4,Y)	tries 3rd clause	fails
b(4,Y)	tries 4th clause	fails
a(4)	uses 2nd clause	new subgoal
c(4)	tries 1st clause	fails
c(4)	tries 2nd clause	fails
a(4)	no more clauses	fails

A little simpler because no subgoal succeeds at all.

D.2 Exercise 3.2

1.

Subgoals	Comment	Result
a(1,X)	uses 1st clause	new subgoals
b(1,X)	tries 1st clause	succeeds with X=2
a(1,X)	using 1st clause	succeeds with X=2

Another straightforward solution.

2.

Subgoals	Comment	Result
a(2,X)	uses 1st clause	new subgoals
b(2,X)	tries 1st clause	fails
b(2,X)	tries 2nd clause	succeeds with X=3
a(2,X)	using 1st clause	succeeds with X=3

Again, not too difficult.

3.

Subgoals	**Comment**	**Result**
a(3,X)	uses 1st clause	new subgoal
b(3,X)	tries 1st clause	fails
b(3,X)	tries 2nd clause	fails
a(3,X)	uses 2st clause	new subgoals
c(3,X_1)	tries 1st clause	fails
c(3,X_1)	tries 2nd clause	fails
c(3,X_1)	tries 3rd clause	fails
c(3,X_1)	tries 4th clause	succeeds with X_1=4
a(4,X)	uses 1st clause	new subgoal
b(4,X)	tries 1st clause	fails
b(4,X)	tries 2nd clause	fails
a(4,X)	uses 2nd clause	new subgoals
c(4,X_2)	tries 1st clause	fails
c(4,X_2)	tries 2nd clause	fails
c(4,X_2)	tries 3rd clause	fails
c(4,X_2)	tries 4th clause	fails
a(4,X_2)	no more clauses	fails
a(3,X)	no more clauses	fails

This is a challenging one. First, because we get involved with the unpleasant second clause for **a/2**. In general, when using the second clause for **a/2**, the goal **a(X,Y)** requires that we set up two new subgoals **c(X,Z),a(Z,Y)**. This introduces a new variable. Textually, it is the 'Z' but every time we introduce a variable we have to use a different name. Here, we have provided a suffixed 'X' each time we introduce the new variable.

4.

Subgoals	**Comment**	**Result**
a(X,4)	uses 1st clause	new subgoal
b(X,4)	tries 1st clause	fails
b(X,4)	tries 2nd clause	fails
a(X,4)	uses 2st clause	new subgoals
c(X,X_1)	tries 1st clause	succeeds with X=1, X_1=2
a(2,4)	uses 1st clause	new subgoal
b(2,4)	tries 1st clause	fails
b(2,4)	tries 2nd clause	fails
a(2,4)	uses 2nd clause	new subgoals
c(2,X_2)	tries 1st clause	fails
c(2,X_2)	tries 2nd clause	succeeds with X_2=4
a(4,4)	uses 1st clause	new subgoal
b(4,4)	tries 1st clause	fails
b(4,4)	tries 2nd clause	fails
a(4,4)	uses 2nd clause	new subgoals
c(4,X_3)	tries 1st clause	fails
c(4,X_3)	tries 2nd clause	fails
c(4,X_3)	tries 3rd clause	fails
c(4,X_3)	tries 4th clause	fails
a(4,4)	no more clauses	fails
a(2,4)	no more clauses	fails
c(X,X_1)	tries 2nd clause	succeeds with X=1, X_1=4
a(4,4)	uses 1st clause	new subgoal
b(4,4)	tries 1st clause	fails
b(4,4)	tries 2nd clause	fails
a(4,4)	uses 2nd clause	new subgoals
c(4,X_3)	tries 1st clause	fails
c(4,X_3)	tries 2nd clause	fails
c(4,X_3)	tries 3rd clause	fails
c(4,X_3)	tries 4th clause	fails
a(4,4)	no more clauses	fails
c(X,X_1)	tries 3rd clause	succeeds with X=2, X_1=4
a(4,4)	uses 1st clause	new subgoal
b(4,4)	tries 1st clause	fails
b(4,4)	tries 2nd clause	fails
a(4,4)	uses 2nd clause	new subgoals
c(4,X_3)	tries 1st clause	fails
c(4,X_3)	tries 2nd clause	fails
c(4,X_3)	tries 3rd clause	fails
c(4,X_3)	tries 4th clause	fails
a(4,4)	no more clauses	fails
c(X,X_1)	tries 4th clause	succeeds with X=3, X_1=4
a(4,4)	uses 1st clause	new subgoal
b(4,4)	tries 1st clause	fails
b(4,4)	tries 2nd clause	fails
a(4,4)	uses 2nd clause	new subgoals
c(4,X_3)	tries 1st clause	fails
c(4,X_3)	tries 2nd clause	fails
c(4,X_3)	tries 3rd clause	fails
c(4,X_3)	tries 4th clause	fails
a(4,4)	no more clauses	fails
c(X,X_1)	no more clauses	fails
a(X,4)	no more clauses	fails

This is even worse —mainly because the first time we use the second clause for **a/2** we get involved in a subgoal **c(X,X_1**. This can be solved in four different ways —but in each case the next subgoal (**a(X_1,4)**) fails. Hence **c(X,X_1)** fails and therefore, because there are no more clauses for

a/2, **a(X,4)** fails as well.

Note that each time we attempted a new subgoal **c/2** we said we would creat a new variable: we do not need a new name for a variable if we are trying to resatisfy a goal. Look at the references to **c(...,$\mathbf{X}_3$)**: there are three different places in the above where we try to solve such a goal and fail.

5.

Subgoals	**Comment**	**Result**
a(1,3)	uses 1st clause	new subgoal
b(1,3)	tries 1st clause	fails
b(1,3)	tries 2nd clause	fails
a(1,3)	uses 2st clause	new subgoals
c(1,X_1)	tries 1st clause	succeeds with X_1=2
a(2,3)	uses 1st clause	new subgoal
b(2,3)	tries 1st clause	fails
b(2,3)	tries 2nd clause	succeeds
a(2,3)	using 1st clause	new subgoal
a(1,3)	using 2nd clause	succeeds

So, with this example, we end with a simpler case.

