Artificial Intelligence Programming in Prolog

Tim Smith

September 26, 2004

School of Informatics,
University of Edinburgh.

Acknowledgements: This document was originally entitled *An Introduction to Prolog* by Helen Pain and Robert Dale. Early versions were based on *Prolog Step-by-Step* by Graeme Ritchie and *Quick Prolog* by Dave Robertson, Mandy Haggith, Louise Pryor & Geraint Wiggins. The structure, content, and practical exercises in this version have received major revision by Tim Smith.

Contents

1 Course Details

1.1 Schedule

The Artifical Intelligence Programming in Prolog course consists of 19 one-hour lectures (**Monday and Thursday 4pm**) beginning on Thursday 23rd September. All lectures will be held in the **A9/11, ground floor, Forrest Hill**.

Students are required to attend one 2hr practical session per week (beginning in week 2). Practical sessions will be held in **Appleton Tower, level 5, Computer Lab West on Wednesday's 4-6pm and Friday's 3-5pm.** Practical exercises allow students to get hands-on experience of the topics discussed in lectures and aquire the skills necessary for completion of the assessed assignments. Each practical exercise will contain sections that need to be shown to the practical demonstrator and logged as complete. These exercises are not formally assessed or kept on file but their completion is compulsory.

1.2 Assessment

This course will be examined during the summer term exam period. The examination counts for 70% of the course mark. The other 30% is split between two assignments.

- **Assignment 1: An automatic puzzle solver.** This assignment is worth 10% of the course mark. It will be handed out at the end of week 2 and is due in **Monday 1st November at 4pm**.
- **Assignment 2: Six Degrees of Kevin Bacon.** This assignment if worth 20% of the course mark. It will be made available during week 6 and is due in **Friday 3rd December at 5pm**.

Both assignments will be submitted electronically using the **submit** command on DICE. Details of how to use this will be provided with each assignment. Penalties will be awarded for late submissions and extensions will only be given in extreme circumstances.

1.3 Requirements

All students who choose to take AIPP for credit must have previous programming experience. Students who have not previously learnt a programming language or who don't know Java specifically are advised to take Introduction to Java Programming (IJP). Students may also opt to take both and this will be accomodated by the lecturers. Any student who questions whether their level of ability is sufficent for AIPP should speak to me in person or contact me by e-mail (`tim.smith@ed.ac.uk`).

If students have previously studied Prolog or used it in industry then they can be made exempt from taking AIPP. Students should discuss their case for exemption with myself.

1.4 Materials

All course material including lecture notes, practicals, assignments, and important information will be available on the course website.

`http://www.informatics.ed.ac.uk/teaching/courses/aipp/`)

Practical exercises are also included in this document after the corresponding chapters.

This document constitutes the main reference for the AIPP course. All information within this document is examinable unless otherwise indicated. However, the concepts within this document will be explored and extended in the lectures which makes attendance to lectures compulsary if you are to learn the appropriate material.

There is no required text book but there are three which I can recommend:

[Clocksin & Mellish 03] A good basic introductory text. This is the most up to date text book and conforms to the new ISO standard Prolog. This is the form of Prolog that will be taught in AIPP. If you choose to use a text book other than Clocksin and Mellish refer to the course notes for correct syntax.

[Bratko 01] An introductory book that leans heavily towards AI applications. It is more chatty than Clocksin and Mellish, but has the major disadvantage of frequently using a poor programming style. A major source of example AI programs.

[Sterling & Shapiro 94] Possibly the best general Prolog book around, but definitely not an introduction, especially if you don't have much programming experience.

1.5 Software

There are many different implementations of Prolog available, each slightly different. Fortunately there is a language core that is provided by most implementations (implementations you may come across include Quintus, SICstus, Arity, LPA MacProlog and NIP, among others). The standard implementation for this module is SICStus Prolog 3, which is available on departmental machines by typing `sicstus`.

A free version of Sicstus Prolog for Windows is available for all Informatics students. Fill in the Informatics Support form to request a copy.

`http://www.inf.ed.ac.uk/cgi-bin/support.cgi`

You are free to work on assignments on a Windows PC but all submitted code should work on the DICE installation of Sicstus. Therefore, please test all code on the DICE system before submission.

2 Introduction

2.1 How Do You Learn A Programming Language?

Artificial Intelligence is a rapidly changing field. It is constantly borrowing from the latest technologies and being applied to new areas. As a result of this it is impossible to predict what language you may ultimately end up writing AI programs in — new ones will certainly come along and old ones will change. Even the major high level languages used in AI (Prolog and Lisp) are being continually upgraded and released in slightly different versions, for different machines and computer architectures. For example, whereas Prolog was designed years ago in Edinburgh[1] to run on DEC-10 mainframe computers, you can now get versions of Prolog which run on a small PC or Macintosh and other versions which run on giant, massively parallel architectures.

As a result of this, the most important skill for AI programmers is not knowledge of a particular language, but the ability to *learn* new programming languages quickly, effectively and quite possibly often! This is not an innate talent; it is a skill that can be learned. Useful approaches include the following:

- Getting the *gist* of a language before you begin. Some of the questions you might want to ask, together with possible answers, are:

  What sort of language is this? A logic programming language, an imperative language, an object oriented language, a high-level language, a low-level language, compiled, interpreted;

  What sorts of things can it do for me? Pattern matching, searching, efficient data structures, good interface tools etc.;

  What is the main control mechanism?

  What is the syntax of this language?

  How are variables handled?

  How do I write comments?

- Using reference manuals efficiently, by guessing keywords, reading only a brief overview then using the index, skimming to get ideas;
- Finding out about debugging tools early on, before you write that huge bug-ridden program;
- Starting with small programs, or bits of programs, and getting them working before you move on. Not only does this make you feel satisfied, and reminds you that you have learnt something already, but it is also sound methodology. A big program that is a collection of small *working* programs is much more likely to work than a huge program which has evolved in a permanently buggy form;
- Thinking of simple solutions to problems, or simplifications of existing solutions. There is absolutely no merit in a complicated or long program where a shorter or simpler one will suffice;

[1] In fact it was conceived in Marseilles, though not as a general purpose programming language

- Talking to people who know the language. People respect honest ignorance and are usually flattered by being asked advice (within reason!);
- Looking at "good" programs and trying to see what is elegant about them: an aesthetic sense for the language often corresponds well with either reliability, efficiency or both;
- Always laying out your own programs in as clear a manner as possible, taking pride in your work, and annotating them with comments;
- Learning a little and often and in varied ways. If your program doesn't work, write down the problem and then leave it alone till tomorrow, think about the program away from the computer, leaf through the most appropriate chapter in the reference book, think about how you wished it worked, make sure you know as much as you could about the debugger ...only then return to your program;
- Practising and being honest about your abilities. Don't expect to be able to get an entire language sussed in one day; practice a little and often and try to learn from your mistakes; give yourself a mental pat on the back when you get things right and your program works; keep a positive but not unrealistic attitude to your current level of ability.
- Finally, and most importantly, by trying to get the feel of the language, in a very informal sense, at a very high level. This is important as you proceed to become more familiar with the detail of the language – it gives you a framework from which to hang things.

2.2 The gist of Prolog

Some facts about Prolog:

- Prolog is a *high-level logic programming* language (PROgramation et LOGique);
- You interact with the Prolog system directly by typing commands directly into the terminal;
- Good at *pattern matching* (by *unification*) and *searching*;
- Not very good for repetitive number crunching;
- Excellent for language processing, rule-based expert systems, planning and other AI applications;
- Uses *depth-first search* and *backtracking* to search for solutions automatically;
- Best written in little chunks (modular code): indeed this is assumed in its syntax;
- Uses a `%` to prefix comments or `/* ... */` to surround them.

The three most important concepts in Prolog are *unification*, *backtracking* and *recursion*. If you understand these concepts thoroughly you can probably write pretty good Prolog programs.

3 Prolog Basics

3.1 Main concepts

Some of the main Prolog concepts that you will be introduced to are:

- facts;
- questions;
- logical variables;
- matching (or 'unification');
- conjunctions; and
- rules.

Each of these is described in more detail below.

3.2 An example Prolog program

Figure 1 shows an example of a Prolog program, along with some of the dialogue that might take place between the user and the Prolog interpreter.

```
| ?-
```

This is the Prolog prompt which informs the user that the program expects some input (it isn't currently processing anything). The rest of the file is made up of **Facts** and **Comments**. Facts are always followed by a **full stop**. A line of comments has to be preceded by '%'. Several lines can be commented out of your program (so that the compiler ignores them) by preceding the section with '/*' and ending it with '*/'. The 'yes' and 'no' statements are made by the Prolog interpreter in response to user queries (the mechanics of this will be explained more later).

3.3 Facts, Questions and Variables

3.3.1 Facts

A **fact** asserts some property of an object, or states some relation between two (or more) objects; it states that something is known to be true.

A fact is made up of a **predicate** (which states the relation or property) and a number of **arguments** (which are the objects): see Figure 2.

A fact can contain any number of arguments; so, for example:

`drinks(alan,beer,export,lorimers).`	has four arguments
`likes(alan,coffee).`	has two arguments
`listing.`	has no arguments

```
?- listing.                        % lists the program

hates(heather, whisky).            % heather hates whisky

likes(alan, coffee).               % alan likes coffee
likes(alan, whisky).               % alan likes whisky
likes(heather, gin).               % heather likes gin
likes(heather, coffee).            % heather likes coffee

drinks(alan, beer).                % alan drinks beer
drinks(heather, lager).            % heather drinks lager

yes
?- likes(alan, coffee).            % is it true that alan likes coffee?

yes                                % it is true that alan likes coffee
?- drinks(heather, lager).         % is it true that heather drinks lager?

yes                                % it is true that heather drinks lager
?- drinks(alan, lager).            % is it true that alan drinks lager?

no                                 % it is not true that alan drinks lager
```

Figure 1: An interaction with the Prolog interpreter

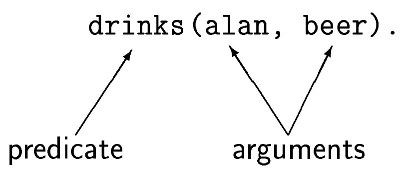


Figure 2: The structure of a fact

We refer to the number of arguments in a fact as its **arity** which we write in the form **drinks/2** . This tells us that the fact 'drinks' takes two arguments or, in other words, an arity of two.

A Prolog program may have any number of facts. When a program is given to the Prolog interpreter (loaded into Prolog, or **consulted**), the set of facts that are part of the program represent all that is known to be true. They are really a set of **logical assertions**. Together they are often referred to as the **database**. They have no real 'meaning', however: the person writing the program defines their own interpretation of what they mean. Whilst it might seem silly to interpret

```
drinks(alan,beer)
```

as *drinks alan beer*, there is nothing to stop us using this interpretation— it's all the same to Prolog. What is important is to be consistent, so we have to keep arguments that are meant to refer to the same objects in the same argument slots.

The best way to be consistent and make your code readable is to think of your code as a **Subject Verb Object** construction in English. The predicate should always be the verb as it can then be applied to many subjects and objects. The **Subject** should always be the first argument and the **Object** the second argument. If you stick to this format then your code will be easily readable by others.

3.3.2 Questions

If a question is asked by the user, the Prolog interpreter looks at the database of facts[2] that it has to see if there is enough information to answer it. Given the example program we saw in Figure 1, we might ask questions like the following:

- What does Heather drink?
- Does Alan like coffee?
- Who drinks whisky?

The Prolog interpreter doesn't understand English, however, so we have to re-express these questions in Prolog itself. Taking the second of these questions as an example, we ask:

```
?- likes(alan, coffee).
```

and the Prolog interpreter replies:

```
yes
```

The Prolog interpreter matches the question to each fact (or assertion) in the database, as follows:

[2]The database can also contain **rules**, as we will see later.

1. First, Prolog finds a fact that matches the predicate in the question.
2. If this match succeeds, Prolog then matches the first argument to the predicate.
3. If this match succeeds, Prolog matches the second argument, and so on for the rest of the arguments.
4. If the match fails at any point, Prolog looks for the next assertion that the predicate matches and tries again to match the arguments.
5. If the predicate and all the arguments are successfully matched, the process stops and the interpreter prints `yes`, meaning 'there is a match—I can show this to be true'. The goal of finding a match has been satisfied.
6. If there is no match at all then `no` is printed, meaning 'there is no match': the goal of finding a match cannot be satisfied.

3.3.3 Variables

Suppose the question we want to ask is *What does Heather like?*; some way has to be found of asking the 'what'. The goal is to find some 'what' such that Heather likes it; anything that satisfies the question will do. In Prolog the question becomes:

```
?- likes(heather,What).
```

Note that the '`What`' begins with a capital letter. This is to indicate that it is a **variable**. A variable is a placeholder that can take on any value through instantiation. It performs a similar function to the word *thing* in English, or x in algebra.

You will all be familiar with the concept of variables from other programming languages but in Prolog variables are a lot more flexible. You do not have to declare variable types, initialise them, or define their scope (e.g. local vs. global). **To create a variable in Prolog all you have to do is write a string which begins with a capital letter.** Because of this simplicity you have to ensure that all other elements of your code start with lower case letters otherwise you will create unintended variables.

The other arguments encountered until now represent particular people or specific objects:

```
alan
heather
whisky
gin
coffee
lager
beer
```

These do not change: they always represent the same object, and so are called **constants**.

Both variables and constants are examples of structures called **atoms**. They cannot be broken down into smaller objects that mean anything to Prolog.

When a match succeeds, any variables are given the value of any constant they are matched to. This matching process is one of the very powerful facilities that comes free with Prolog; the Prolog interpreter does this matching automatically and remembers what is matched to what.

When a variable has no value, we refer to it as an **uninstantiated variable**. When an uninstantiated variable is returned to the user it is represented as an underscore ('_') followed by a number randomly allocated to it by the prolog interpreter (e.g. '_45'). When it gets given a value by the matching process, it is referred to as an **instantiated variable**.

So, if we match

```
?- likes(heather, What).
```

to

```
likes(heather, gin).
```

then we say that `What` has become instantiated to `gin`. The Prolog interpreter prints out the values of any variables that have become instantiated in the process of matching:

```
What = gin
```

Note that the same variable can appear in different questions and provide different answers. This is because the variable names only apply to each question: this is referred to as the **scope** of the variable. In Prolog, the scope of the variables is said to be **local**, that is, they only have the same value in a limited (local) environment. All the variables in the same question with the same name become instantiated at the same time to the same value. Any instance of the variable in different questions/parts of the prolog program are independent and do not become instantiated. Don't worry if this seems strange to begin with; like a lot of Prolog, it takes some time to get used to.

3.4 Prolog Terminology and Syntax

Prolog programs consist of propositional statements. These statements are based on **Horn Clauses** or **clauses** for short. Horn Clauses are a form of logical representation that denotes an implication between two or more positive facts and a single positive facts.

$$(parent(X, Z) \wedge ancestor(Z, Y)) \supset ancestor(X, Y)$$

In Prolog the range of relationships that can be represented is expanded. This permits three types of **clauses**:

- **facts** declare things that are always true.
- **rules** declare things that are true depending on a given condition.

- **questions** test if a particular goal is true.

These clauses are all constructed using other elements that can be referred to using a hierarchy of terminology.

'Term' is used to refer to any data object in Prolog.

- a constant is a term
- a variable is a term
- a compound term is a term (e.g. a fact)

Constants can be:

- atoms
- integers
- real numbers

Atoms are made up of:

- letters
- digits
- the underscore
- symbols (+, -, *, /, \, ^, <, >, =, ~, :, ., ?, @, #, $, &)

A quoted string is also an atom.

Constants are used to refer to objects. Constants begin with lower case letters. Note that all predicate names are constants—don't use a variable for a predicate:

```
X(jane, jim).
```

Certain conventions are used when writing Prolog—in other words, Prolog, like other languages, has a **syntax**:

- All predicates start with a lower case letter.
- All variables begin with an upper case letter.[3]
- The format of each fact or assertion is:
  - a predicate followed by any number of arguments;
  - the arguments are separated by commas and enclosed by round brackets;

[3] Later, you will discover other ways of writing variables.