Appendix E

Solutions and Comments on Exercises for Chapter 4

E.1 Exercise 4.1

1. **2+1=3** fails.

   We can tell immediately that 3 is an atom but what about **2+1**? This does not look like an atom —indeed it is not. The only way that **2+1=3** is if **Prolog** were to automatically try to evaluate any 'sum' it finds before trying to do the unification. **Prolog** does not do this.

2. **f(X,a)=f(a,X)** succeeds with X=a.

   Here, the predicates are the same (so far so good). Now we match the first arguments: they can be made the same if X=a (so far so good). Now we look at the second argument: does **a** match with **X**? yes.

3. **fred=fred** succeeds.

4. **likes(jane,X)=likes(X,jim)** fails.

   Here, the predicates are the same (so far so good). Now we match the first arguments: they can be made the same if X=jane (so far so good). Now we look at the second argument: does **X** match with **jim**? Well, **X** is bound to **jane** and **jane** does not match with **jim**. So the unification fails.

5. **f(X,Y)=f(P,P)** succeeds with X=Y=P.

   Here, the predicates are the same. Now we match the first arguments: they can be made the same if X=P. Now we look at the second argument: does **Y** match with **P**? yes, and since X=P we get our final result.

E.2 Exercise 4.2

1. **[a,b|X]=[A,B,c]** succeeds with A=a, B=b and X=[c].

   First, the left and right hand terms are both lists. Now to match their 1st elements: **A** matches with **a**. The second elements **B** matches with **b**. What happens now? Let us discard the first two elements of each list. We are left with matching **X=[c]** —this succeeds.

Quite a few think the result should be X=c. Remember that the syntax **[H|T]** means that the term following the | symbol is a list —so, the **X** in the problem is a list. Therefore we cannot have X=c.

2. **[a,b]=[b,a]** fails.

   Look at the first elements: does **a** match with **b**? No —so the unification fails. Some may see lists as 'bags' of things where the order of occurrence is immaterial. This is not so.

3. **[a|[b,c]]=[a,b,c]** succeeds.

   The first elements of the two lists are identical. Throw them away and we are left with **[b,c]=[b,c]** which succeeds. The main point to note is again that the term following the | symbol is a list and that it is specifically **[b,c]**.

4. **[a,[b,c]]=[a,b,c]** fails.

   We can tell quickly that the unification must fail because the first list has two elements and the second has three. Therefore they cannot unify.

   If we discard the (equal) heads we have **[[b,c]]=[b,c]**. The left hand side is a list consisting of a single element (which just happens to be a list itself). The right hand side is a list of two elements. Going on, what is the first element of each of these two lists? On the left we have **[b,c]** and on the right we have **b**. These terms are not unifiable.

5. **[a,X]=[X,b]** fails.

   The first element of each list (the heads) can be unified —with X=a. Looking at the second elements, we need to unify **X** with **b** —but X=a so the process fails.

6. **[a|[]]=[X]** succeeds with X=a.

   The list **[a|[]]** is exactly equivalent to [a]. Therefore the problem becomes **[a]=[X]**. This unifies with X=a.

7. **[a,b,X,c]=[A,B,Y]** fails.

   The simple answer is that the left hand list has four elements and the right has three —therefore these two lists will not unify.

   To see why, we match the head elements —we get A=a. Throwing away the heads, we end up with **[b,X,c]=[B,y]**. Repeating, we have B=b. Again, discarding the heads, we have **[X,c]=[y]**. Repeating we get X=y. We now end up with **[c]=[]**. Fails.

8. **[H|T]=[[a,b],[c,d]]** succeeds with H=[a,b] and T=[[c,d]].

   The right hand list has two elements: the first (head) is **[a,b]** and the second element is **[c,d]**. The head elements unify with H=[a,b]. If we now discard the head elements we are left with deciding whether **T** unifies with **[[c,d]]**. Succeeds with **T=[[c,d]]**.

9. **[[X],Y]=[a,b]** fails.

   If we try to unify the head elements of these lists we have the problem **[X]=a**. This fails.

E.3 Exercise 4.3

1.

```
print_every_second([]).
print_every_second([X]).
print_every_second([X,Y|T]):-
        write(Y),
        print_every_second(T).
```

The trick is to realise that the notation for the general list actually allows a fixed number of elements to be ripped off/stuck on the front of a list. Here, we destruct the list by specifying that we take the first two elements off the front of any list with more than one element and then print the second of these two elements.

Note that this does not do anything clever with nested lists: *i.e.* **print_every_second([a,[b],c,[d]])** will print **[b][d]**!

2.

```
deconsonant([]).
deconsonant([A|B]):-
        vowel(A),
        write(A),
        deconsonant(B).
deconsonant([A|B]):-
        deconsonant(B).

vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).
```

Note that we need three clauses to cover the three basic cases: either the list has no elements or we want to print the first element (because it is a vowel) or we don't want to print the first element.

Provided the list is not empty then, for either of the remaining cases, we want to take off the first element and process the remaining list —we have described this procedurally as there is no good declarative reading.

> Observant readers will note that the logic of the case analysis is none too good. The third clause should really be something like
>
> ```
> deconsonant([A|B]:-
>         consonant(A),
>         deconsonant(B).
> ```
>
> But it would be very tedious to write out all the clauses for **consonant/1** —*e.g.* consonant(b) etc. There is another way of doing this, however, which we meet in chapter 7.

3.

```
head([H|T],H).
```

The reading is that the second argument is related to the first via the 'head' relation if the first element of the first argument (a list) is the second argument.

4.

```
tail([H|T],T).
```

A straightforward adaption of the previous case.

5.

```
vowels([],[]).
vowels([H|T],[H|Rest]):-
        vowel(H),
        vowels(T,Rest).
vowels([H|T],Rest):-
        vowels(T,Rest).

vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).
```

This is quite a different style of program from the very procedural **deconsonant/1**.

The same case analysis has been done but now we have to think what these clauses mean. Procedurally, we can tell quite similar story: the first case is that whenever we encounter an empty list then we will return an empty list.

The second case is that whenever we have a list with a vowel at the front then we return a list with that vowel at the front —the rest of the list has to be determined by gathering up all the vowels from the tail of the input list.

The third case is that whenever the previous two cases fail then we discard the first element and go off to pick up all the vowels in the tail.

The second clause could have been written as:

```
vowels([H|T],Ans):-
        vowel(H),
        vowels(T,Rest),
        Ans=[H|Rest].
```

if you really wanted to do so. You might find this easier to understand but the two versions are logically identical here.

The declarative reading runs something like this: for the first clause, the list of vowels in the empty list is the empty list.

The second case has the meaning that the list of vowels in another list with a vowel at the front has that vowel at the front and its tail is the list of vowels found in the other list.

The third case has the meaning that the list of vowels in another list with a consonant at the front is the list of vowels in the tail of that list.

6.

```
find_every_second([],[]).
find_every_second([X],[]).
find_every_second([X,Y|T],[Y|Rest]):-
        write(Y),
        find_every_second(T,Rest).
```

The first clause states that the list of every second element in the empty list is the empty list. The second states the corresponding thing for a list with a single element.

The third clause is the interesting one: the list of every second element of another list is the second element together with the list of every second element of the remainder of the other list.

Appendix F

Solutions and Comments on Exercises for Chapter 6

F.1 Exercise 6.1

All the programs in these examples can be done by selecting the right schema and then instantiating it correctly.

1. We now produce solutions making use of the schema *Test For Existence*.

```
list_existence_test(Info,[Head|Tail]):-
                    element_has_property(Info,Head).
list_existence_test(Info,[Head|Tail]):-
                    list_existence_test(Info,Tail).
```

   (a) We discard all the parameters from the schema (**Info**). We rename *list_existence_test* to **an_integer** and *element_has_property* to **integer**. We will not show how the others programs are written using the schema.

```
an_integer([H|T]):-
        integer(H).
an_integer([H|T]):-
        an_integer(T).
```

   If the head of the list is an integer then the list contains an integer (this describes the first clause) —otherwise we require that the tail of the list has an integer somewhere (the second clause).

   Note that the second clause does not strictly have to be second. The two clauses could be the other way round.

```
an_integer([H|T]):-
        an_integer(T).
an_integer([H|T]):-
        integer(H).
```

   If this were so, however, the program would execute much less efficiently. You could try defining the program both ways round and look at what happens using the **trace** command. What is

going on? Either way round, we have the same declarative reading but the procedural reading is not so straight forward.
For the first way, it goes roughly like this: examine the head of the list and stop with success if this is an integer —otherwise discard the head of the list and examine the remainder for whether it contains an integer. For the second way: throw away the head of the list and examine the remainder to see whether it has an integer in it —otherwise look at the head of the list to see whether it is an integer. This sounds very peculiar but it works. What happens is that **Prolog** throws away all the elements, gets to the empty list and then fails. Backtracking now leads it to try showing that the list consisting of the last element in the original list is an integer: if this is not so then further backtracking will lead **Prolog** to try the list consisting of the last two elements of the original list has an integer at the front of the list. This keeps on until either an integer is found (working back through the list) or there is no way to backtrack.

(b)
```
has_embedded_lists([H|T]):-
        H=[Embeddedhead|_Tail].
has_embedded_lists([H|T]):-
        has_embedded_lists(T).
```

A list has an element which is itself a (non-empty) list if the head of the list is a non-empty list (the first clause) or else the list's tail has an element in it that is a (non-empty) list (the second clause).
We can also rewrite this to perform implicit unification rather than the explicit unification **H=[Embeddedhead|_Tail]**:

```
has_embedded_lists([[Embeddedhead|_Tail]|T]).
has_embedded_lists([H|T]):-
        has_embedded_lists(T).
```

Does the order of these clauses matter? The same issues apply as with the previous example.

Note that if we want to fix the problem of extending the code to handle the empty list as well we need:

```
has_embedded_lists([H|T]):-
        H=[].
has_embedded_lists([H|T]):-
        H=[Embeddedhead|_Tail].
has_embedded_lists([H|T]):-
        has_embedded_lists(T).
```

That is, another clause to handle the case where the head of the list is an empty list.
By the way, this can be rewritten as:

```
has_embedded_lists([[]|T]).
has_embedded_lists([[Embeddedhead|_Tail]|T]).
has_embedded_lists([H|T]):-
        has_embedded_lists(T).
```

That is, we can rewrite the explicit unification (**H=[Embeddedhead|_Tail]** as an implicit unification.
Why does this solution become dubious for *e.g.* the query **?- has_embedded_lists([a,X,b])**? The check that the head

element is a list will eventually encounter the equivalent of **X=[Embeddedhead|_Tail]** (or **X=[]**) —which will succeed! Is this what is wanted? For then, a list containing a variable will always have an embedded list in it. This may be OK but we would want to know a little more before making a decision.

2. We now produce solutions for the schema *Test All Elements*.

```
test_all_have_property(Info,[]).
test_all_have_property(Info,[Head|Tail]):-
                element_has_property(Info,Head),
                test_all_have_property(Info,Tail).
```

(a) We discard all the parameters from the schema (**Info**). We rename *test_all_have_property* to **all_integers** and *element_has_property* to **integer**. We will not show how the others programs are written using this schema.

```
all_integers([]).
all_integers([H|T]):-
        integer(H),
        all_integers(T).
```

We now require that every member of the input list has a common property —*viz* that of being an integer.

We note that the reading of the first clause is that every element of the empty list is an integer. The second clause states that for every list, every element is an integer if the head of the list is an integer and every element of the remaining list is an integer.

(b)
```
no_consonants([]).
no_consonants([H|T]):-
        vowel(H),
        no_consonants(T).

vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).
```

Again, the empty list is such that every element in it is not a consonant. And, again, a list has no consonants if the head of the list (first element) is not a consonant and the remainder (tail) of the list has no consonants in it.