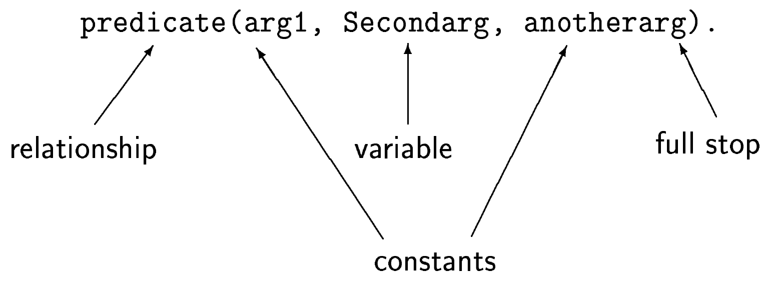


Figure 3: The elements of a fact

- there is **no space** between the predicate and the opening bracket; and
- a full stop follows the enclosing round bracket.

This is shown diagrammatically in Figure 3.

- The predicate can be a string like `son_of`, `drinks`, `likes`, and so on.
- The arguments can be **constants**, **variables**, or even other assertions.

The Prolog interpreter prints the characters '`| ?-`' to indicate that it is waiting for the user to ask a question or to set a goal to be solved; we refer to these characters as the Prolog **prompt**.

3.5 Expressing Relationships in Prolog

Suppose I want to say:

the capital of paris is France

In Prolog we might write this as:

```
has_capital(france, paris).
```

where we are expressing a relationship `has_capital` between 2 arguments, `france` and `paris`. Note that the full stop is used to terminate the clause, and that objects are referred to with words beginning with lower case letters. We can express a relationship with more than two things:

```
ate(robert,curry,breakfast).
```

Or one argument relations (usually called predicates):

```
ate_curry_for_breakfast(robert).
robert_ate_for_breakfast(curry).
robert_ate_curry(breakfast).
```

Or 0-argument relations:

```
robert_ate_curry_for_breakfast.
```

It is all a matter of how you choose to represent the relationships, which will depend on what you want to do with them. For example, as the predicate becomes more specific the number of instances the predicate can be applied becomes less and more rules will have to be written to accomodate all the exceptions. This leads to a long and repetitive program.

3.6 Unifying Terms

The process of matching is also referred to as **unification**. When two terms match we say that they **unify**. It is by the process of matching or unification that variables get instantiated. Unification is a two-way matching process. It operates on any pair of Prolog **terms**. For example, unifying `loves(john, X)` and `loves(Y, mary)` results in `loves(john, mary)`. There is a predicate defined for us in Prolog that allows us to test whether or not two things unify. Because it is provided by the system it is known as a **system predicate**. It is the infix predicate `=/2`. Note that **infix** means that it is placed between the two things that are being unified, and the `/2` is known as the **arity** of the predicate, and indicates how many arguments the predicate has. So, the predicate `=/2` takes two arguments and tries to unify them.

For example, we can unify the following sets of terms, resulting in the outcomes shown. Some comments are given in () after each match.

```
        Pairs of terms                          Outcome

1a.     ?- fred=X.                              X=fred  yes
        (the variable X unifies with the constant fred)

1b.     ?- c=letter(c).                                 no
        (a constant cannot unify with a one argument predicate)

1c.     ?- f(tee)=f(S).                         S=tee  yes
        (the predicate names are the same; the variable S unifies with
        the constant tee)

1d.     ?- father(john,tom)=father(tom,Who).            no
        (the constants john and tom cannot unify)

1e.     ?- centre(a,X,c)=centre(Y,b,c).         Y=a X=b  yes
        (the constant a unifies with variable Y, constant b with X)
```

```
1f.     ?- colour(N,N)=colour(green,X).         N=green  X=green  yes
        (variable N unifies with constant green; variables N and X
        match, so X becomes instantiated to green also. A
variable will always unify with another variable. We refer to them then
as shared variables. If one then unifies with a constant or term, the
other shares that value also.).

1g.     ?- first(sue,dave,bob)=first(N,dave,N).         no
        (variable N unifies with constant sue, but it then will not
        match in the 3rd argument with another constant bob)

1h.     ?- drink(beer(lorimers,eighty),Pub)=drink(What,mathers).
                        What=beer(lorimers,eighty)  Pub=mathers
        (the first argument here is a term, which will unify with the
        variable What)

1i.     ?- havecar(metro)=havecar(Yes).         Yes=metro
        (variable Yes and constant metro unify)

1j.     ?- f(X,Y,z)=f(z,a,Z).                   X=z Y=a Z=z  yes
        (X unifies with z in the first argument; Y with a in the 2nd and
        Z with z in the 3rd.)
```

If you wanted to check if two terms **don't** unify then you need to negate the unification operator. Negation of any term can be achieved by prefixing it with the built-in operator \+.

```
?- \+fred=X.
no.
?- \+fred=sue.
yes.
?- \+first(sue,dave,bob)=first(N,dave,N).
true?
yes.
```

Notice that the value of the variables don't change if the terms are found to not unify. Can you think how this feature could be used to test whether two terms unify without instantiating any of the variables being tested?

There is also a not-equal-to operator `\==/2` which is placed between two terms just like the equal-to operator `=/2`. This functions slightly differently and so its discussion will be saved for the later section on Arithmetic Operators.

3.7 Conjunctions

3.7.1 Asking Questions that Contain Conjunctions

Earlier we saw how to ask Prolog simple questions. Suppose we wanted to ask more complex questions like the following:

> Is it true that both Alan and Heather like coffee?
> Is there anything that Heather hates but Alan likes?

To answer these questions, we have to break them down into simpler questions, and ask each of them, one after the other. In Prolog we do this as follows:

```
| ?- ⟨question1⟩, ⟨question2⟩.
```

So, for example, we might write:

```
| ?- likes(heather, coffee), likes(alan, coffee).

yes
| ?- likes(heather, gin), likes(alan, whisky).

yes
| ?- likes(alan, beer), likes(heather, beer).

no
```

Prolog takes each subgoal in the query (one at a time, going from left to right) and an attempt is made to satisfy it. As each subgoal succeeds, the next is tried. If all the subgoals succeed, then the question as a whole succeeds. If at any point one of the subgoals fails, then the whole question fails.

The comma in the query here is read as *and*, and is referred to as the **conjunction** (the subgoals being **conjoined subgoals**). So, if we want to show that two assertions are both true then we first have to show that they are each true in isolation. In logic, this is equivalent to saying that:

> A and B is true if A is true and B is true.

3.7.2 Using Variables in Conjunctions

As with simpler questions, variables can appear in conjunctions too. However, if a variable appears more than once in a series of conjoined goals it will always match to the same value: i.e., if the variable `What` is instantiated to `beer` in one subgoal it will also be instantiated to `beer` in all other subgoals in the same question.

```
| ?- likes(heather, What), likes(alan, What).

X = coffee ?
yes
| ?- likes(alan, Something), hates(heather, Something).

Something = whisky ?
yes
```

Variables with different names may or may not share the same instantiations, depending on what they match to:

```
| ?- drinks(alan, X), likes(heather, S).

X = beer
S = gin ?
yes
| ?- drinks(X, Y), hates(X, Y).

no
```

Note that a query can contain any number of conjoined goals.[4]

3.8 Summary

The basic mode of operation of Prolog is thus as follows.

1. We provide Prolog with a question.
2. Prolog matches the question with assertions in a database,
   - looking for exact matches of predicates and constants; and
   - looking for variables to match to anything (including other variables).

In the next sections, we look at how Prolog works, then go on to introduce the notions of **rules** and **backtracking**.

3.9 Exercises

These exercises are provided to assist your learning of the concepts discussed in this chapter. They are not assessed but undertaking them is encouraged. Answers to all exercises are given in the final chapter of these notes.

[4]This is why when you forget the full stop and go on to the next line it doesn't matter: you might have wanted to have a lot of subgoals that would not all fit on one line.

Question 3.1 The predicate =/2 takes 2 arguments and tries to unify them. For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

`3.1a` `?- Pear = apple.`

`3.1b` `?- car = beetle.`

`3.1c` `?- likes(beer(murphys),john) = likes(Who,What).`

`3.1d` `?- f(1) = F.`

`3.1e` `?- name(Family) = smith.`

`3.1f` `?- times(2,2) = Four.`

`3.1g` `?- 5*3 = 15.`

`3.1h` `?- f(X,Y) = f(P,P).`

`3.1i` `?- a(X,y) = a(y,Z).`

`3.1j` `?- a(X,y) = a(z,X).`

Question 3.2 The following Prolog program is consulted by the Prolog interpreter.

```
vertical(seg(point(X,Y),point(X,Y1))).
horizontal(seg(point(X,Y),point(X1,Y))).
```

What will be the outcome of each of the following queries?

`3.2a` `?- vertical(seg(point(1,1),point(1,2))).`

`3.2b` `?- vertical(seg(point(1,1),point(2,Y))).`

```
3.2c     ?- horizontal(seg(point(1,1),point(2,Y))).
```

```
3.2d     ?- vertical(seg(point(2,3),P)).
```

```
3.2e     ?- vertical(S), horizontal(S).
```

Question 3.3 The following Prolog program is consulted by the Prolog interpreter.

```
parent(pat,jim).
parent(pam,bob).
parent(bob,ann).
parent(bob,pat).
parent(tom,liz).
parent(tom,bob).
```

What will be the outcome of each of the following queries?

```
3.3a     ?- parent(bob,pat).

3.3b     ?- parent(liz,pat).

3.3c     ?- parent(tom,ben).

3.3d     ?- parent(Pam,Liz).

3.3e     ?- parent(P,C),parent(P,C2).
```

Question 3.4 The following Prolog program is consulted by the Prolog interpreter.

```
colour(b1,red).
colour(b2,blue).
colour(b3,yellow).
shape(b1,square).
shape(b2,circle).
shape(b3,square).
size(b1,small).
size(b2,small).
size(b3,large).
```

What will be the outcome of each of the following queries?

```
3.4a     ?- shape(b3,S).
```

```
3.4b     ?- size(W,small).
```

```
3.4c    ?- colour(R,blue).
```

```
3.4d    ?- shape(Y,square),colour(Y,blue).
```

```
3.4e    ?- size(X,large),colour(X,yellow).
```

```
3.4f    ?- shape(BlockA,square),shape(BlockB,square).
```

```
3.4g    ?- size(b2,S),shape(b2,S).
```

```
3.4h    ?- colour(b1,Shape),size(X,small),shape(Y,circle).
```

Question 3.5 The following Prolog program is consulted by the Prolog interpreter.

```
film(res_dogs,dir(tarantino),stars(keitel,roth),1992).
film(sleepless,dir(ephron),stars(ryan,hanks),1993).
film(bambi,dir(disney),stars(bambi,thumper),1942).
film(jur_park,dir(spielberg),stars(neill,dern),1993).
```

What will be the outcome of each of the following queries?

```
3.5a    ?- film(res_dogs,D,S,1992).
```

```
3.5b    ?- film(F,dir(D),stars(Who,hanks),Y).
```

```
3.5c    ?- film(What,Who,stars(thumper),1942).
```

3.5d Write the query that would answer the question:

"Who directed Jurassic Park (jur_park)?"

and give the outcome of the query.

3.5e Write the query that would answer the question:

"What film did hanks appear in in 1993 and who was the other star?"

and give the outcome of the query.

4 How Prolog Works

4.1 Introduction

This section deals with the basics of running Prolog programs.

4.2 Running, Consulting and Editing Prolog Programs

4.2.1 Starting Prolog

Giving the command `sicstus` on DICE will get you into Prolog:

```
snoopy[18]: sicstus
```

You'll see something like this:

```
SICStus 3.10.1 (x86-linux-glibc2.2): Fri Apr 11 19:15:45 CEST 2003
Licensed to dai.ed.ac.uk
| ?-
```

This tells you that the dialect of Prolog that you are using is SICStus Prolog.

4.2.2 Consulting a File

When you get the `| ?-` prompt, type in the name of the file you want to load into the Prolog database (i.e., the file to be **consulted**):

```
| ?- consult(family).
% consulting c:/program files/sicstus prolog 3.10.1/bin/demo/family.pl...
% consulted c:/program files/sicstus prolog 3.10.1/bin/demo/family.pl in module user,
yes
| ?-
```

or use the following abbreviated form instead (but don't do both):

```
| ?- [family].
```

Both forms can be used to access files in other folders than the one in which you loaded Sicstus by replacing the **filename** with **'foldername/filename'**.

You will now have the contents of the file `family` in Prolog's database where it can be instantly accessed by the interpreter. Subsequent Prolog files can be added to the database by consulting them in the same way and they will not overwrite the existing data unless the new predicates exactly match those previously defined.

4.2.3 Seeing What's in the Database

You can see what facts and rules (i.e., what clauses) Prolog knows about at any point by **listing** the contents of the prolog database:

```
| ?- listing.
```

Instead of listing the whole program, you can just list parts of it by using the predicate `listing` and the name of the predicate to be listed:

```
| ?- listing(parent).

parent(A, B) :-
        father(A, B).
parent(A, B) :-
        mother(A, B).

yes
```

4.2.4 Getting Out of Prolog

Type `halt` to get out of Prolog:

```
| ?- halt.

snoopy[19]:
```

Sicstus can also exited by pressing `Ctrl+C` followed by `e` at any point. Run away processes can be aborted without loosing the contents of the Prolog database by pressing `Ctrl+C` followed by `a`.

4.2.5 Writing and Modifying Programs

We can write new programs or alter existing ones using the editor `emacs`. We can add some new clauses to the file `family`; then if we go back into Prolog, and consult this file, the new clauses will be known by the Prolog interpreter.

In general, when writing prolog programs or altering existing ones, you should do the following:

- Write the clauses out on paper first.
- Get into the editor.
- Add to or change the text of the program.
- Go into Prolog.
- Consult the file containing the program.

- List the contents of the database.
- Test the program.
- Work out any changes you need to make.
- Exit Prolog and start again.

Although it may seem a waste of time to work out the clauses on paper first, if you don't do this you may degenerate into hacking away at the program without having a clear idea of what is really going on.

5 Rules

5.1 Introduction

In previous sections, we looked at a simple Prolog program, and introduced **facts** (or assertions), **questions**, **variables**, **syntax** and **conjunctions**. In this section, we go on to discuss **rules**.

5.2 Rules

5.2.1 Expressing Rules in Prolog

As we saw earlier, from the information that the Prolog interpreter has it can answer the question:

```
| ?- drinks(alan, beer).

yes
```

If you had the same information—that is, that *Alan drinks beer*—and were asked the question *Does Alan like beer?*, you would probably answer *Yes*. Knowing that *Alan drinks beer* allows you to infer that *Alan likes beer* (on the grounds that people don't usually drink things that they don't like). To see whether or not the Prolog interpreter is able to make the same inference—i.e., to see if it is able to reason that *If Alan drinks beer, then he must like it*—the question we must ask is:

```
| ?- likes(alan, beer).
```

The Prolog interpreter would reply:

```
no
```

meaning 'I cannot prove that Alan likes beer'. To allow Prolog to make the same inferences as we do, we have to provide the information about the relationship between *liking* and *drinking* explicitly; i.e., we have to find some way of encoding the rule that would be expressed in English as

> If someone drinks beer then we can infer that that person likes beer.

or

> We can say that someone likes beer if we know can prove that they drink it.

In Prolog this becomes:

```
likes(Person, beer):-                   % a Person likes beer if ...
       drinks(Person, beer).            % ... that Person drinks beer
```

Some things to note here:

- The first predicate-argument structure here is what you are trying to show to be true (to prove), in the same way as you would try to satisfy any other assertion.
- The '`:-`' is read as *if*.
- Whatever comes after the *if* and before the full stop is what has to be satisfied for the whole rule to succeed.

In the present example, there is one subgoal to satisfy. So, to satisfy the goal `likes(Person, beer)`, the subgoal `drinks(Person, beer)` must be satified.