We could have done this a little differently with the help of the predicate **\+/1**.

```
no_consonants([]).
no_consonants([H|T]):-
        \+ consonant(H),
        no_consonants(T).
```

```
consonant(b).
consonant(c).
(etc.)
```

The need to specify 26 consonants is a little tedious but acceptable.

3. We now produce the solutions that make use of the schema *Return a Result —Having Processed One Element*.

   We use this schema:

```
return_after_event(Info,[H|T],Result):-
                property(Info,H),
                result(Info,H,T,Result).
return_after_event(Info,[Head|Tail],Ans):-
                return_after_event(Info,Tail,Ans).
```

   (a) This one can be shown to be an example of the schema but the 'obvious' solution doen't fit exactly.

```
nth(1,[H|T],H).
nth(N,[H|T],Ans):-
        NewN is N-1,
        nth(NewN,T,Ans).
```

   The first clause has the declarative reading that the first element of the list is its head. The second clause has the declarative reading that the nth element of the list is the n-1th element of the list's tail.

   It can be difficult to appreciate how this works. So the procedural reading for the second clause can be taken as: to find the nth element of the list, lop off the first element (the head) and then look for the n-1th element in the remainder of the list (the tail).

   Note that the order of the subgoals is important here. If an attempt is made to write this as:

```
nth(1,[H|T],H).
nth(N,[H|T],Ans):-
        nth(NewN,T,Ans),
        NewN is N-1.
```

   then we get into trouble: when the first subgoal (assuming the first argument is an integer greater than 1) is executed then the variable **NewN** will not be bound to an integer. This will mean that once we have recursed down the list until the first clause succeeds then we will have a number of subgoals awaiting execution —the first of which will be **1 is Variable -1**. This fails with an error message as the **is/2** predicate requires that the right hand side (second argument) be an arithmetical expression and it is not.

   We can generate another solution with a little bit of trickery.

```
nth2(N,[H|T],H):-
        1 is N.
nth2(N,[H|T],Ans):-
        nth2(N-1,T,Ans).
```

This, if you trace the execution, generates a series of subgoals of the form *nth2(somenumber-1-1-1-1-1...,list,variable.* The first clause succeeds when the first argument evaluates to 1. Note that the second clause is much neater as a consequence.
The observant will notice that these various versions of **nth/3** do not fit the schema *that* well. This is partly because the recursion variable is the first argument and is on the natural numbers rather than lists.

(b)

```
next(PossibleElement,[PossibleElement,NextElement|T],NextElement).
next(PossibleElement,[H|T],Ans):-
        next(PossibleElement,T,Ans).
```

This program has a straightforward (declarative) reading: the first clause states that the next element (third argument) after the named one (first argument) is when the list (second argument) begins with the named element and followed by the desired next element. Note the flexible use of the list notation which allows the user to specify a fixed number of elements at the front of a list (here, two).

(c)

This solution fits the desired schema exactly (and also makes use of the schema *Building Structure in the Clause Head*).

We use one parameter from the schema (**Info**). We rename *return_after_event* to **del_1st1**, *property* to =, *result* to =. This results in:

```
del_1st1(ToGo,[ToGo|T],Ans):-
        Ans=T.
del_1st1(ToGo,[H|T],[H|NewT]):-
        del_1st1(ToGo,T,NewT).
```

which can be rewritten to

```
del_1st1(ToGo,[ToGo|T],T).
del_1st1(ToGo,[H|T],[H|NewT]):-
        del_1st1(ToGo,T,NewT).
```

The declarative reading is that when the first argument is the head of the list (which is the second argument) then the third argument is the tail (remainder) of the list —otherwise, the third argument is a list with the first element the same as that of the second argument and the tail is the list with the desired element deleted.

We can also describe this procedurally, but we will assume that it is intended that the third argument is output and the other two are inputs.

When we find the desired element at the head of the input list (the second argument) then we return the tail of that list. When we do not find this, we copy over the head into the output list and go looking for the result of deleting the desired element from the remainder (tail) of the input list.

Now here is a 'solution' using the schema *Building Structure in the Clause Body.*

```
del_1st2(ToGo,L,Ans):-
        del_1st2(ToGo,L,[],Ans).
del_1st2(ToGo,[ToGo|T],Acc,Ans):-
        append(Acc,T,Ans).
del_1st2(ToGo,[H|T],Acc,Ans):-
        del_1st2(ToGo,T,[H|Acc],Ans).

append([],L,L).
append([H|L1],L2,[H|L3]):-
        append(L1,L2,L3).
```

Note that we have defined a predicate **del_1st2/3** which interfaces to **del_1st2/4** and initialises the accumulator (third argument) to the empty list.

The two clauses have a procedural reading: when we find the desired element then we glue the accumulator onto the front of the remainder (tail) of the list found in the second argument using **append/3** — otherwise, we copy the head of the list to the head of the accumulator and then try to delete the desired element from the tail of the list using the new accumulator.

4. For each of these examples, we use the schema *Return a Result —Having Processed All Elements*. We have, however, two ways of writing each with each way corresponding to a different (list processing) schema. These are, namely, *Building Structure in the Clause Head* and *Building Structure in the Clause Body*.

   Here is the schema making use of the *Building Structure in the Clause Head*:

   process_all(**Info**,[],[]).
   process_all(**Info**,[H1|T1],[H2|T2]):-
   process_one(**Info**,H1,H2),
   process_all(**Info**,T1,T2).

   where **process_one/1** takes **Info** and **H1** as input and outputs **H2**

   (a) We keep one parameter from the schema (**Info**). We rename *process_all* to **nple1** and *process_one* to **is**. We will not show how the others programs are written using the schema.

   ```
   nple1(N,[],[]).
   nple1(N,[H|T],[NewH|NewT]):-
           NewH is N*H,
           nple1(N,T,NewT).
   ```

   The declarative reading: every element in the empty list (third argument) is the given multiple (first argument) of the corresponding element in the empty list (second argument) —otherwise, the list found in the third argument position is in the desired relation to the list found in the second argument position if the head of one list

is the desired multiple of the head of the other list and the desired relation holds between the tails of these two lists.

```
nple2(N,L,Ans):-
        nple2(N,L,[],Ans).
nple2(N,[],Ans,Ans).
nple2(N,[H|T],Acc,Ans):-
        NewN is N*H,
        nple2(N,T,[NewN|Acc],Ans).
```

Again, when using the schema *Building Structure in the Clause Body* together with an accumulator, we define a predicate **nple2/3** which initialises the accumulator for **nple2/4**.

Now we have the procedural reading for **nple2/4** assuming that the output list is the fourth argument, and the other argument positions are inputs.

We return the result found in the accumulator once the input list is empty —otherwise we remove the head from the input list, multiply it by the desired amount, place the result in the accumulator and repeat the process for the tail of the input list and the new accumulator.

(b)

For this predicate, we need three cases to handle: the empty list, when the head of the list matches the element to be deleted and the case where these two elements do not match.

The first clause results from the observation that the empty list with all the occurrences of the named element removed is the empty list.

```
del_all1(ToGo,[],[]).
del_all1(ToGo,[ToGo|T],Ans):-
        del_all1(ToGo,T,Ans).
del_all1(ToGo,[H|T],[H|Ans]):-
        del_all1(ToGo,T,Ans).
```

The second clause is read declaratively as being true when the element to be deleted unifies with the head of the list (second argument) then the result of deleting all occurrences will be the same as deleting all occurrences from the tail of that list.

The third clause is the "otherwise" case: the result of deleting all occurrences from the list will is the head of the list together with the result of deleting all undesired occurrences from the tail of that list.

> There is a serious problem here. If a program which makes use of **del_all1/3** backtracks to **redo** the call to **del_all1/3** then we will get some undesirable behaviour as this definition will generate false solutions (we assume here that we always call **del_all1/3** with the second argument a list, the first argument some ground term (*i.e.* a term containing no variable) and the third argument a variable).
> Consider the query **del_all1(a,[b,a,n,a,n,a],X)**. The first solution will result in **X=[b,n,n]**. Fine. But the last 'a' was removed through a use of the second clause —the subgoal would be **del_all1(a,[a],X_1)** and originally produced **X=[]**. Now, on

redoing, we try to satisfy the goal with the third clause. The query **del_all1(a,[a],X_1)** matches with the head of the clause — **del_all1(ToGo,[H|T],[H|Ans])**— resulting in **ToGo=a, H=a, T=[] and X_1=[H|Ans]** with a subgoal of **del_all1(a,[],Ans).**. This gets satisfied with **Ans=[]** and therefore we have another solution for the query **del_all1(a,[a],X_1)** —*viz* **X_1=[a]** and this is **wrong**.

The problem arises because any query for which the second argument is a non-empty list such that its head is the element-to-be-deleted will also be guaranteed to match the head of the third clause. This means that there are ways of resatisfying the query which result in undesired (and wrong) solutions.

How do we solve this? There are two basic ways —one of which is fairly easy to read and the other relies on the use of the 'cut'.

i. add an extra test condition to the third clause to ensure that attempts to backtrack will fail. We make it impossible for a goal to simultaneously match against the same goal.

```
del_all1(ToGo,[],[]).
del_all1(ToGo,[ToGo|T],Ans):-
        del_all1(ToGo,T,Ans).
del_all1(ToGo,[H|T],[H|Ans]):-
        \+(H=ToGo),
        del_all1(ToGo,T,Ans).
```

This is straightforward but does effectively require that unification between the element-to-be-deleted and the head of the list is done twice. Fine for simple checks but this rapidly gets more expensive in more complex situations. Now for the cut.

ii. a cut (**!/0**) can be placed to say that once clause 2 has been used then never look for another match...this means:

```
del_all1(ToGo,[],[]).
del_all1(ToGo,[ToGo|T],Ans):-
        del_all1(ToGo,T,Ans),!.
del_all1(ToGo,[H|T],[H|Ans]):-
        del_all1(ToGo,T,Ans).
```

This is much less easy to read but is generally more efficient. Beginners, however, tend to spray cuts around producing code like this:

```
del_all1(ToGo,[],[]):-!.
del_all1(ToGo,[ToGo|T],Ans):-
        del_all1(ToGo,T,Ans),!.
del_all1(ToGo,[H|T],[H|Ans]):-
        del_all1(ToGo,T,Ans),!.
```

They do this because they do not understand the way the cut works. Because of this, the code written has effects they can't predict or understand. Extra, useless cuts also means a loss of efficiency. Therefore, we strongly recommend the first version.

Every program you write that is *intended* to succeed once and once only should be checked to make sure that this will happen *at the time* you write the predicate.

The second version making use of the schema *Building Structure in the Clause Body*:

```
del_all2(ToGo,L,Ans):-
        del_all2(ToGo,L,[],Ans).
```

```
del_all2(ToGo,[],Ans,Ans).
del_all2(ToGo,[ToGo|T],Acc,Ans):-
        del_all2(ToGo,T,Acc,Ans).
del_all2(ToGo,[H|T],Acc,Ans):-
        del_all2(ToGo,T,[H|Acc],Ans).
```

Again, we use **del_all2/3** to initialise the accumulator for **del_all2/4**.

> Again, note that we would need protection against unexpected backtracking if this program is to be used in another program. Again, we would want a cut in the second clause of **del_all2/4**.

(c)

Here we have the version making use of the schema *Building Structure in the Clause Head*:

```
sum1([X],X).
sum1([H|T],Ans):-
        sum1(T,RestAns),
        Ans is RestAns+H.
```

In this first version, we have a straightforward declarative reading. The first clause reads that the sum of integers in a list with a single (assumed) integer is that single integer. The second clause states that the sum of integers in a list is found by summing the integers in the tail of the list and then adding the head of the list to the result.

Now for the version making use of an accumulator.

```
sum2(X,Y):-
        sum2(X,0,Y).
sum2([],Ans,Ans).
sum2([H|T],Acc,Ans):-
        NewAcc is Acc+H,
        sum2(T,NewAcc,Ans).
```

The second version uses an accumulator. Here, we make use of **sum2/2** to call **sum3** with the accumulator initialised to 0. In this case, the first clause can br read procedurally as saying that once we have an empty list then the answer desired (third argument) is the accumulated total (second argument). The second clause states that we find the answer (third argument) by adding the head of the list (first argument) to the accumulator (second argument) and then repeating the process on the remainder of the list (with the accumulator set appropriately).