The whole structure is generally referred to as a **rule**. A rule is one type of **clause**, just as a fact or assertion is also a clause; in particular, a rule is a clause with a **head** (the left-hand side of the '`:-`') and a **body** (everything to the right-hand side). There can be any number of subgoals in the body of the clause.

A fact or assertion is a clause with no body.

Restrictions on the form of a rule:

- Only one goal may appear in the head.
- Any number of goals may appear in the body, separated by commas.

So, we can't have:

```
happy(fred), powerful(fred):-
              rich(fred).
```

We can have more than one goal in the body of a rule:

> A man is happy if he is rich and famous.

can be expressed in Prolog as:

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

Here we have three **conjoined subgoals**.

To express:

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

in Prolog, we first rewrite it as

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

Then:

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

So the query "is someone happy" translates as `?- happy(Someone).` and will succeed if any of the thrre rules succeed i.e. if either `healthy(Someone).` or `wealthy(Someone).` or `wise((Someone).` can be proved. We may want to be more specific:

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

In Prolog this becomes:

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

Here we have to check in every rules that Someone is a woman. We could have avoided this by using another rule in additional to the one above:

```
happy_woman(P):-
        woman(P),happy(P).
```

5.2.2 Returning to the 'Drinks' Example

If you want to write some conjunction of goals and use it generally (that is, you want to be able to vary what the variables match to), then it would be more convenient to write a rule for it (with all the conjoined subgoals as its body) than to keep rewriting all the conjoined goals separately each time.

If, for example, you wanted to know about the drinks that two people both like, we could write a rule `bothlike(Person, Other, Drink)` that is satisfied if both `Person` and `Other` like the same `Drink`:

```
bothlike(Person , Other, Drink):-       % Person and Other bothlike Drink if
        likes(Person, Drink),           % .. Person likes Drink and
        likes(Other, Drink).            % .. Other likes Drink
```

We would have to add this to the file in which we keep the rest of the clauses about what people like and drink. We would use an editor to do this, and then go back into Prolog and consult the amended file. The following goal (or query) could then be tried:

```
| ?- bothlike(alan, heather, S).
```

This would match to the head of the rule, with `Person` instantiated to `alan` (we will write this as `alan/Person`), `heather/Other` and `S/Drink`.[5] The first subgoal to satisfy will then be:

```
likes(alan, Drink)
```

which will match with `S/coffee`. The next subgoal to satisfy is then

```
likes(heather, coffee)
```

which succeeds. There are no more subgoals, so `bothlike(alan, heather, S)` succeeds with `S` instantiated to `coffee`.

```
| ?- bothlike(alan, heather, S).

S = coffee ?
```

You may also have noticed that, by this rule, if we asked what two people both like coffee we would also find out that both Alan and Alan like coffee! Whilst this may seem silly (it is not quite what we intended), it is perfectly sensible logically.

```
| ?- bothlike(A, B, coffee).

A = alan
B = alan ?
yes
```

If we want to also say that we don't want to have both people being the same person, then we have to explicitly state that they are not the same person. This means that we have to say that the variables that match to the first two arguments of `bothlike(Person, Other, Drink)` must not be instantiated to the same value; in other words, `Person` must not be the same as `Other`. We add a subgoal to the rule in order to say this:

[5] Note that neither `X` nor `S` have a value at this point, but that they will share the same value as soon as one or the other of them becomes instantiated by matching.

```
bothlike(Person , Other, Drink):-  % Person and Other bothlike Drink if
             likes(Person, Drink), % .. Person likes Drink and
             likes(Other, Drink).  % .. Other likes Drink and
             Per\==Other.          % .. Person and Other are not the same
```

For the remainder of this section, however, we will leave this clause with the possibility of `Person` and `Other` being the same.

5.2.3 Recursion

Suppose we decide that, given two people A and B, A likes B if A and B both like the same drink. So we are saying:

> some person likes some other person if
> the first person likes some drink and
> the other person likes the same drink.

We can express this in Prolog as follows:

```
likes(Person, Other):-                 % Person likes Other if ..
        likes(Person, Drink),          % .. Person likes Drink and
        likes(Other, Drink).           % .. Other likes Drink.
```

This program is **recursive**, that is, it calls itself. Some recursive statements:

- An ancestor is a parent or a parent's ancestor.
- A string of characters is a single character or a single character followed by a string of characters.

An example recursive program:

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

In English:

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

Given the 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).
```

and the goal:

```
| ?- talks_about(X,Y).
```

what do we get? Try this for yourself and see.

Another example: Given the database:

```
1.        has_flu(X):- infected(X).
2.        has_flu(X):- kisses(X,Y), has_flu(Y).

3.        infected(john).
4.        kisses(sue,john).
5.        kisses(john,ann).
6.        kisses(sally,ann).
7.        kisses(fred,sue).
```

the outcome of each of the following queries is given, together with some explanation. Note that the above predicates are numbered for ease of reference.

```
a.     ?- has_flu(john).                 yes

                (rule 1:has_flu(john):-infected(john).
                subgoal infected(john) matches 3.
                all subgoals succeed, rule 1 succeeds)


b.         ?- has_flu(sue).        yes

                (rule 1:has_flu(sue):-infected(sue).
                        fails.
                rule 2: has_flu(sue):- kisses(sue,Y),has_flu(Y).
                subgoal kisses(sue,Y) matches 4, Y=john
                subgoal has_flu(john)
                        rule 1:has_flu(john):-infected(john).
                        subgoal infected(john) matches 3.
                        all subgoals succeed, rule 1 succeeds
                all subgoals succeed, rule 2 succeeds)
```

```
c.        ?-  has_flu(sally).     no

                (rule 1:has_flu(sally):-infected(sally).
                        fails.
                rule 2: has_flu(sally):- kisses(sally,Y),has_flu(Y).
                subgoal kisses(sally,Y) matches 6, Y=ann
                subgoal has_flu(ann)
                        rule 1:has_flu(ann):-infected(ann).
                        fails
                        rule 2:has_flu(ann):-kisses(ann,Y2),has_flu(Y2).
                        subgoal kisses(ann,Y2)
                                fails)


d.        ?- has_flu(fred).       yes

        (rule 1:has_flu(fred):-infected(fred).
                fails.
        rule 2: has_flu(fred):- kisses(fred,Y),has_flu(Y).
        subgoal kisses(fred,Y) matches 7,  Y=sue
        subgoal has_flu(sue)
                rule 1:has_flu(sue):-infected(sue).
                        fails.
                rule 2: has_flu(sue):- kisses(sue,Y1),has_flu(Y1).
                subgoal kisses(sue,Y1) matches 4, Y1=john
                subgoal has_flu(john)
                        rule 1:has_flu(john):-infected(john).
                        subgoal infected(john) matches 3.
                        all subgoals succeed, rule 1 succeeds
                all subgoals succeed, rule 2 succeeds
        all subgoals succeed, rule 2 succeeds)
```

5.2.4 Getting multiple answers

Returning to our previous example, the entire program that results from adding the recursive likes/2 rule is shown in Figure 4; here are some examples of queries and responses given this database:

```
| ?- likes(alan, beer).

yes
| ?- likes(alan, heather).

yes
| ?- likes(alan, alan).

yes
| ?-
```

```
drinks(alan, beer).
drinks(heather, lager).
likes(alan, coffee).
likes(alan, whisky).
likes(heather, gin).
hates(heather, whisky).
likes(heather, coffee).
bothlike(Person, Other, Drink):-
        likes(Person, Drink),
        likes(Other, Drink).

likes(Person, beer):-
        drinks(Person, beer).

likes(Person, Other):-
        likes(Person, Drink),
        likes(Other, Drink).
```

Figure 4: The complete program so far

Consider the query *What are all the things that Alan likes?*. In Prolog we would ask this by saying:

```
| ?- likes(alan, What).
```

and in response we would get:

```
What = coffee ? ;
```

The question mark here is Prolog's way of saying 'what do you want me to do next?'. If we just type return, Prolog will say `yes` and return us to the Prolog prompt. However, if we type a semi-colon before we hit the return key, the Prolog interpreter will look for the next solution and print that:

```
What = whisky ? ;
```

We can keep asking for more answers:

```
What = beer ? ;

What = alan ? ;

What = heather

yes
```

If we decide that this is enough we can stop here.

5.3 Summary

The behaviour we have just seen, where Prolog can provide us with more than one answer to a query, raises some interesting questions:

- How does Prolog get these solutions?
- What order does it get them in?
- What are we really doing (or what is the Prolog interpreter doing) when we ask it to find the next solution?
- Does it matter what order the clauses are in?

These are all questions that will be answered in the next section where we will be looking at **backtracking** in Prolog, and the use of trees to represent what the Prolog interpreter is doing.

5.4 Exercises

Question 5.1 The following Prolog program is consulted by the Prolog interpreter.

```
big(bear).
big(elephant).
small(cat).
brown(bear).
black(cat).
grey(elephant).
dark(Animal):- black(Animal).
dark(Animal):- brown(Animal).
```

What will be the outcome of each of the following queries?

```
5.1a     ?- dark(X), big(X).

5.1b     ?- big(X), grey(Y).

5.1c     ?-  dark(D), small(D).

5.1d     ?- big(Animal), black(Animal).

5.1e     ?- small(P), black(P), dark(P).
```

Question 5.2 The following Prolog program is consulted by the Prolog interpreter.

```
knows(A,B):-
        friends(A, B).
```

```
knows(A,B):-
        friends(A, C),
        knows(C, B).

friends(john, alice).
friends(alice, tom).
friends(sue, john).
friends(sue, clive).
friends(fred, tom).
friends(tom, sue).
```

State whether the following queries succeed or fail. If a query fails, explain why.

5.2a `?- knows(alice, john).`

5.2b `?- knows(clive, sue).`

5.2c `?- knows(alice, fred).`

5.2d `?- knows(sue, john).`

5.5 Practical 1

This practical exercise is to be completed during the week 2 practical session. Questions followed by a '____________' should have the answer entered in the space and then shown to the lab demonstrator. The demonstrator will mark on the register that you have completed the section. All practical exercises should be completed by the end of the module.

5.5.1 Getting Started

1. Copy the file `famtree.pl` to your home directory:

```
snoopy[14] cp /home/infteach/prolog/code/famtree.pl  family.pl
```

   This makes a copy of the file `famtree.pl` in your area and calls it `family.pl`.

5.5.2 Running Prolog

1. Give the command `sicstus` to get into Prolog:

```
snoopy[15] sicstus
```

   You'll see something like the following:

```
snoopy[15] sicstus
SICStus 3  #5: Tue Aug 26 10:14:51 BST 1997
| ?-
```

2. When you get the '`| ?-`' prompt, type in the name of the file to be loaded into the Prolog database (i.e., the file to be `consulted`):

```
| ?- consult(family).
{consulting /hame/helen/family.pl...}
{/hame/helen/family.pl consulted, 10 msec 1552 bytes}


yes
| ?-
```

   or use the abbreviation for this instead (but don't do both):

```
| ?- [family].
```

   You will now have the contents of the `family.pl` file in the Prolog database. The **Prolog interpreter** will know everything that is in this file.

3. Look to see all the facts and rules (i.e., clauses) that Prolog knows about at the moment by typing `listing`:

```
| ?- listing.
```

Don't forget the full stop.

4. Instead of listing the whole program, just list parts of it by using the predicate `listing` and the name of the predicate to be listed:

```
| ?- listing(parent).

parent(A, B) :-
        father(A, B).
parent(A, B) :-
        mother(A, B).

yes
| ?- listing(son).

son(A, B) :-
        parent(B, A),
        male(A).

yes
| ?- listing(daughter).

daughter(A, B) :-
        parent(B, A),
        female(A).

yes
| ?-
```

5. Now try the following **goals**, typing each in response to the '`| ?-`' prompt. Guess what you think will happen. See if it does. If it doesn't quite do as you expect, can you see why?

```
| ?- mother(mary,fred).     _____________________________

| ?- father(tom,sue).       _____________________________

| ?- father(cecil,fred).    _____________________________

| ?- male(jim).             _____________________________

| ?- father(tom,Who).       _____________________________

| ?- mother(Mother,fred).   _____________________________

| ?- mother(X,Y).           _____________________________

| ?- parent(fred,cecil).    _____________________________

| ?- daughter(jane,Who).    _____________________________
```

```
| ?- son(What,How).          _____________________________

| ?- son(cecil,jane).        _____________________________
                                                [answers 1]
```

6. Try seeing if there is more than one match for some of the above goals: by typing ‘;’, you ask Prolog to look for another solution.

```
| ?- mother(mary, S).

S = tom ? ;

S = jane ? ;

S = fred ? ;

no
| ?-
```

‘no’ here means ‘no more solutions’.

Try this with other goals.

7. Think of some goals of your own for Prolog to satisfy. Remember you can use `','` to conjoin goals. By usingasking `| ?- sad(jim),lonely(jim).` you are asking if Jim is both sad **AND** lonely.

How would you ask the following?

- Is Tom the father of Jim and of Sue?
- `| ?-`____________________________________
- Who is the father of Cecil and the son of Mary?
- `| ?-`____________________________________
- Is there anyone who has a son and is themself the son of someone?
- `| ?-`____________________________________
- Do Jane and Fred have the same mother?
- `| ?-`____________________________________ [answers 2]

8. Load family.pl in Emacs by typing `emacs family.pl`.

9. Look at the rules and figure out how they work. Now see if you can write similar rules for sister, brother, aunt and uncle. Write your rules in the spaces below. We will type them into `family.pl` later.

Sister:

____

____

____

____

Brother:

Aunt:

Uncle:

[answers 3]

5.5.3 Getting out of Prolog

Type `halt` to get out of Prolog:

```
| ?- halt.
```

You'll see that you get the unix prompt again:

```
| ?- halt.

snoopy[16]
```

5.5.4 Editing the Program

We can alter the file `family` by using the editor `ue` (Emacs).

We can add some new clauses to the file. Then if we go back into Prolog and consult this file, the new clauses will be known by the Prolog interpreter.

Because we have made the changes in the file itself, we can save them to be used again, and we can also change them easily (for example, if we make a mistake in writing the clauses).

1. Get into the editor by typing the following:

   ```
   snoopy[16] emacs family
   ```

2. Add some more people to the database.

3. We could also add the `grandparent` rule. You can put this anywhere in the file you like.

```
grandparent(Grandparent, Grandchild):-
        parent(Grandparent, Parent),
        parent(Parent, Grandchild).
```

4. Save the file by typing `CONTROL-X, CONTROL-C`, or select save from the emacs menu.

You are now back at Unix command level.

5.5.5 Testing the Program

1. Look at the file to see your latest version.

```
snoopy[17] more family
```

2. Get into Prolog, and consult your new version of `family`.

3. Try out some goals that make use of the new clauses you've added. If you spot any errors, you will need to edit the file again to alter it.