Appendix G

Solutions and Comments on Exercises for Chapter 8

G.1 Exercise 8.1

1.

```
 [the,clever,boy,buys,a,sweet]
 [the,clever,sweet,buys,a,clever,clever,boy]
(etcetera)
```

These are legitimate inputs produced by the query

```
?- s([the,clever,boy,buys,a,sweet],[]).
?- s([the,clever,sweet,buys,a,clever,clever,boy],[]).
(etcetera)
```

It is not so immediately apparent that this grammar can generate sentences as well. What is the order in which sentences are generated?

```
[a,boy,buys,a,boy]
[a,boy,buys,a,sweet]
[a,boy,buys,the,boy]
[a,boy,buys,the,sweet]
[a,boy,buys,a,clever,boy]
[a,boy,buys,a,clever,sweet]
[a,boy,buys,a,clever,clever,boy]
[a,boy,buys,a,clever,clever,sweet]
[a,boy,buys,a,clever,clever,clever,boy]
(and so on)
```

2. Here is just one example —there are various ways of doing this.

```
s(sentence(NP-VP))          -->   np(NP),
                                  vp(VP).
np(nounphrase(Det-N))       -->   det(Det),
                                  noun(N).
```

```
np(nounphrase(Det-Adjs-N))  -->  det(Det),
                                 adjs(Adjs),
                                 noun(N).
vp(verbphrase(V-NP))        -->  verb(V),
                                 np(NP).
det(determiner(a))          -->  [a].
det(determiner(the))        -->  [the].
adjs(adjectives(Adj))       -->  adj(Adj).
adjs(adjectives(Adj-Adjs))  -->  adj(Adj),
                                 adjs(Adjs).
adj(adjective(clever))      -->  [clever].
noun(noun(boy))             -->  [boy].
noun(noun(sweet))           -->  [sweet].
verb(verb(buys))            -->  [buys].
```

which produces (in a much less readable form than the following):

```
X = sentence(
        nounphrase(
                determiner(a)
                -
                adjectives(adjective(clever))
                -
                noun(boy))
        -
        verbphrase(
                verb(buys)
                -
                nounphrase(
                        determiner(a)
                        -
                        adjectives(adjective(clever)-adjectives(adjective(clever)))
                        -
                        noun(boy))))
```

3. This is very hard to do in general. The issue here is one of 'robust parsing' and it is a major research topic. Consequently, there is no complete answer but a first attempt might look like:

```
s(sentence(NP-VP))          -->  np(NP),
                                 vp(VP).
np(nounphrase(Det-N))       -->  det(Det),
                                 noun(N).
np(nounphrase(Det-Adjs-N))  -->  det(Det),
                                 adjs(Adjs),
                                 noun(N).
vp(verbphrase(V-NP))        -->  verb(V),
                                 np(NP).
det(determiner(a))          -->  [a].
det(determiner(the))        -->  [the].
det(unknown_det(X))         -->  [X],
```

```
                                        {\+(known(X))}.
adjs(adjectives(Adj))           -->     adj(Adj).
adjs(adjectives(A-Adjs))        -->     adj(A),
                                        adjs(Adjs).
adj(adjective(clever))          -->     [clever].
adj(unknown_adj(X))             -->     [X],
                                {\+(known(X))}.
noun(noun(boy))                 -->     [boy].
noun(noun(sweet))               -->     [sweet].
noun(unknown_noun(X))           -->     [X],
                                        {\+(known(X))}.
verb(verb(buys))                -->     [buys].
verb(unknown_verb(X))           -->     [X],
                                        {\+(known(X))}.

known(X):-noun(noun(X),_,_).
known(X):-verb(verb(X),_,_).
known(X):-det(determiner(X),_,_).
known(X):-adj(adjective(X),_,_).
```

Some points to note:

(a) It cannot cope with missing words so this goal fails badly. We could try to extend it to meet this problem. For example, we might like the following query to succeed:

```
s(X,[the,clever,buys,a,sweet],[]).
```

(b) It does cope quite well with more than one misspelling provided the sentence structure is acceptable —as in the query:

```
s(X,[the,clever,silly,buoy,buys,a,sweet],[]).
```

(c) The known/1 predicate is not at all clever.

Appendix H

Solutions and Comments on Exercises for Chapter 9

H.1 Exercise 9.1

1. First, we examine the execution of the query **female_author**. We take the first clause for **female_author/0** and solve for **author(X)**. We use the first clause of **author/1** and solve the resulting subgoal, **name(X)**, using the first clause of **name/1** to get **X=sartre**. The subgoals **write(X)**, **write(' is an author')** and **nl** succeed with the side-effect of writing:

   sartre is an author

   on the screen. Then we solve the subgoal **female(X)** with **X** still bound to **sartre**. Neither of the heads of the clauses for **female/1** match the goal **female(sartre)** so **Prolog** fails and backtracks. We keep backtracking until we get to **redo** the subgoal **author(X)**. This means that we now try to **redo name(X)** and we satisfy this with **X=calvino**. Again, we generate the side-effect on the screen of

   calvino is an author

   and try to satisfy **female(X)** with **X** bound to **calvino**. Again, this fails and we backtrack. Again, the subgoal **name(X)** can be satisfied —this time, for the last time, with **X=joyce**. On the screen we get

   joyce is an author

   and another attempt to prove that **female(X)** with **X=joyce** (which fails). This time, on backtracking, there are no more solutions for **name(X)**. We now move on to resatisfy **author(X)** by using its second clause. This generates, on the screen,

   no more found!

   then fails. We now backtrack and, since there are no more ways of satisfying **author(X)**, we are through with the first clause of **female_author/0**. The second succeeds by writing

no luck!

and succeeds.

We now explain how to place the cuts to get the desired outputs.

(a) The first solution requires the use of one cut to produce the side-effect:

sartre is an author
no more found!
no luck!

Note that, compared with before, we have much the same but we infer that there is only one solution for **name(X)**. This suggests that we place the cut so that **name(X)** succeeds once only. This can be done by rewriting the first clause of **name/1** as

name(sartre):-!.

(b) The next solution requires the use of one cut to produce the side-effect:

sartre is an author
calvino is an author
joyce is an author
no more found!

Compared with the original output, we observe that the phrase 'no luck!' is not generated. This suggests that we want to *commit* ourselves to the first clause of **female_author** and not use the second at all. Hence we have the solution:

female_author:- !,author(X),write(X),and so on

but note that now the original query fails after producing the desired side-effect.

Also note that we have to put the cut before the call to **author/1** —otherwise we would only generate one of the names rather than all three.

(c) The next solution requires the use of one cut to produce the side-effect:

sartre is an author
no luck!

This time we observe that we only get one name and we don't generate the phrase 'no more found!'. This suggest that we want **author(X)** to succeed once and once only —and go on to use the second clause of **female_author** (this suggests that the cut cannot be one of the subgoals in the first clause of **female_author**). We don't want to generate the phrase 'no more found' —so this suggests that we commit to the first clause of **author/1**. We will put a cut in the body of this clause —but where? If we put it thus:

author(X):- !,name(X).

then we would generate all three names by backtracking. Hence the desired solution is:

```
author(X):- name(X),!.
```

We can read this as being committed to the first, and only the first, solution for **name(X)**.

(d) The next solution requires the use of one cut to produce the side-effect:

```
sartre is an author
```

Now we don't want to generate either 'no more found!' or 'no luck!' —and we only want one of the names generated.

So we definitely want to be committed to the first clause of **female_author/0**. This suggests putting the cut in the body of the first clause of this predicate —but where? If we put it after the subgoal **female(X)** then we would get all three solutions to **name(X)** and their associated side-effects. If we put it before **author(X)** we also get roughly the same. Therefore we want something like:

```
female_author:- author(X),!,write(X),and so on
```

(e) The next solution requires the use of one cut to produce the side-effect:

```
sartre is an author
calvino is an author
joyce is an author
no luck!
```

Now we don't get the message 'no more found!'. This suggests that we want to commit to the first clause of **author/1**. If we put the cut after the subgoal **name(X)** then we will commit to the first solution and not be able to generate the other two. Hence we must put the cut before as in:

```
author(X):- !,name(X).
```

2. We will assume a mode of **mode delete(+,+,?)**. *i.e.* the first two arguments are always completely instantiated and the third argument can be either instantiated or a variable.

```
delete([], _, []).
delete([Kill|Tail], Kill, Rest) :-
        delete(Tail, Kill, Rest),!.
delete([Head|Tail], Kill, [Head|Rest]):-
        delete(Tail, Kill, Rest).
```

The cut is placed in the body of the second clause. This is needed because, in the code given in the probem, any usage of the second clause to delete an element allows the query to be resatisfied on **redo**ing. This is caused by the fact that any query matching the head of the second clause will also match the head of the third clause.

3. To define **disjoint/2**, here is the solution found in the standard DEC-10 library.

```
%       disjoint(+Set1, +Set2)
%       is true when the two given sets have no elements in common.
%       It is the opposite of intersect/2.

disjoint(Set1, Set2) :-
        member(Element, Set1),
        memberchk(Element, Set2),
        !, fail.
disjoint(_, _).

member(X,[X|Rest]).
member(X,[Y|Rest]):-
        member(X,Rest).

memberchk(X,[X|Rest]):- !.
memberchk(X,[Y|Rest]):-
        memberchk(X,Rest).
```

Note the definition is quite interesting: First, note that we make use of the *generate—test* schema: the first clause of **disjoint/2** uses **member/2** as a (finite) *generator* to generate each of the elements in the first list one by one. Next, the solution is *test*ed using **memberchk/2** for the second set. Second, if the element is in both sets then we meet the *cut—fail* schema. This means the call to **disjoint/2** fails. If the *generate*d element *never* passes the *test*, then the attempt tosatisfy the call to **disjoint/2** using the first clause fails and we use the second clause which makes use of a catch-all condition.

Other solutions are possible:

```
disjoint(Set1, Set2) :-
        \+(member(Element, Set1), member(Element, Set2)).
```

Here, disjoint succeeds whenever it is impossible to find a common element for both lists. If such an element exists then it fails.

4. To define **plus/3**:

```
plus(A, B, S) :-
        integer(A),
        integer(B),
        !,
        S is A+B.
plus(A, B, S) :-
        integer(A),
        integer(S),
        !,
        B is S-A.
plus(A, B, S) :-
        integer(B),
        integer(S),
        !,
```

```
A is S-B. plus(A, B, S) :-
plus_error_message(A, B, C).
```

The first three clauses cope with the cases where two or more of the arguments to **plus/3** are instantiated to integers.

We need the cut to implement a *commit* schema as we don't have disjoint cases.

The last clause is intended to point out to the programmer that an instantiation fault has occurred. The exact form of the message is up to you.

Appendix I

Solutions and Comments on Exercises for Chapter 11

I.1 Exercise 11.1

1. Here is the answer in **Prolog** Form:

```
:(rule31,
        if(then(and(=(of(colour,wine),white),
                and(or(=(of(body,wine),light),          =(of(body,wine),medium)),
                        or(=(of(sweetness,wine),sweet),=(of(sweetness,wine),medium)))),
        then    confidence_factor(=(wine,riesling),1000))))
```

   The tree is up to you!

Appendix J

Solutions and Comments on Exercises for Chapter 12

J.1 Exercise 12.1

1. We want the following behaviour:

   ```
   ?- diff_append([a,b|X]-X,[c,d,e|Y]-Y,Answer).
   Answer = [a,b,c,d,e|Y] - Y
   ```

   Here is the solution:

   ```
   diff_append(List1-Tail1,Tail1-Tail2,List1-Tail2).
   ```

2. We want the following behaviour:

   ```
   ?- add_at_end(e,[a,b,c,d|X]-X,Answer).
   Answer = [a,b,c,d,e|Y] - Y
   ```

   Here is the solution:

   ```
   add_at_end(X,List-Tail,List-NewTail):-
           Tail=[X|NewTail],!.
   add_at_end(X,List-[X|NewTail],List-NewTail).
   ```

   Note the need to add the cut (**!/0**) to prevent unwanted behaviour on backtracking to **redo** a call to this predicate.