```
snoopy[19] prolog
SICStus 2.1 #9: Mon Jul 11 10:16:09 BST 1994
| ?- consult(family).
{consulting /usr/local/dai/docs/dai/teaching/modules/1-prolog/code/famtree.pl...}
{/usr/local/dai/docs/dai/teaching/modules/1-prolog/code/famtree.pl
consulted, 10 msec 2784 bytes}

yes
| ?- listing.
```

5.5.6 More Things to Add

Try adding brother/sister and aunt/uncle clauses:

- Go back to the versions you wrote in the workbook;
- get into the editor;
- add to or change the text;
- save the file;
- get back into Prolog;
- consult the file;
- list the database;
- test the new clauses;
- look for any changes needed; and
- start again.

5.5.7 Harder Things

If you get this far, and are quite happy, then try something harder.

1. How would you add facts and rules (i.e., clauses) to the database to give information about people's ages, and to be able to answer questions like *Is Fred older than Jim?* or *How old is Sue?*

   Some hints:

   - You could have a clause called `age` with two arguments, one for a person's name and the other for that person's age; for example:

     ```
     age(cecil,30).
     age(mary,72).
     ```

   - You can compare ages using '`>`', which means 'is greater than'. `X > Y` is true (the goal succeeds) if whatever matches to `X` is greater that whatever matches to `Y`.

     So, if `X` becomes instantiated to 72 and `Y` to 30, then the goal `X > Y` would succeed. Try:

     ```
     | ?- age(cecil,X), age(mary,Y), Y>X.
     ```

     (you have to add some ages to the database first).

2. There might be a rule

   ```
   is_older(OldPerson,YoungPerson)
   ```

   which is true when the `OldPerson` is older than the `YoungPerson`.

   The whole rule, if we were comparing Mary and Cecil's ages, would be

   Mary is older than Cecil if
   we know Mary's age and
   we know Cecil's age and
   Mary's age is greater than Cecil's age.

   Try to write this rule in Prolog.

   [answers 4]

3. We could have different rules for cases where we don't know everyone's age, but we know that if Fred is the parent of Cecil then Fred must be older than Cecil. See if you can write the corresponding Prolog rule.

   [answers 5]

4. We might try to write a rule that looks like this:

    Mary is older than Cecil if
        Mary is older than someone and
        that someone is older than Cecil.

[answers 6]

We are getting into more hairy ground here with clauses that have subgoals with the same name as themselves. If a clause calls a clause of the same name (i.e., if it has itself as a subgoal), we say that it is **recursive**.

If you repeatedly ask for multiple answers using ; you may find that eventaully Prolog loops i.e. it is continually asking the same questions and not finding a solution. This is the main problem with recursive clauses and will be dealt with in the next tutorial.

6 Backtracking

This section deals with Prolog's search strategy, explaining how the Prolog interpreter backtracks to get more than one solution to a problem. The idea of using trees to represent the way that the Prolog interpreter searches for solutions is also introduced.

6.1 Getting more than one answer

The version of the `drinks` program that we will use below is shown in Figure 5. The clauses are numbered here to make it easier to refer to them later. Note that we are using a more general version of clause 9 here than that introduced earlier: we have substituted a variable for the occurences of `beer` in the corresponding rule.

Given this database, the question *What are all the things that Alan likes?* will cause the following answers to be generated:

```
| ?- likes(alan, What).

What = coffee ? ;

What = whisky ? ;

What = beer ? ;

What = alan ? ;

What = heather ? ;
```

After this solution we get no more; instead, the program seems to stop working, and we hear nothing back from Prolog. In some implementations of Prolog, we will be presented with an error message like the following:

```
%%% Local stack overflow - forced to abort %%%
```

In SICStus Prolog, the program doesn't come back at all. To restore things to normal, you have to type a Control-C character (hold down the key marked `CONTROL` or `CTRL`, and press the '`c`' key while the `CONTROL` key is depressed).

Prolog will then respond with

```
^C
Prolog interruption (h for help)?
```

At this point, you should type `a` to abort the program. Prolog will respond with

```
{ Execution aborted }
| ?-
```

Why this problem occurs will become clear below; for the moment, we will focus on how the Prolog interpreter gets all these solutions to the question.

```
1.  drinks(alan, beer).
2.  drinks(heather, lager).
3.  likes(alan, coffee).
4.  likes(alan, whisky).
5.  likes(heather, gin).
6.  hates(heather, whisky).
7.  likes(heather, coffee).
8.  bothlike(Person, Other, X):-
           likes(Person, X),
           likes(Other, X).
9.  likes(Person, Drink):-
           drinks(Person, Drink).
10. likes(Person, Other):-
           likes(Person, Drink),
           likes(Other, Drink).
```

Figure 5: The complete `drinks` program

6.2 Representing Prolog's behaviour using trees

The first solution to the query used above comes from the first match of `likes(alan, What)` with `likes(alan, coffee)`, so we have the instantiation `What/coffee`.

If we take all the clauses in the database in order, this is the first clause to match (clause 3 in the table above). We can represent this as a tree with the root node being the top level goal (`likes(alan, What)`) and its daughter nodes being either:

- subgoals that must be satisfied for this top level goal to succeed;
- an indication of success (shown as $\bigcirc$) where this goal can succeed directly; or
- an indication of failure (shown as $\otimes$) where there is no match that can made, either directly or if other subgoals are satisfied.

The arcs of the tree will be labelled with the number of the matching clause, and alongside will be written any instantiations that are made in the process of matching. Our first solution is thus as shown in Figure 6.

If we want to find other solutions, what we are doing (in effect) is failing the match that we have just found and saying 'Look for the next one'. Whenever a match is made, the clause that matches is noted and any instantiations are also noted by the interpreter. If we say 'Go back and redo that match' (by typing '`;`') then the search for a solution continues from where it left off. All instantiations made by this last match (that has now been made to fail) are forgotten. So here, the Prolog interpreter 'forgets' `What/coffee`, then looks for the next clause (after clause 3) that will match.

This will be clause number 4, with `What/whisky` (`What` instantiated to `whisky`); this is shown in Figure 7.

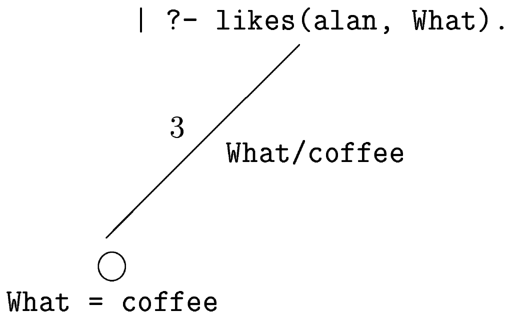


Figure 6: The first solution

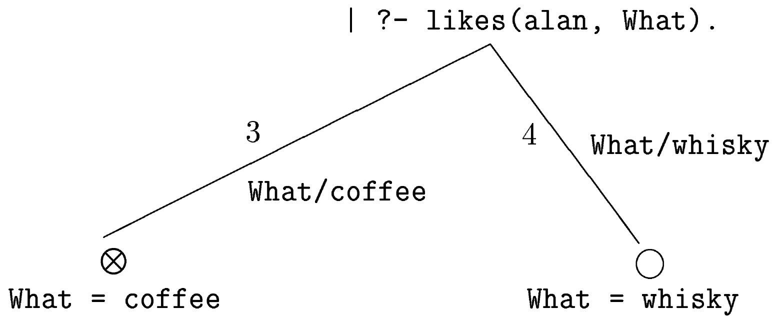


Figure 7: The second solution

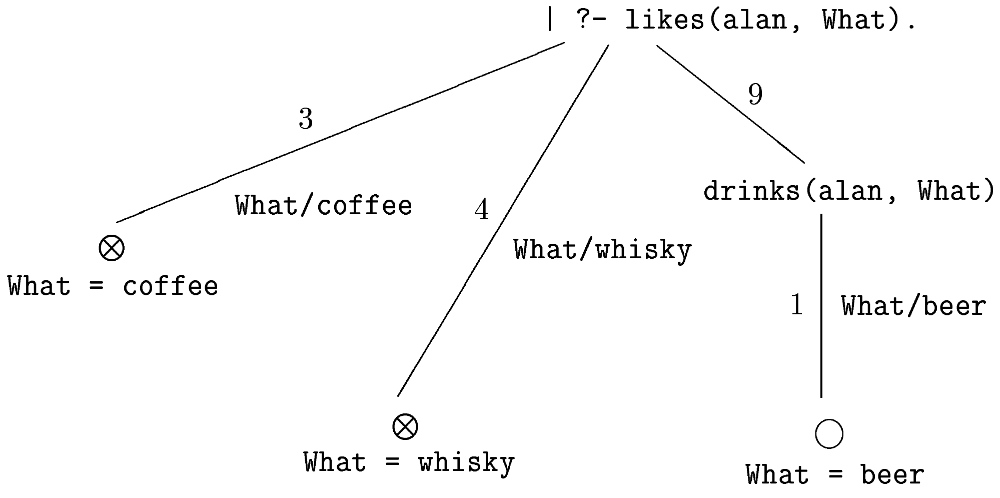


Figure 8: The third solution

With game playing programs, we can draw a tree to show all the possible or potential paths to a solution (all the legal moves); here, we use a slightly different kind of tree here to show the solutions to the query.

After the match with clause 4, there are no more simple assertions that match. The next match is with clause 9: in this clause, `alan` matches `Person` and `What` matches `Drink`. This is a rule, however, so now we have to prove the subgoal in the right-hand side of the rule.

We represent the subgoal explicitly in the tree, and look for matches for it. This subgoal is treated as if it is a completely new goal, so a match is attempted for the subgoal `drinks(alan, What)`.

Prolog will look through the clauses in the database from the beginning until a match is found; in the present case, we get a match with clause 1. So, the subgoal `drinks(alan, What)` succeeds with `What/beer`. Since all of its subgoals have been satisfied, `likes(alan, What)` also succeeds. The corresponding tree is shown in Figure 8.

On redoing after this match, the interpreter will look for another match for `drinks(alan, What)` first, forgetting the match of `What` to `beer`.

There is no other match for this goal, so Prolog will go back further up the tree and see if it can redo the previous goal. This means looking for another match for `likes(alan, What)` after clause 9.

There is a match in clause 10. However, to satisfy clause 10, two new subgoals must be satisfied first. These are `likes(alan, Drink)` and `likes(What, Drink)`.

Note here that because `What` in the goal matches to `Other` in clause 10, both variables will share the same value. The variable called `Drink` in the rule is a new one, so the Prolog interpreter generates some new name for it. This will eventually get a value on matching, but will not be passed back to the top level goal (simply because it doesn't appear in it).

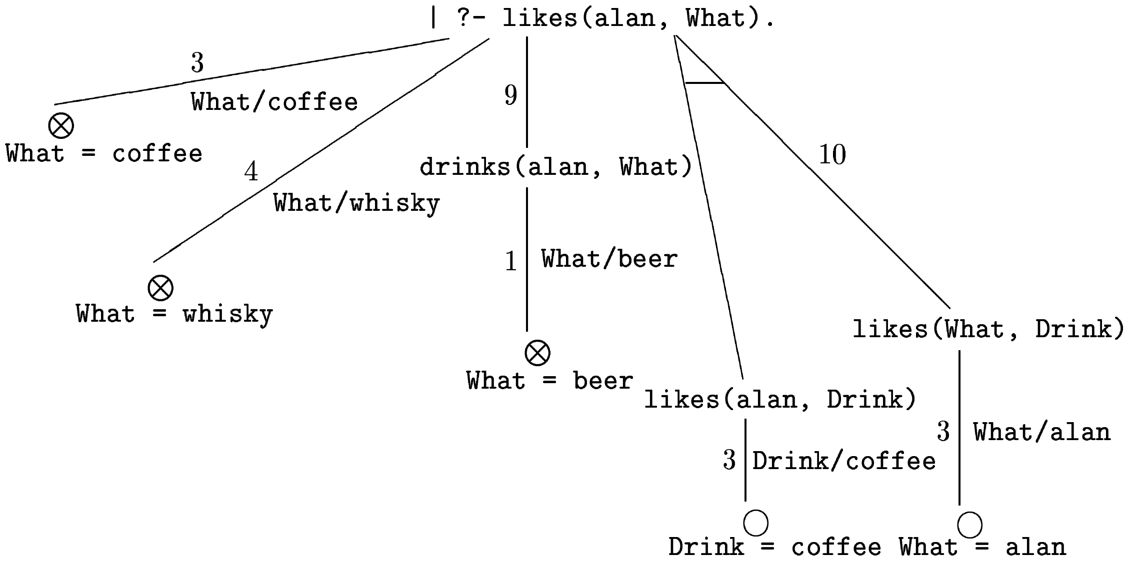


Figure 9: The fourth solution

Because there are now conjoined subgoals to match (`likes(alan, Drink)` and `likes(What, Drink)`), these are both drawn in the tree and joined together to indicate that they both have to be satisfied before the goal above them (their parent) can be satisfied; see Figure 9.

So, the first subgoal to be matched from clause 10 is `likes(alan, Drink)`, which succeeds with `D/coffee` (as the first call of `likes(alan, What)`). The second subgoal now becomes `likes(What, coffee)`. The first match to this new subgoal will be `What/alan`. So `likes(What,coffee)` succeeds with `What/alan`.

It may not make to much sense in English to say *Alan likes himself if he and himself both like coffee*, but logically it is fine.

If we redo once more, we go back to the last goal that succeeded, and redo it (that is, we fail the last goal and look for the next match). In the present case, this means we forget `What/alan` and look for another solution to `likes(What, coffee)`.

This will match to clause 7 with `What/heather`, as shown in Figure 10.

Things get messier from here on. The last goal to succeed was `likes(What,coffee)` with `What/heather` (from clause 7); we try and redo this. The next clause that may give a solution to `likes(What,coffee)` is clause 9, which could be paraphrased here as *What likes coffee if What drinks coffee*. So, `likes(What, coffee)` succeeds if subgoal `drinks(What, coffee)` succeeds. This fails altogether and we redo again.

Next we try the match with clause 10, meaning that *What likes coffee if What likes SomeD and coffee likes SomeD*.[6] We get a first match to this with `What/alan` and `Somedrink/coffee`; so the second subgoal `likes(D, SomeD)` is now instantiated to `likes(coffee, coffee)`.

[6] We will introduce new variables names from here on to make it easier to see what is happening.

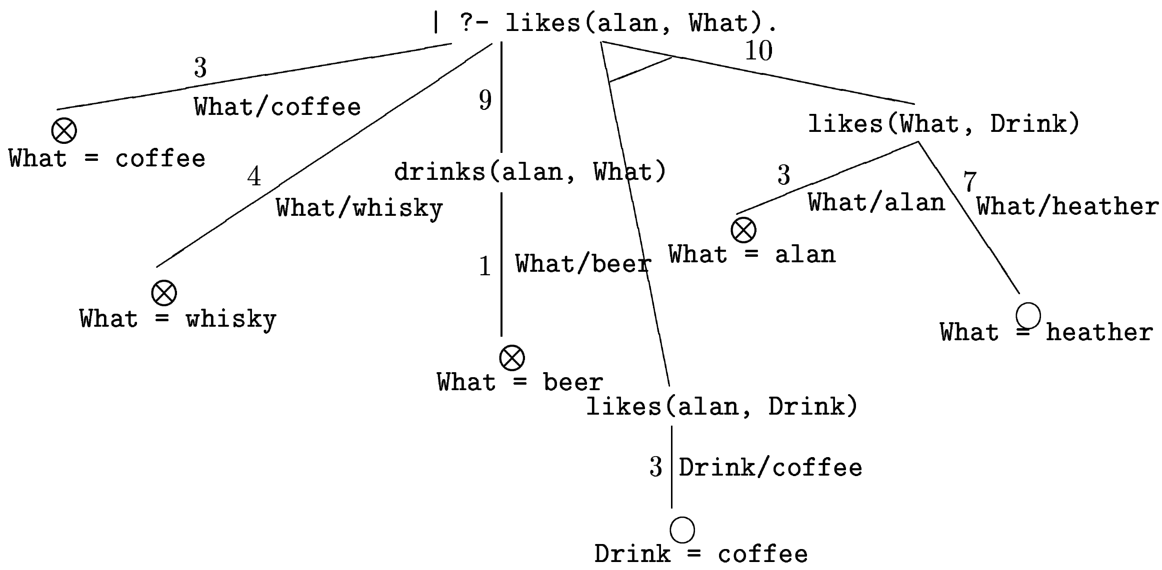


Figure 10: The fifth solution

Clauses 1 to 8 fail, and clause 9 is tried: `likes(coffee, coffee)` if `drinks(coffee, coffee)`. This fails, so clause 10 is tried again.

`likes(coffee, coffee)` if `likes(coffee, D2)` and `likes(coffee, D2)`

This means we try again on `likes(coffee, D2)`; again, clauses 1–9 all fail, and so we try clause 10:

`likes(coffee, D2)` if `likes(coffee, D3)` and `likes(D2, D3)`

Our new subgoal is `likes(coffee, D3)`, which again fails to match clauses 1–9, and so once more we have clause 10:

`likes(coffee, D3)` if `likes(coffee, D4)` and `likes(D3, D4)`

which creates a new subgoal ...

As you may have realised by now, we are never going to get a solution to this, and the program will keep on trying to find new subgoals to match for clause 10, which each in turn call clause 10, and so on. We have an **infinite loop** here.

6.3 Summary

You should have some idea now about how different solutions to a goal are achieved, and how we can use tree to represent 'getting all the solutions'. We will call this particular kind of tree an augmented AND/OR tree, referred to hereafter as an AND/OR tree.

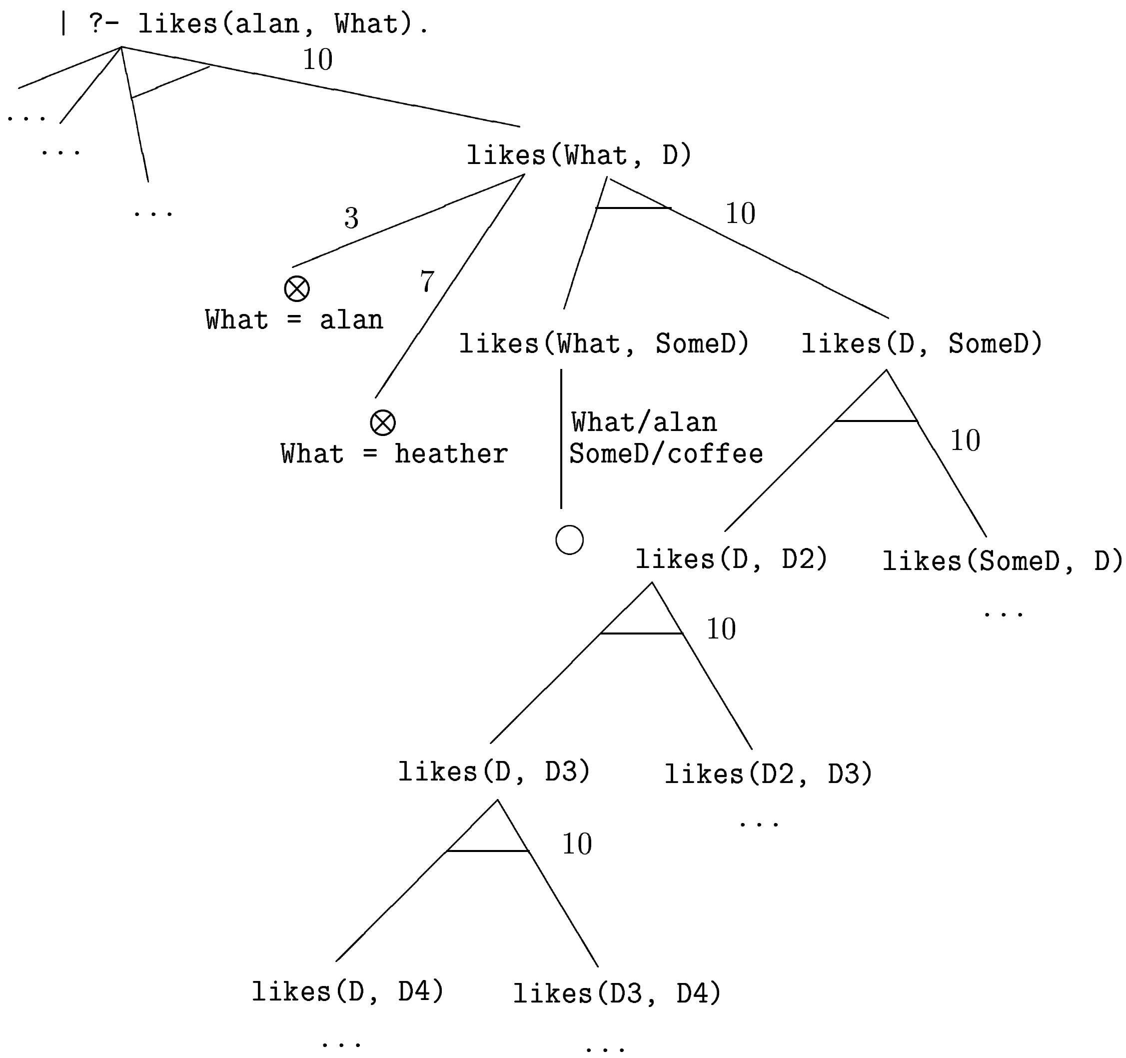


Figure 11: The infinite loop

In the next section, we will use a different example that includes a simple database and clauses to help us answer questions about the contents of the database. Over the next few sections, we'll extend the program in various ways to make it easier to use.

6.4 Exercises

For each of the following programs, say if the query given fails or succeeds. Give any bindings made as a consequence.

Question 6.1