3. We want the following behaviour:

   ```
   ?- diff_reverse([a,b,c|X]-X,Answer).
   Answer = [c,b,a|Y] - Y
   ```

There are two cases: the first covers the case where we have the difference list equivalent of the empty list.

The second case covers every other situation: we have to take off the head element, reverse the remaining difference list and then stick the element at the end of the difference list. We use **add_at_end/3** that we have already defined.

Here is a solution:

```
diff_reverse(X-X,Y-Y):-
        var(X),!.
diff_reverse([H|List]-Tail,Answer):-
        diff_reverse(List-Tail,NewDiffList),
        add_at_end(H,NewDiffList,Answer).
```

Note that, for the first clause, we state that the tail of the input is a variable (via **var/1**). The use of cut (**!/0**) is necessary to stop unwanted behaviour if we ever backtrack to **redo** this goal.

Also note that we will get nasty behaviour if the predicate **add_at_end/3** has not been defined to prevent unwanted behaviour on backtracking.

4. We want the following behaviour —assuming that the list is composed of integers or atoms or lists of these. This means that every element is either the empty list, a list or some **Prolog** term that is *atomic*. *i.e.* the term satisfies **atomic/1**.

```
?- diff_flatten([1,2,[3,4,[5,4,[3],2],1],7|X]-X,Ans).
Ans=[1,2,3,4,5,4,3,2,1,7|Z]-Z
```

We have three cases: we are going to flatten the empty list by outputting the difference list version of [] —*i.e.* **X-X**. We flatten an 'atomic' element other than the empty list by returning a difference list with a single element —*viz* **[someelement|X]-X**. The third case is designed to handle the case where the head of the first argument is a list. In this case, we flatten the head, and then flatten the tail.

Here is a solution:

```
diff_flatten([H|T],X-Y):-
        diff_flatten(H,X-Z),
        diff_flatten(T,Z-Y).
diff_flatten(X,[X|T]-T):-
        atomic(X),
        \+(X=[]).
diff_flatten([],X-X).
```

Note how the result of flattening the head is a difference list with a hole. We get the same for flattening the tail and join the lists together by identifying the hole for the flattened head with the open list resulting from flattening the tail.

5. We want the following behaviour:

```
?- diff_quicksort([3,1,2|X]-X,Ans).
Ans=[1,2,3|Z]-Z
```

We should note that the obvious advantage in using difference lists here is in avoiding the various calls to **append/3** which can get very expensive. On the other hand, we have to realise that there is an overhead in carrying difference lists around in this form.

Here is a solution:

```
diff_quicksort(X-X,Y-Y):-
        var(X).
diff_quicksort([H|T]-Hole1,Ans-Hole2):-
        diff_split(H,T-Hole1,SmallDiffList,BigDiffList),
        diff_quicksort(SmallDiffList,Ans-[H|Z]),
        diff_quicksort(BigDiffList,Z-Hole2).

diff_split(_,X-X,Y-Y,Z-Z):-
        var(X).
diff_split(X,[Y|Tail]-Hole1,[Y|Small]-Hole2,BigDiffList):-
        X > Y,
        diff_split(X,Tail-Hole1,Small-Hole2,BigDiffList).
diff_split(X,[Y|Tail],SmallDiffList,[Y|Big]-Hole2):-
        X =< Y,
        diff_split(X,Tail,SmallDiffList,Big-Hole2).
```

The correspondence between the difference and proper list versions is close.

Wherever we have an empty list ([]) we introduce the difference list version (**X-X**). We have to be careful to distinguish *unrelated* empty lists. For example, consider the first clause of **quicksort/2** in the notes: **quicksort([],[])**. The correspondence might suggest **diff_quicksort(X-X,X-X)**. But in this case, there can be very unpleasant consequences as we are forcing certain variables to be identical that might have been distinct up to the point of solving the goal **diff_quicksort(A,B)**. Consequently, we introduce different version of the difference list 'empty list' and write **diff_quicksort(X-X,Y,Y)**.

Note that the 'stopping condition' is that we have run out of elements to sort by reaching the 'hole'. This means the test is for having encountered a variable. So the procedural reading of the first clause is effectively: return a difference list equivalent to the 'empty list' when we find that we have consumed all the non-variable elements from the front of the list. To do this, we use **var/1** which takes a **Prolog** term as input and is true whenever that term is an uninstantiated variable (it can be bound to another uninstantiated variable though).

The remaining two clauses are structurally very similar to the last two clauses for **quicksort/2** as in the notes. The main difference is the loss of the call to **append/3** and the means by which we can partially 'fill in' the hole of the **Small** difference list with the result of sorting the **Big** difference list.

As for, **diff_split/4**, we have a very similar situation which we not explain further here.

What about the efficiency? Is **diff_quicksort/2** faster than **quicksort/2**? And are we comparing *like with like*? The empirical answer is that **quicksort/2** is faster (there are ways of improving the efficiency of the above version of **diff_quicksort/2** but they are not sufficient)!

One reason why this version of **diff_quicksort/2** is slower than **quicksort/2** is that the former predicate transforms a difference list into a difference list while the latter transforms a proper list to a proper list. An improvement is achieved by writing a version that takes a proper list to a difference list as with:

```
diff_quicksort_v2([],[]).
diff_quicksort_v2([H|T],Ans-Hole2):-
        split(H,T,SmallProperList,BigProperList),
        diff_quicksort_v2(SmallProperList,Ans-[H|Z]),
        diff_quicksort_v2(BigProperList,Z-Hole2).
```

Note that we can now use the same **split/4** as for **quicksort/2**.

The efficiency of this version is now better than the performance of **quicksort/2**.

6. This one is up to you!