```
a:-b,c.
b.
c:-d.
d:-e.

?- a.
```

Question 6.2

```
a:-b,c.
c:-e.
b:-f,g.
b:-n.
e.
f.
n.

?- a.
```

Question 6.3

```
do(X):-a(X),b(X).
   a(X):-c(X),d(X).
   a(X):-e(X).
   b(X):-f(X).
   b(X):-c(X).
   b(X):-d(X).
   c(1).
   c(3).
   d(3).
   d(2).
   e(2).
   f(1).
```

```
6.3a        ?- do(1).
```

```
6.3b        ?- do(2).

6.3c        ?- do(3).

6.3d        ?- do(A).
```

7 Built-in System Predicates

7.1 Introduction

In this section, we introduce a new example. This is a simple program that represents knowledge about books, their publishers and the shops that stock these publishers. Over the next sections, this program will be extended in a number of ways to make it easier to use. In order to do so, we'll make use of **system predicates** or **built-in predicates**.

7.2 The Example Program

The program shown in Figure 12 will be referred to as `books1`, and will be the simplest version of the program that we will use.

The program contains a number of different predicates:[7]

- The predicate `stocks/2` has two arguments: the first represents the name of a bookshop, and the second the name of a publisher stocked by that shop.
- The first argument of `book/2` is a book title, and the second is the book's publisher.
- The `open/1` and `closed/1` predicates indicate whether particular shops are open or closed.

7.3 Asking the Database Some Questions

We can ask Prolog questions about this database. The sort of questions we might want to ask, in English, are things like:

- Which shop stocks Virago?
- Who publishes *ET Rides Again*?
- Where can I buy *I Claudius*?

In Prolog, the first two of these questions would be:

```
| ?- stocks(Shop, virago).

| ?- book(et_rides_again, Publisher).
```

The third question is a little more complicated. To answer *Where can I buy I Claudius?*, we need to answer a number of questions as follows:

[7]Note: a predicate is a collection of clauses with the same predicate name and the same number of arguments; the number of arguments of a predicate is the **arity** of the predicate.

```
stocks(james_thin, sfipubs).
stocks(james_thin, virago).
stocks(james_thin, penguin).
stocks(menzies, sfipubs).
stocks(menzies, sams).
stocks(better_books, penguin).
stocks(better_books, virago).
stocks(edinbooks, sfipubs).
stocks(edinbooks, virago).
stocks(edinbooks, sams).

book(son_of_et, sfipubs).
book(et, sfipubs).
book(i_was_a_teenage_robot, sfipubs).
book(et_rides_again, sfipubs).
book(biggles_and_wendy, virago).
book(freda_the_fire_engine, virago).
book(dict_of_computing, penguin).
book(i_claudius, penguin).
book(of_mice_and_men, penguin).
book(cookbook, sams).

open(menzies).
open(better_books).
open(edinbooks).
closed(james_thin).
```

Figure 12: The `books1` program

- Who publishes *I Claudius*?
- What shop stocks this publisher?
- Is this shop open?

To do this in Prolog, we need a conjunction of goals:

```
| ?- book(i_claudius, Publisher), stocks(Shop, Publisher), open(Shop).
```

and in this case the answer would be:

```
Publisher = penguin,
Shop = better_books ?
```

Note that the first instantiation of `Shop` would be `james_thin`, but this would fail in the attempt to match the third goal; the Prolog interpreter would then backtrack and redo the second goal, instantiating `Shop` to `better_books`, after which `open(better_books)` succeeds.

7.4 Using Rules

If questions about where books can be purchased are to be asked often, it is sensible to write a rule to save repetition:

```
canbuy(Book, Shop):-
          book(Book, Publisher),
          stocks(Shop, Publisher),
          open(Shop).
```

We can now ask simply:

```
| ?- canbuy(i_claudius,Shop).

Shop = better_books ?

yes
| ?-
```

7.5 Adding Built-in or System Predicates for User Interaction

We can now go on to make the program more interesting and interactive by using a number of predicates provided by the system. These come for free and do not have to be defined by the user. They are called **system predicates** or **built-in predicates**. One example of a system predicate that we have already encountered is the predicate `listing/0`.

```
| ?- go.
What book would you like to buy?
|: et.
You can buy et at menzies.
Would you like another book?
|: yes.
What book would you like to buy?
|: noddy_and_big_ears.
I don't know where you can buy that book, sorry.
Would you like another book?
|: yes.
What book would you like to buy?
|: biggles_and_wendy.
You can buy biggles_and_wendy at better_books.
Would you like another book?
|: no.
I hope I was of some help to you. Have a nice day.
yes | ?-
```

Figure 13: An interactive session with the `books` program

```
| ?- listing.
```

This shows all clauses currently in the Prolog database. `listing/1`, on the other hand, takes a predicate name as argument and lists all the clauses with that predicate name: so, for example

```
| ?- listing(book).
```

shows all the clauses that have the predicate name `book`.

The Prolog interpreter prints out the values of all variables appearing in the top level goal, but does not print the values of any other variables which are instantiated during the process of satisfying the top level goal. The printing of variables and their values is really just a side-effect of the matching process; it would be better if we had explicit control of the printing of variable values. The system predicate `write/1` gives us this control. There is also a system predicate `read/1` which allows a value to be given to a variable by being typed in by a user. Both these predicates take a single argument that can be instantiated to any term: that is, a variable, a constant, a number or any other atom, or a clause. **Atoms** include characters enclosed in single quotes, such as 'What book would you like?; this permits strings of characters commencing with capital letters and containing spaces to be written out.

So if we wanted another user to use the `books` program, we could make it interact with the user, having the program ask questions and read the user's responses. The dialogue might go something like that shown in Figure 13 (program output is printed here in *italics*).

To carry out this dialogue, we need to augment the program by adding a predicate that:

- asks the user what book they would like;

- reads the reply;
- finds where the book can be bought; and
- tells the user.

We might call this predicate `askbook`, and it might look something like the following:

```
askbook:-
        write('What book would you like to buy?'), nl,
        read(Book), nl,
        canbuy(Book, Shop),
        write('You can buy '),
        write(Book),
        write(' at '),
        write(Shop),
        write('.').
```

The system predicates used here are `write/1`, `read/1`, and `nl/0`. `nl/0` has the effect of printing a new line.

When the `read/1` predicate is called, the interpreter prints a new prompt, '`|:`', and waits for the user to type a term terminated by a full stop. The value of the variable argument of `read/1` becomes instantiated to the term the user types. This provides a way to input values to a program.

If the book whose name was input does not exist or cannot be bought—that is, if the `canbuy/2` predicate fails—the Prolog interpreter will attempt to backtrack through `nl`, `read`, and `write`. These predicates cannot be re-satisfied (or redone), so an attempt will then be made to redo the predicate `askbook/2`. If there were no other `askbooks/2` clause, the whole program would fail here. However, we can add a second clause that simply writes a message to the user (telling them that the book is unknown) and succeeds:

```
askbook:-
        write('I don''t know where you can buy that book, sorry.').
```

Note the use of the double single quote here to allow us to print a single quote as part of a string that is itself delimited by single quotes.

7.6 Looking for Multiple Responses

We could extend the program further, allowing the user to ask about more books. This can be done by putting `askbook/0` as a subgoal to another `clause go/0`; the predicate `go/0` then becomes the top level predicate, and will do the following:

- call `askbook/0`, asking the user which book they want and responding accordingly;
- offer to find another book;
- check the user's reply;

```
go:-
        askbook, nl,
        write('Would you like another book?'), nl,
        read(Reply), nl,
        check(Reply), nl.

check(Reply):-
        Reply = yes,
        go.

check(Reply):-
        write('I hope I was of some help to you.  Have a nice day.').
```

Figure 14: The Prolog code to allow repeated requests

- if the user types `yes`, then `go/0` will be called again;
- if anything else is typed in response, the program stops.

The corresponding Prolog code looks like that shown in Figure 14.

7.7 Dealing with Arithmetic Operators

Arithmetic operators are another type of system predicate. The operators then enable us to do arithmetic in Prolog are:

+ - * / add minus times divide

There is also a system predicate that the results of applying these operators, called `is/2`. All of the arithmetic operators can be used as infix (between arguments) or as prefix (like a predicate), for example:

```
e.g.

?- X is 3+7.

     X=10

?- B is +(2,99).

     B=101

?- B is 8 + 3.

     B=11
```

Note that you need to put spaces either side of `is/2`:

```
Dis7+2.
no

?- 3 is 2 + 1.
yes

?- 4 is 4.
yes

?- 4 = 4.
yes
```

Note also that `=` and `is` mean different things: `=` means `will unify with` whereas `is/2` means `evaluates to`.

```
?- 4 = 3 + 1.
no

?- 4 is 3 + 1.
yes
```

Prolog must be able to evaluate the right hand side of `is/2`: If there are variables on the right hand side that are uninstantiated then it will fail.

```
?- S is H+2.

*** Error: uninstantiated variable in arithmetic expression: _68
no

?- X is 3+3.

    X=6
yes

| ?- S is X+3.

*** Error: uninstantiated variable in arithmetic expression: _68
no

?- X is 3+3,Y is X+2.

     X=6
     Y=8
yes
```

Expressions can be complex, in the same way as ordinary arithmetic expressions.

```
| ?- Sum is 4+3-6+2.
     Sum=3
yes

| ?- Sum is 3*4.
     Sum=12
yes

| ?- Ans is 3*4-5.
     Ans=7
yes
```

The left hand side is never evaluated:

```
| ?- 3+4 is 7.
no

| ?- 3+4 is 3+4.
no

| ?- P is 5/8.
     P=0.625
yes
```

Comparisons of the sort `</2`(less than), `>/2`(greater than), `>=/2`(greater than or equal to), and `=</2`(less than or equal to) can also be made:

```
| ?- 4>3.
yes

| ?- 3>4.
no

| ?- 5<10.
yes

| ?- 5<3+8.
yes

| ?- 5*6>9+10.
yes

| ?- 5>=5.
yes
```

```
| ?- 8=<16.
yes

| ?- 3-1+6*4-2<7*8+8-10.
yes
```

Note that system predicates cannot be traced [8]:

```
| ?- trace,3-1+6*4<7*8+8-10.
yes
```

Prefix notation can also be used instead of the more common infix notation:

```
| ?- <(+(-(3,1),*(6,4)),+(*(7,8),-(8,10))).
yes

| ?- T is +(-(3,1),*(6,4)).
     T=26
yes

| ?- P is +(*(7,8),-(8,10)).
     P=54
yes

| ?- 26<54.
yes
```

We have so far represented two types of equality: `=/2` tests whether two terms can unify, and `is/2` checks if the value of an arithmetic expression matches the value of a term. But these arn't the only types. We can check if the value of two arithmetic expressions are the same by infixing them with `=:=` or not equal using `=\=`.

```
| ?- 2+3 =\= 3+3.
yes
| ?- 2+3 =\= 3+2.
no
| ?- 2+3 =:= 3+2.
yes
| ?- X=2,Y=3,X+Y =:= Y+X.
X = 2,
Y = 3 ?
yes
| ?- X=2,Y=3,X+Y =\= Y+X.
no
```

[8] see later section on Tracing and Debugging

We can also check if two terms have *literal equality* using ==/2. This checks if the two terms are identical; that is, they have exactly the same structure and all the corresponding components are the same. In particular, the variable names also have to be the same. The complementary relation is 'not identical' written \==.

```
| ?- X == X.
true ?
yes
| ?- X == Y.
no
| ?- 2+3 == 2+3.
yes
| ?- 2+3 == 3+2.
no
| ?- X \== Y.
true ?
yes
```

7.8 Summary

Whilst this extended program (which will be called books2) is an improvement on the basic program we started out with, it is still a little limiting in a number of respects:

1. There is little flexibility in the set of responses that can be given to the question *Would you like another book?*. In particular, note that responses other than yes have the following effects:
   - |: y. fails to match.
   - |: Yes. is a variable, and so becomes bound to yes and succeeds.
   - |: No. also becomes bound to yes and succeeds.
2. In each case, the canbuy/2 predicate only finds the first solution, rather than offering all possible solutions to the query.

In the next section we consider how these limitations can be avoided by introducing data structures called **lists** that can be used to represent a collection of objects (for example, all the possible shops that a particular book can be bought from).

7.9 Practical 2: Constructing Databases

7.9.1 Part 1: Writing a Program to Find Who is Where

This practical is about writing a program to incorporate information about the location of people and their phone numbers. Your program should be able to answer questions like *Where is Fred?* and *What is Fred's phone number?* (given in appropriate Prolog terms, of course). Your program will also know about people visiting others and how to contact them.

1. Write predicates to incorporate the following information:

   Helen has room F5.
   Frank has room E6.
   Paul has room F9.
   Han has room E6.
   Robert has room F9.
   Dave has room E10.
   Henry has room E12.
   Janet has room E11.
   Graeme has room E13.

   For each fact, use a predicate `room` with two arguments: the first argument will be the person, and the second the room number.

   Wite a couple of example facts here:

   ---

2. Use MicroEmacs (`ue`) to put these clauses into a file on your area. Check that there are no errors. Then go into Prolog and consult this file.

3. Test the program by asking questions such as (write the prolog version in the spaces):

   Which is Dave's room? ________________________________
   Who has room F9? _____________________________________

4. Add some more clauses to store the telephone number for each room. Use a predicate `phone` with two arguments (hereafter referred to as `phone/2`): the first argument should be the room number and the second should be the extension number. The extensions are as follows:

```
Room       Tel No.
E10         231
E12         233
E11         244
E13         237
F5          242
```

```
F9          239
E6          247
```

Wite a couple of example facts here:

____

____

____

5. Again, use the editor to edit your original file, then go back into Prolog and consult the file and test it. Ask questions like:

   Which room has extension 239? ________________________________
   What is room E6's extension? ________________________________

6. Add a rule that will allow you to find out a person's phone number, if you know their room number and the phone number for that room. Use a predicate `ring/2`, whose first argument should be the person and second argument their phone number. So, a declarative reading for the rule you have to write might be something like

   P can be rung at number N if P is in room R and room R has number N.

   Wite your Prolog code here:

   ____

   ____

   ____

   ____

7. Again, edit the file, consult it and test it.

7.9.2 Complicating the Program

Suppose there are other people we know about, but instead of knowing their room numbers, we know whether or not they are visiting the office of someone else we know about. In this section, we'll incorporate this kind of information into the program.

1. To find out where someone is, first you would check to see if you have their room number, and if not, you would then check to see if they are visiting someone else and you have that person's room number.

   Add the following information, using a predicate `visiting/2`:

   Alan is visiting Helen. ______________________________
   Liam is visiting Paul. ______________________________
   Jane is visiting Alan. ______________________________

   Also, add a rule that tells you how to find people: you can find person P if they are in their room, or (if that fails) you can find person P if they are visiting person Q and you can find person Q.

Use a predicate `find/2`, the first argument being the person you want to find and the second being the room that they are found in (even if it is the room of the person they are visiting).

Wite your Prolog code here:

____

____

____

____

____

____

2. Edit the program, run it and test it by asking the following questions and some others of your own:

   Who is Alan visiting? [Answer]______________________
   Where can you find Dave?______________________
   Where can you find Jane?______________________
   Who is in room F9?______________________

3. If you alter the predicate `ring`, so that it has the subgoal of `find(Person,Room)` instead of `room(Person,Room)`, you should be able to get the phone number you need to know in order to contact any person, even if they are visiting somebody else (or even if they are visiting someone who is already visiting someone!).

   Do this and test it: find everyone's phone numbers.

7.9.3 Part 2: The Basic `books` Program

We start by using the first version of the `books` program as discussed earlier in this chapter. More complicated versions of this program will be used later.

1. Copy the file that contains the `books` program from the `/home/infteach/prolog/code` area.

```
cp /home/infteach/prolog/code/books1.pl books1
```

2. Start Prolog and consult the file:

```
consult(books1).
```

3. List the contents of the database and see if you can work out what sort of questions you could ask. A complete listing is provided in Figure 15.

4. Work out how to ask the following questions:

   (a) Which publisher publishes *ET*?

```
stocks(james_thin,sfipubs).
stocks(james_thin,virago).
stocks(james_thin,penguin).
stocks(menzies,sfipubs).
stocks(menzies,sams).
stocks(better_books,penguin).
stocks(better_books,virago).
stocks(edinbooks,sfipubs).
stocks(edinbooks,virago).
stocks(edinbooks,sams).

book(son_of_et,sfipubs).
book(et,sfipubs).
book(i_was_a_teenage_robot,sfipubs).
book(et_rides_again,sfipubs).
book(biggles_and_wendy,virago).
book(freda_the_fire_engine,virago).
book(dict_of_computing,penguin).
book(i_claudius,penguin).
book(of_mice_and_men,penguin).
book(cookbook,sams).

open(menzies).
open(better_books).
open(edinbooks).
closed(james_thin).

canbuy(Book, Shop):-
    book(Book, Publisher),
    stocks(Shop, Publisher),
    open(Shop).
```

Figure 15: The books program, first version

(b) [Question]____________________ [Answer]__________________
(c) Which shop stocks Penguin books?
(d) ______________________________ __________________________
(e) Where can you buy *I Claudius*?
(f) ______________________________ __________________________
(g) Is there any publisher who publishes both *Freda the Fire Engine* and *Biggles and Wendy*?
(h) ______________________________ __________________________
(i) Are there two shops who both stock *Of Mice and Men* and *Biggles and Wendy*?
(j) ______________________________ __________________________

5. Try a few questions of your own. Test the program fully to see exactly how it works, and where it fails.

7.9.4 The `books` Program—Simple Interactive Version

Once you are quite happy and sure of what is going on with the first version of the `books` program, you should explore the next version.

1. Get out of Prolog. Copy the file that contains the interactive version:

   ```
   cp /home/infteach/prolog/code/books2.pl books2
   ```

2. Get back into Prolog again and then consult your new file.

   This is a more complicated program; a partial listing is provided in Figure 16. Try and see what it does and how it works by asking some questions.

3. Try adding a new book to the database (try `book(bratko,addison_wesley)`). What happens when you try to buy this book? Does the systems response differ from when it doesn't recognise the name of the book?

4. Think of the different ways in which your request can fail and try to add extra clauses to the database to generate specific responses in this instances. Write your additions below.

7.9.5 Writing your own Database

Once you have made sure you know exactly what is going on in the `books2` program (and you might want to play a little with the `read` and `write` predicates to check this), then write a database of your own.

```
stocks(james_thin,sfipubs).
stocks(james_thin,virago).
...
stocks(edinbooks,sams).

book(son_of_et,sfipubs).
book(et,sfipubs).
...
book(cookbook,sams).

open(menzies).
open(better_books).
open(edinbooks).
closed(james_thin).

canbuy(Book,Shop):-
      book(Book,Pub),
      stocks(Shop,Pub),
      open(Shop).

go:-
      askbook, nl,
      write('Would you like another book?'), nl,
      read(Reply), nl,
      check(Reply), nl.

askbook:-
      write('What book would you like to buy?'), nl,
      read(Book), nl,
      canbuy(Book, Shop),
      write('You can buy '),
      write(Book),
      write(' at '),
      write(Shop),
      write('.').

askbook:-
      write('I don''t know where you can buy that book, sorry.').

check(Reply):-
      Reply = yes,
      go.

check(Reply):-
      write('I hope I was of some help to you.  Have a nice day.').
```

Figure 16: The second version of the books program

Extend it in the same way the `books` program was extended, so that someone else can use it.

For example, you could put your timetable in it, and allow the user to query when you are free. You might want to include information about time and place of lectures, tutorials and practicals.

Test it on your neighbour. Explain how it works and then let them try it.

[This section is optional but is good practice if you have time left during the practical session]

8 Lists

8.1 Introduction

In this section, we introduce a third version of the `books` program. This version will collect together all solutions to the `canbuy(Book, Shop)` goal and then print out the list of open shops.

In order to understand how this program works, this section is largely devoted to presenting the **list** data structure.

8.2 The `books3` Program

8.2.1 The Example Database

The basic database that the `books3` program uses is as before, and repeated in Figure 17.

8.2.2 The Rules

This version of the program uses the same predicates as before, as shown in Figure 18. Note, however, that the `askbook/0` and `canbuy/2` clauses do more in this version of the program: we have added subgoals that use the new predicates `prlist/1`, `possible/2`, and `filter/2`, and changed the behaviour of `canbuy/2` and `check/1` a little.[9]

8.2.3 How It Works

The basic structure of the `books3` program is as follows:

- ask what book the user wants;
- see where it can be bought:
  - find out its publisher;
  - see which shops stock this publisher (make a list of these);
  - make a new list of the open shops that stock that publisher; and
  - print out the list.
- ask the user if she would like another book:
  - if the reply is positive, do the whole thing again;
  - otherwise, stop.

[9] The `possible/2` and `filter/2` predicates will not be discussed here.

```
stocks(james_thin, sfipubs).
stocks(james_thin, virago).
stocks(james_thin, penguin).
stocks(menzies, sfipubs).
stocks(menzies, sams).
stocks(better_books, penguin).
stocks(better_books, virago).
stocks(edinbooks, sfipubs).
stocks(edinbooks, virago).
stocks(edinbooks, sams).

book(son_of_et, sfipubs).
book(et, sfipubs).
book(i_was_a_teenage_robot, sfipubs).
book(et_rides_again, sfipubs).
book(biggles_and_wendy, virago).
book(freda_the_fire_engine, virago).
book(dict_of_computing, penguin).
book(i_claudius, penguin).
book(of_mice_and_men, penguin).
book(cookbook, sams).

open(menzies).
open(better_books).
open(edinbooks).
closed(james_thin).
```

Figure 17: The `books3` database

```
go:-
        askbook, nl,
        write('Would you like another book?'), nl,
        read(Reply), nl,
        check(Reply), nl.

askbook:-
        write('What book would you like to buy?'), nl,
        read(Book), nl,
        canbuy(Book, Shops),
        write('You can buy '),
        write(Book),
        write(' at '),
        prlist(Shops),
        write('.').

askbook:-
        write('I don''t know where you can buy that book, sorry.').

canbuy(Book, OpenShops):-
        book(Book, Publisher),
        possible(Publisher, ListOfShops),
        filter(ListOfShops, OpenShops).

check(Reply):-
        member(Reply,['Yes',yes,'Y',y]),
        go.

check(Reply):-
        write('I hope I was of some help to you.  Have a nice day.').
```

Figure 18: The rules used in the `books3`

8.2.4 The `member` Predicate

In our new rules, notice that `check/1` includes the goal

```
member(Reply, ['Yes',yes,'Y',y])
```

The `member/2` predicate is used here to check whether the `Reply` is one of a list of possibilities. More generally, `member/2` tests whether an element is a member of a list; the Prolog code for `member` is as follows:

```
member(X, [X|_]).

member(X, [_|Rest]):-
        member(X,Rest).
```

Note the use here of the variable _. This is referred to as the anonymous variable and is used in place of a named variable (such as A or _73) in cases where there is no further need to refer to this variable at any point later in the clause. In this case, once we have matched the head of the list to the element in the first argument, we no longer need the rest of the list, so use _ to denote it.

`member/2` is probably the most commonly used predicate in Prolog but it is not a built-in predicate, in otherwords, it is not available to the Prolog interpreter by default. Everytime you want to use it you must include it in your code explicitly or consult a file which contains it. You will find that there are a lot of predicates like this (a lot of which you will develop in the practical sessions) and it is useful to make a file of these predicates and then consult this file whenever you need them (consulting will be dealt with in chapter 10).

8.2.5 The `prlist` Predicate

The `prlist/1` predicate prints out each element of a list on a separate line. The Prolog code is as follows:

```
prlist([]).

prlist([Head|Rest]):-
        nl, write(Head),
        prlist(Rest).
```

8.3 Lists

8.3.1 Basics

If we need to put a number of items together in one structure, perhaps in order to manipulate them as a single structure, we can build a data structure called a **list**. An example would be a list of all the shops in the `books` program:

List	Head	Tail
`[a, b, c, d]`	`a`	`[b, c, d]`
`[a]`	`a`	`[]`
`[]`	fails	fails
`[[the, cat], sat]`	`[the, cat]`	`[sat]`
`[the, cat]`	`the`	`[cat]`
`[the, [cat, sat]]`	`the`	`[[cat, sat]]`
`[the, [cat, sat], down]`	`the`	`[[cat, sat], down]`
`[X, Y, Z]`	`X`	`[Y, Z]`

Figure 19: Some lists and their heads and tails

```
[better_books, menzies, edinbooks, james_thin]
```

A list is defined as follows:

A list is an ordered sequence of elements.

Lists can be used to represent practically any kind of structure. In fact, lists are such a general data structure that computing languages have been built based entirely upon them; this is true of the language Lisp, for example.

A list can be either

- an empty list, written as '`[]`'; or
- a list containing one or more elements, each separated by commas and all enclosed in square brackets ('`[`' and '`]`').

Here are some examples of lists:

- `[]`
- `[a, b]`
- `[a, X, j]`

8.4 Manipulating Lists

A list can be manipulated by splitting it into its **head** and its **tail**. The head of the list is the first element of the list, and the tail of the list is the rest of the list. Further examples of lists, together with their heads and tails, are shown in Figure 19.

In order to split lists into their heads and tails, we use a **list destructor**. This is represented by the character '`|`'. So, the list with head `X` and tail `Y` is represented as

```
[X|Y]
```

8.4.1 Matching Lists

By using the list destructor together with Prolog's unification (Prolog's matching mechanism), we can easily split any list in order to access any of its elements (or to transform lists in some way, or build new ones).

If we match the two lists

```
[X|Y]
```

and

```
[the, little, dog]
```

then `X` is instantiated to `the`, and and `Y` is instantiated to `[little, dog]`. Some other examples of attempts to unify lists are shown in Figure 20

Element order is important:

- `[foo,bar,baz]` is not the same as `[baz,bar,foo]`

8.4.2 Constructing and Destructing Lists

The basic approach:

- To take a list apart, split the list into the first element and the rest of the list.
- To construct a list from an element and a list, insert the element at the front of the list.

List Destruction:

```
[A|B] = [1,2,3,4]

A = 1
B = [2,3,4]
```

The first element is the **head** of the list; the remainder is the **tail** of the list.

List Construction:

- Take a variable bound to a list: for example

  ```
  OldList = [happy,sad]
  ```

- Add the new element to the front:

```
?- [john,june,A] = [X,Y,tom].   A=tom X=john Y=june    yes
    (simple match of variables and constants)

?- [H|T] = [c,b,a].             H=c T=[b,a]    yes
    (splits list into head and tail)

?- [H|T] = [].                  no
    (the empty list cannot be split by the list destructor '|')

?- [A|[B,C]] = [c,b,a].         A=c B=b C=a    yes
    ([A|[B,C]] same thing as [A,B,C])

?- [a,X] = [X,b].               no
    (X cannot unify with two different constants a and b)

?- [sue,[dick,wendy],P] = [P,Q,R].      P = sue, Q = [dick,wendy],
    (P and R share)                     R = sue     yes

?- [X|Y] = [a(b,c),b,c].        X = a(b,c), Y = [b,c]  yes


?- [[X],Y] = [a,b].     no
    (constant 'a' cannot match to list [X])

?- [a|X] = [A,B,y]              A = a, X = [B,y]  yes

?- [fred|[]] = [X].             X = fred        yes
    ([fred|[]] same as [fred])

?- [a,b,X,c] = [a,b,Y].         no
    (different length lists)

?- [H|T] = [tom,dick,mary,fred]. H= tom T = [dick,mary,fred]   yes

?- [[sue,tom],[10,9]] = [names,ages].   no
    (constants don't match lists)
```

Figure 20: Some examples of list matching

```
NewList = [grumpy|OldList]
```

Bigger Chunks:

- You can always directly access any number of elements at the head of a list.
- To add `grumpy` and `elated` to `OldList`:

  ```
  NewList = [grumpy, elated|OldList]
  ```

- To remove three elements: suppose `OldList` is bound to a list of three or more elements, then

  ```
  OldList = [One,Two,Three|Remainder]
  ```

8.5 Summary

In the next section, we go on to look at list manipulation in more detail.

8.6 Exercises

Question 8.1 How many elements are there in each of the following list structures?

```
e.g.    the length of [foo,bar,baz]  is 3

        the length of [foo,1,[a(X)],[1],[foo,bar]] is 5

8.1a     [a,[a,[a,[a]]]]

8.1b     [1,2,3,1,2,3,1,2,3]

8.1c     [a(X),b(Y,Z),c,X]

8.1d     [[sum(1,2)],[sum(3,4)],[sum(4,6)]]

8.1e     [c,[d,[x]],[f(s)],[r,h,a(t)],[[[a]]]]
```

Question 8.2 The predicate = takes 2 arguments and tries to unify them. For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables;
- if it fails, explain why it fails.

```
e.g.   ?- [H|T] = [1,2,3].               yes    H=1, T=[2,3].

       ?- [a, b] = [c, A].               no     because a and c are constants
                                                and constants cannot unify
                                                with each other.

       ?- [A,john,C] = [jim,B,tom].      yes    A=jim, B=john, C=tom.
```

```
8.2a   ?- [A,B,C,D] = [a,b,c].

8.2b   ?- bar([1,2,3]) = bar(A).

8.2c   ?- [X,[]] = [X].

8.2d   ?- [X,Y|Z] = [a,b,c,d].

8.2e   ?- [1,X,X] = [A,A,2].

8.2f   ?- likes(Y,a) = likes(X,Y).

8.2g   ?- [foo(a,b),a,b] = [X|Y].

8.2h   ?- [b,Y] = [Y,a].

8.2i   ?- [H|T] = [red,blue,b(X,Y)].

8.2j   ?- [1,[a,b],2] = [[A,B],X,Y].

8.2k   ?- test(a,L) = test(E,[b,c,d]).

8.2l   ?- [[a,[b]],C] = [C,D].

8.2m   ?- [fred|T]=[H|[sue,john]].
```

Question 8.3 Imagine that this program is consulted by the Prolog interpreter:

```
foo([],[]).

foo([H|T], [X|Y]):-
        H = X,
        foo(T,Y).
```

[Note: this program tests if two lists unify by testing if the heads unify then recursing on the tails]

What will be the outcome of each of the following queries?

```
8.3a   ?- foo([a,b,c], A).

8.3b   ?- foo([c,a,t], [c,u,t]).

8.3c   ?- foo(X, [b,o,o]).

8.3d   ?- foo([p|L], [F|[a,b]]).

8.3e   ?- foo([X,Y], [d,o,g]).
```

9 Manipulating lists

9.1 Introduction

In this section we will look more closely at manipulating lists.

9.2 The predicate `member/2`

9.2.1 Building the predicate

Suppose we have a list of names like the following:

```
[fred, john, ann, mary].
```

and we want to know if a given person is in this list.

We first ask if the person has the same name as the head of the list:

- if so, we have succeeded;
- if not, then we see if the person is one of the rest of the list, that is, if the person is in the tail of the list.
- So, we take the tail of the list and ask if the person has the same name as the head of this list:
  - if so, we have succeeded;
  - if not, then we see if the person is in the tail of this list.
  - So, we take the tail of the list ...

And so on, until we run out of list and fail to find the person.

So we need to write a set of clauses to test membership of a list. We will call it the `member` predicate. It will need two arguments, the item to be looked for and the list to look for it in. We'll refer to this predicate as `member/2`.

In this predicate,

- either the item will be the head of the list;
- or it will be in the tail of the list;
- or there will be no match and it will fail.

So we need to consider these three possibilities, or cases. The first clause we can specify, in English, as:

> X is a member of a list if X is the same as the head of the list.

The second clause will be:

X is a member of a list if it is a member of the tail of the list.

Consider how we write these in Prolog. The first argument of the `member/2` predicate is the element to be found, and the second is the list:

```
member(X, AList):-
```

But we have to somehow get at the head of this list. We use the list destructor and unification to do this for us:

```
member(X, [H|T]):-
```

So, the first case becomes

```
member(X, [H|T]):-
        X = H.
```

and the second becomes:

```
member(X, [H|T]):-
       X \== H,
       member(X, T).
```

Note that here `member/2` has the subgoal `member/2`, so by definition it is **recursive**: it calls itself.

9.2.2 Cleaning Up

We have some redundancy in these two clauses.

Let's take the first clause first:

```
1:      member(X, [H|T]):-
                X = H.
```

We want to test whether or not `X` and `H` unify. It is easier to test that by calling them both the same; that is, by making them the same variable. Note that doing this makes the subgoal redundant:

```
1:      member(X, [X|T]).
```

If this succeeds then we never test the second clause. Put another way, if we reach the second clause, the goal `X = H` must have failed. So by the time we get to the second clause, we know that `X \== H` and so we do not need to test it again. The second clause then becomes:

```
2:      member(X, [H|T]):-
               member(X, T).
```

9.2.3 Recursion in List Processing

Any program that calls itself as a subgoal is **recursive**. A lot of other list processing programs are recursive. They have the following general pattern:

- split the list;
- do something to the head of the list (test it or process it); and
- recurse on the tail of the list.

Or, in a schematic Prolog form:

```
recpred([H|T]):-
        test(H),
        recpred(T).
```

In recursive predicates there are usually at least two cases:

1. the **base case** or **boundary condition**, which stops the recursion (eventually): this could be when some condition is true (for example, when the name sought is the head of the list, or when we are left with the empty list);
2. the **recursive** case, which has the goal that recurs, often calling subgoals with shorter and shorter lists (the tail of original list, the tail of the tail of the original list, and so on) as arguments.

An example of a recursive program we have already encountered is `likes/2`, in the case where some person *likes* another if both the person and the other *likes* the same drink.

9.2.4 And/Or Trees for Recursive Predicates

Here is the predicate `member/2` with an example goal and the AND/OR tree to illustrate the execution of this goal:

First, the `member/2` predicate:

```
1:      member(X, [X|T]).
2:      member(X, [H|T]):-
                  member(X, T).
```

Suppose we present Prolog with the following goal:

```
| ?- member(ann, [fred, john, ann, mary]).
```

The behaviour of Prolog that follows is represented in the AND/OR tree in Figure 21.

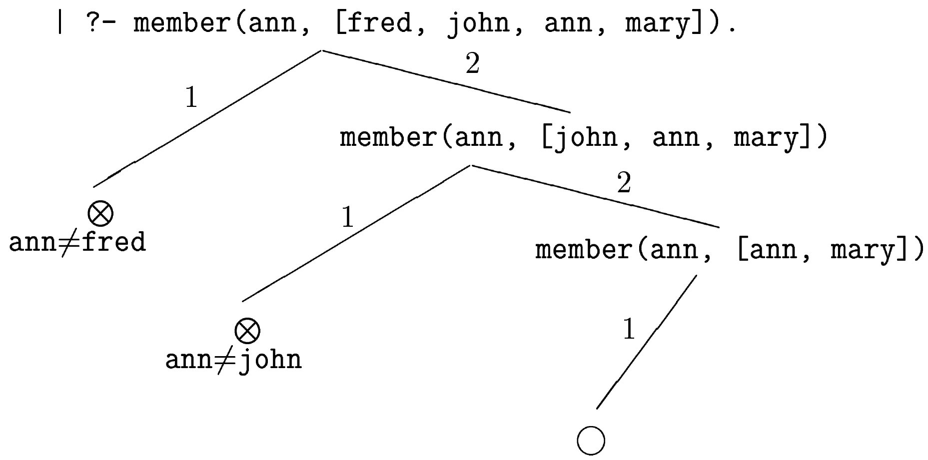


Figure 21: An AND/OR tree representation of `member`

9.3 Other List Processing Predicates

9.3.1 Printing a list of elements

Suppose we have a list `[a, b, c, d]`, and we want to print it out. We can define a predicate `prlist/1`, which takes a list as its argument and simply uses `write/1` to write it out:

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

For our example list above, this would produce the output

```
| ?- prlist([a, b, c, d]).
[a,b,c,d]
yes
| ?-
```

However, suppose we want to print each element of the list on a separate line. This means we have to print the elements one at a time. We can define a recursive procedure to do this:

To `prlist` a list of elements:

- write the head of the list; and
- `prlist` the tail of the list.

The Prolog code to do this is as follows:

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

We also need to know when to stop, of course. The printing should stop when we have no more elements to write, i.e., when the list to be `prlist`ed is the empty list. So, we also need the following clause:

```
prlist([]).
```

The full program is then as follows:

```
1:      prlist([]).


          \item[{\rm 2:}]
          prlist([H|T]):-
                 write(H), nl,
                 prlist(T).
```

9.3.2 Checking that no element of a list of letters is a consonant

Suppose we want to be able to test each of the elements of a list to check that none of them are consonants. We can define a predicate `no_cons/1`, which has the same basic pattern as before: we test or process the head of the list, and then recurse on the tail of the list. The empty list provides the base case.

The test we'll use here is actually whether the element is a vowel, rather than whether it isn't a consonant. The code is then as follows; note that we have to include facts that tell Prolog which characters are vowels.

```
1:      no_cons([]).
2:      no_cons([H|T]):-
                  vowel(H),
                  no_cons(T).
3:      vowel(a).
4:      vowel(e).
5:      vowel(i).
6:      vowel(o).
7:      vowel(u).
```

Here are some example goals presented to Prolog with this database loaded:

```
max([H|T], Answer):-
        max(T, H, Answer).

max([], Answer, Answer).
max([H|T], Temp, Answer):-
        H > Temp,
        max(T, H, Answer).
max([H|T], Temp, Answer):-
        max(T, Temp, Answer).
```

Figure 22: The `max` predicate

```
| ?- no_cons([a,e,e,o,u]).

yes
| ?- no_cons([a,r,e]).

no
| ?-
```

9.4 More List Processing Predicates

9.4.1 Finding the maximum of a list of numbers

The predicate `max/2` has two arguments: the first is a list of numbers, and the second is the maximum of that list. If the predicate is called with the second argument uninstantiated, then this argument will become instantiated to the maximum of the list; this might be thought of as returning a single element as the result of processing the complete list.

As before, the head of the list is processed, and the program then recurses on the rest of the list. The Prolog code is shown in Figure 22.

Note that the predicate `max/3` is called by the predicate `max/2`. The additional middle argument is the current (temporary) maximum, instantiated each time a higher value is found as the program recurses down the list. Initially this is set to the head of the list.

`max/3` has three cases:

- The first case of `max/3` is the base case: if the empty list is being processed, then the current maximum is the final one.
- In the second case, if the head of the list is greater than the current maximum then it becomes the new current maximum, and the rest of the list is recursed on.
- In the third case, if the head of the list is not greater than the current maximum then the current maximum stays the same, and again the rest of the list is recursed on.

```
sentence:-
      noun(N1), write(' '), write(N1),
      verb(V1), write(' '), write(V1),
      noun(N2), write(' '), write(N2).

noun(fred).
noun(beer).
noun(doris).
noun(gin).

verb(likes).
verb(drinks).
```

Figure 23: A simple sentence generator

9.4.2 Building new list structures

As well as recursing down a list, testing all the elements (as we did in `member/2` and `no_cons/1`), or returning a single element (as we did in `max/2`), we may want to process each element and build a completely new list data structure.

Suppose, for example, we want to generate a sentence. We could simply pattern match for each word category (to begin with, *noun* and *verb*) and write the resulting word: such a generator might look like that shown in Figure 23.

Note that this code provides no way of saving the sentence generated as a whole. To get around this, we could make each new word a successive element of a new list. We can do this in the head of the `sentence` clause directly:

```
sentence([N1,V1,N2]):-
        noun(N1),
        verb(V1),
        noun(N2).
```

In a later section, we will look at the special mechanism Prolog provides for writing grammar rules.

9.5 Exercises

Question 9.1 The predicate member/2 succeeds if the first argument matches an element of the list represented by the second argument.

```
e.g.    ?- member(1,[2,3,1,4]).         yes

member/2 is defined as:        1. member(El,[El|T]).
```

```
                              2. member(El,[H|T]):-
                                     member(El,T).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
9.1a ?- member(a,[c,a,b]).

9.1b ?- member(a,[d,o,g]).

9.1c ?- member(one,[one,three,one,four]).

9.1d ?- member(X,[c,a,t]).

9.1e ?- member(tom,[[jo,alan],[tom,anne]]).
```

9.1f Complete the AND/OR tree below which represents the execution of the query:

```
        ?- member(a,[b,r,e,a,d]).
        /        \
       /1       2 \
      /            \
    a=b          member(a,[r,e,a,d])
   fails        /         \
          1  /          2 \
            /              \
          a=r          member(a,[e,a,d])
         fails
```

Question 9.2 The predicate no_cons/1 succeeds if all elements of the list represented by the one argument are vowels (as specified by vowel/1).

```
e.g.    ?- no_cons([a,e,i]).
        yes

        ? no_cons([a,b,c]).
        no

        no_cons/1 is defined as:
        1. no_cons([]).
        2. no_cons([H|T]):-vowel(H),no_cons(T).
```

```
3. vowel(a).
4. vowel(e).
5. vowel(i).
6. vowel(o).
7. vowel(u).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
9.2a ?- no_cons([a,a,e,i]).
```

```
9.2b ?- no_cons([A,e,e]).
```

9.2c Complete the AND/OR tree below which represents the execution of the query:

```
       ?- no_cons([a,e,i]).
         /      |\
      1/       2|__\                      H/a      T/[e,i]
      /         |    \
  a=[]       vowel(a) no_cons([e,i])
 fails          |        |\
               3|       2|__\                      H1/e    T1/[i]
                |        |    \
            succeeds  vowel(e) ......
                         |
                        4|
                         |
                     succeeds
```

Question 9.3 The predicate max/2 succeeds if the first argument is a list of numbers and second argument unifies with the maximum value in that list.

```
e.g.    ?- max([2,4,6,8],M).
        M = 8
        yes
```

```
max/2 is defined as:
```

```
0. max([H|T],Max):-
        max(T,H,Max).
1. max([H|T],Temp,Max):-
        H>Temp,
```

```
        max(T,H,Max).
2. max([H|T],Temp,Max):-
        max(T,Temp,Max).
3. max([],Finalmax,Finalmax).
```

9.3 Complete the AND/OR tree below which represents the execution of the query:

```
?- max([2,1,4,3],Ans).
           |
          0|             H/2      T/[1,4,3]
           |
   max([1,4,3],2,Ans)
      /     \
   1/        2\                   H1/1    Temp/2  T1/[4,3]
  /            \
1>2       max([4,3],2,Ans)
fails          |\
              1|__\                       H2/4    Temp1/2   T2/[3]
               |    \
              4>2   .......
          succeeds
```

Question 9.4 The predicate prlist/1 is supposed to write out the elements of a list structure, regardless of the levels of embedding that are present in the list.

```
 e.g. intended behaviour:
        ?- prlist([a,b,[c,d,e],f,[g]]).
        abcdefg
        yes
```

Instead, the predicate as defined below has the following behaviour:

```
?- prlist([a,b]).
ab[]
yes

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

prlist/2 is defined as:

1. prlist([H|T]):-
```

```
        prlist(H),
        prlist(T).
2. prlist(X):-
        write(X).
```

Explain why this predicate produces this behaviour (instead of the intended behaviour), using an AND/OR tree or a trace in your explanation.

Question 9.5 The predicate checkvowels takes a list representing a word, checks each letter to see whether it is a vowel, and if it is it writes out the vowel.

```
checkvowels([H|T]):-
        vowel(H),
        write(H), nl,
        checkvowels(T).

checkvowels([H|T]):-
        checkvowels(T).

checkvowels([]).

vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).
```

```
e.g.    ?- checkvowels([c,a,t,i]).
        a
        i
        yes
```

9.5a Using this as a model, write a predicate results/1 that takes a list of names, checks whether each is a pass (using a predicate that you also must define, pass/1) and writes out the names if they pass.

For example, the following query:

```
?- results([tom,bob,sue,jane]).
```

should give the output:

```
bob
sue
yes
```

9.5b Modify this program so that instead of 'writing out' the name of each person who passes it should output a list of them. (You may need to read ahead to answer this part).

```
?- results2([tom,bob,sue,jane],Passlist).
Passlist = [bob,sue]
yes
```

9.6 Practical 3: Basic List Processing

Introduction

Here are some exercises in list-processing, which is a very common technique in Prolog, and one which you should try to get comfortable with. Since lists are defined recursively as a head, followed by a tail, which is itself a list, predicates which process lists are typically recursive, and in which you:

- Hit the base case when there's no more of the list to process.
- Otherwise, do something with the head of the list, then recurse with the tail to do the rest of the processing.

These exercises are *not* assessed, but you are expected to complete them. Your Prolog code should be entered in the spaces provided. After you have tested the code on the computer show it to the demonstrator and they will mark that section as complete.

1. Define two predicates, `evenlength/1` and `oddlength/1`, which take a list as their argument, and succeed if that list has an even length and an odd length respectively. For example:

```
| ?- evenlength([1,2,3,4]).    ________________________________
yes                            ________________________________
                               ________________________________

| ?- oddlength([1]).           ________________________________
yes                            ________________________________
                               ________________________________
```

2. Define a predicate `palindrome/1` which succeeds if its single argument is a list which is the same in the forward and backward directions. For example:

```
| ?- palindrome([r,e,d,i,v,i,d,e,r]).
yes
_________________________________________________
_________________________________________________
_________________________________________________
_________________________________________________
```

3. Define a predicate flatten/2, whose first argument is a list of any complexity, and which succeeds by instantiating its second argument to a 'flattened' version of the list, which contains the contents of all of the embedded lists. For example:

```
| ?- flatten([a, [[b]], [[], [d, e]]], X).

X = [a, b, d, e]
yes
-------------------------------------------------
-------------------------------------------------
-------------------------------------------------
-------------------------------------------------
```

4. Define a predicate count_atoms/2, which accepts a list containing atoms, numbers and/or such lists, and 'returns' a list containing terms of the form Item=Count to show how many times an Item appears in any embedded list. For example:

```
| ?- count_atoms([a, b, b, [[a], [c], b]], L).

L = [a=2, b=3, c=1]
yes
-------------------------------------------------
-------------------------------------------------
-------------------------------------------------
-------------------------------------------------
```

Note that the order of the count terms in the result doesn't matter; any order will do.

5. Define a predicate replace_elements/4, which replaces all occurrences of a given element in a list by another, and instantiates a given variable to the answer. The arguments should be, in order:

   (a) the element to be replaced;
   (b) the element to replace it with;
   (c) the list to do the replacing in;
   (d) a variable to be instantiated to the final list.

   (The predicate needn't bother to delve inside lists within lists.) For example:

```
| ?- replace_elements(pronoun(he),pronoun(we),
        [pronoun(he),verb(said),pronoun(he),verb(did)], L).

L=[pronoun(we), verb(said), pronoun(we), verb(did)]
yes
-------------------------------------------------
```


----

6. Define a predicate `unifiable/3(List, Term, Answer)`, where `Answer` is to be instantiated to all those terms in `List` which *could* unify with `Term`. Make sure that they are not actually unified, though. For example:

```
| ?- unifiable([X, b, t(Y)], t(a), L).
```

should give the answer:

```
L = [X, t(Y)]
```

and *not*:

```
L = [t(a), t(a)]
```

(Hint: consider the behaviour of `\+(Element=Term)` carefully).

----

10 Further list processing predicates

10.1 Changing one sentence into another: the predicate alter/2

Suppose you want to write a simple predicate that enables you to alter an input sentence to get a new output sentence (Eliza style). An example of the sort of dialogue produced might be:

```
You: you are a computer
Prolog: i am not a computer

You: do you speak french
Prolog: no i speak german
```

How do we go about this? First we break the task down into steps.

1. Accept the sentence typed in;
2. If there are any *you*'s change them to *i*'s
3. and change *are* to *am not*
4. and change *french* to *german*
5. and change *do* to *no*

[this may lead to some obvious problems in other examples, but we will ignore those for now]

The predicate we will use will be called `alter/2`. It will need two arguments, both of which will be lists.

An example goal might be:

```
| ?- alter([do,you,know,french],Rep).

Rep=[no,i,know,german]
```

How is `alter/2` defined? Think of cases to be dealt with:

- the list to be altered
- the empty list

Take the latter:

> Alter the empty list to the empty list

```
alter([],[]).
```

This will give us the boundary condition and will stop the recursion.

What about the rest of the list to be altered?

- change one word at a time (or leave it if not to be changed)
- build a new list of changed and unchanged words (=reply)

We do this by changing the head of the list and then recursing on the tail, building a new list as we go and stopping when we run out of list.

1. - Change the head of the input list (represented by the first argument) into another word and
   - let the head of the output list be the same as that word (by matching)
2. - Alter the tail of the input list and
   - let the tail of the output list be the same as the altered tail
3. If the end of the input list is reached then there is no more to do.

We now have to deal with changing one word into another. We will define a predicate `change/2` that will take two arguments: the element to be changed, and the element that it is to be changed to:

```
change(you,i).
change(are,[am,not]).
change(french,german).
.....
.....
change(X,X).
```

As these are all single list elements, any replacement of more than one word will have to be a sub-list. Note the *catchall* in the last clause of `change/2`. This will take care of all the words that we don't want changed.

So, the two clauses that make up the predicate `alter/2` are:

```
1:      alter([],[]).
2:      alter([H|T],[X|Y]):-
           change(H,X),
           alter(T,Y).
```

If we now put the whole program together with predicates for *changing* and *altering* we get the following:

```
1:      alter([],[]).
```

```
| ?- trace,alter([i,like,your,shirt],P).
{The debugger will first creep -- showing everything (trace)}
   1  1  Call: alter([i,like,your,shirt],_99) ?
     2  2  Call: change(i,_237) ?
     2  2  Exit: change(i,you) ?
     3  2  Call: alter([like,your,shirt],_238) ?
       4  3  Call: change(like,_505) ?
       4  3  Exit: change(like,like) ?
       5  3  Call: alter([your,shirt],_506) ?
         6  4  Call: change(your,_772) ?
         6  4  Exit: change(your,my) ?
         7  4  Call: alter([shirt],_773) ?
           8  5  Call: change(shirt,_1038) ?
           8  5  Exit: change(shirt,shirt) ?
           9  5  Call: alter([],_1039) ?
           9  5  Exit: alter([],[]) ?
         7  4  Exit: alter([shirt],[shirt]) ?
       5  3  Exit: alter([your,shirt],[my,shirt]) ?
     3  2  Exit: alter([like,your,shirt],[like,my,shirt]) ?
   1  1  Exit: alter([i,like,your,shirt],[you,like,my,shirt]) ?

P = [you,like,my,shirt] ?

yes

(the trace is indented for additional clarity)
```

Figure 24: Tracing the behaviour of the predicate `alter/2`

```
2:      alter([H|T],[X|Y]):-
           change(H,X),
           alter(T,Y).
1:      change(i,you).
2:      change(me,you).
3:      change(your,my).
4:      change(their,our).
5:      change(X,X).
```

An example trace of the program is given below in Figure 24.

We now will consider a few more examples of list processing predicates, looking at some of the general techniques used in them.

10.2 Deleting the first occurrence of an element from a list: the predicate `delete/2`

What do we mean? The predicate `delete/3` will need 3 arguments:

- the element E to be deleted
- the list L from which it is to be deleted
- the new list Newl which has the item deleted

We assume we know the first 2 and want to build the third, so the goal might be:

```
?- delete(a,[c,a,m,e,l],Ans).
```

So we would expect the result:

```
Ans=[c,m,e,l]
Yes
```

What do we do? We want to look at each element of the list L in turn to see if it is the element to be deleted, so need to access the head of L recursively (as in `member/2`). This means breaking L into a Head HL and a tail TL (using matching). We want to process the whole list and to build a new list at the same time.

The two cases we need to consider are:

1. when the head of the list that we are looking at is the one we want to delete
2. when it is not

In the first case we then need to save the rest of the list:

> deleting the element E from the list with head E and tail TL will give the list TL

In prolog:

```
delete(E,[E|TL],TL).
```

In the second case we need to save the head as well as the rest of the list (we don't want to delete all the other elements):

> deleting the element E from the list with head HL and tail TL will give the list with head HL and tail NL if deleting E from the list TL gives the list NL

```
delete(E,[HL|TL],[HL|NL]):-
        delete(E,TL,NL).
```

```
| ?- trace,delete(a,[c,a,p],Nlist).
| ?- delete(a,[c,a,p],Nlist).
   1  1  Call: delete(a,[c,a,p],_100) ?
     2  2  Call: delete(a,[a,p],_241) ?
     2  2  Exit: delete(a,[a,p],[p]) ?
   1  1  Exit: delete(a,[c,a,p],[c,p]) ?

Nlist = [c,p] ?

yes
```

Figure 25: Tracing the behaviour of the predicate `delete/3`

So, the complete program is:

```
delete(E,[E|TL],TL).

delete(E,[HL|TL],[HL|NL]):-
        delete(E,TL,NL).
```

And example goals would be:

```
| ?- delete(a,[c,a,p],Nlist).

     Nlist=[c,p]
yes

| ?- delete(e,[f,e,e,d],Ans).

     Ans=[f,e,d]
yes
```

An example trace of this is shown in Figure 25.

Note: we can use this also to help build a predicate for deleting ALL the elements E in the list L - consider what changes we need to the first clause......

10.3 Reversing a list: the predicates `rev/2` and `rev/3`

We are going to write a predicate to reverse a list here. The predicate will be `rev/3`. For convenience it can be called by a simpler predicate `rev/2`.

```
| ?- listing(rev).
```

```
| ?- rev([a,b,c],Revl).
   1  1  Call: rev([a,b,c],_86) ?
     2  2  Call: rev([a,b,c],[],_86) ?
       3  3  Call: rev([b,c],[a],_86) ?
         4  4  Call: rev([c],[b,a],_86) ?
           5  5  Call: rev([],[c,b,a],_86) ?
           5  5  Exit: rev([],[c,b,a],[c,b,a]) ?
         4  4  Exit: rev([c],[b,a],[c,b,a]) ?
       3  3  Exit: rev([b,c],[a],[c,b,a]) ?
     2  2  Exit: rev([a,b,c],[],[c,b,a]) ?
   1  1  Exit: rev([a,b,c],[c,b,a]) ?

Revl = [c,b,a] ?

yes
```

Figure 26: Tracing the behaviour of the predicate `rev/3`

```
1:      rev(L,Revl) :-
                   rev(L,[],Revl).

1:      rev([],L,L).

2:      rev([H|List],Acc,Revl) :-
                   rev(List,[H|Acc],Revl).
```

We 'pour' each element into the accumulator one at a time, so that the new list builds up with each successive head at the front (and the first head being in first is now last).

When all are 'poured in' we copy the accumulator list across to the answer.

An example trace of `rev/3` is shown in Figure 26.

10.4 Joining two lists together: the predicates append/3

The predicate `append/3` enables us to join two lists together into one list.

```
| ?- listing(append).

1:      append([],L,L).

2:      append([H|L],M,[H|N]) :-
                   append(L,M,N).
```

1. The list L is the same as the list L appended to the empty list.

```
| ?- append([a],[b],Newl).
   1  1  Call: append([a],[b],_88) ?
     2  2  Call: append([],[b],_224) ?
     2  2  Exit: append([],[b],[b]) ?
   1  1  Exit: append([a],[b],[a,b]) ?

Newl=[a,b]
yes
```

Figure 27: Tracing the behaviour of the predicate `append/3`

```
| ?- append([1,2],[3,4],X).
   1  1  Call: append([1,2],[3,4],_128) ?
     2  2  Call: append([2],[3,4],_264) ?
       3  3  Call: append([],[3,4],_363) ?
       3  3  Exit: append([],[3,4],[3,4]) ?
     2  2  Exit: append([2],[3,4],[2,3,4]) ?
   1  1  Exit: append([1,2],[3,4],[1,2,3,4]) ?

X=[1,2,3,4]
yes
```

Figure 28: A further example of the behaviour of the predicate `append/3`

2. The list with head H and tail N is the same as the list with head H and tail L appended to the list M if the list N is the list L appended to the list M.

An example trace of `append/3` is shown in Figure 27.

A further trace of `append/3` using a different goal is shown in Figure 28.

An example trace of using `append/3` with different instantiation patterns in the goal is shown in Figure 29.

What about

```
?- append(F,[3,4],[1,2,3,4]).

     F=[1,2]
yes
```

10.5 Exercises

Question 10.1 The predicate delete/3 succeeds if deleting the element represented by the first argument, from the list represented by the second argument, results in a list represented

```
| ?- append([1,2],R,[1,2,3,4]).
   1  1  Call: append([1,2],_96,[1,2,3,4]) ?
     2  2  Call: append([2],_96,[2,3,4]) ?
       3  3  Call: append([],_96,[3,4]) ?
       3  3  Exit: append([],[3,4],[3,4]) ?
     2  2  Exit: append([2],[3,4],[2,3,4]) ?
   1  1  Exit: append([1,2],[3,4],[1,2,3,4]) ?

R=[3,4]
yes
```

Figure 29: A trace of a different calling pattern of the predicate `append/3`

by the third argument.

```
e.g.    ?- delete(a,[a,p,p,l,e],A).
        A=[p,p,l,e]
        yes

        delete/3 is defined as:

        1. delete(El,[El|T],T).
        2. delete(El,[H|T],[H|NT]):-
                 delete(El,T,NT).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

`10.1a ?- delete(X,[pat,john,paul],Ans).`

`10.1b ?- delete(A,L,[t,o,p]).`

`10.1c ?- delete(a,[c,a,b],Ans).`

`10.1d ?- delete(e,[d,o,g],P).`

`10.1e ?- delete(e,[f,e,e,t],Ans).`

Question 10.2 The predicate deleteall/3 succeeds if deleting all occurrences of the element represented by the first argument from the list represented by the second argument results in a list represented by the third argument.

```
e.g.    ?- delete(p,[a,p,p,l,e],A).     A=[a,l,e]       yes

deleteall/3 is defined as:      1. deleteall(El,[],[]).
                                2. deleteall(El,[El|T],NT):-
                                        deleteall(El,T,NT).
                                3. deleteall(El,[H|T],[H|NT]):-
                                        deleteall(El,T,NT).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
10.2a  ?- deleteall(e,[f,e,e,t],Ans).

10.2b  ?- deleteall(p,[d,o,g],X).
```

10.2c Complete the AND/OR tree below which represents the execution of the query:

```
        ?- deleteall(e,[f,e,e,t],Ans).
          /     |         \
        1/      |2       3 \ Ans/[f|NT]
        /       |           \
[]=[f,e,e,t]   e=f    deleteall(e,[e,e,t],NT)
  fails       fails     /    \
                       /      \
                   1 /       2\ NT/NT2
                    /          \
             []=[e,e,t]    deleteall(e,[e,t],NT2)
              fails
```

10.2d Suppose that the predicate deleteall/3 is defined incorrectly as:

```
        1. delall(E,[],[]).
        2. delall(E,[E|T],Y):-
                delall(E,T,Y).
        3. delall(E,[H|T],Y):-
                delall(E,T,[H|Y]).
```

resulting in the behaviour:

```
i.  ?- delall(e,[f,e,e,t],Ans).
   no
```

However, the following query succeeds, as intended:

```
ii. ?- delall(a,[a,a,a],Ans).
    Ans=[]
    yes
```

Explain why the program does not give the intended answer to query i. using an AND/OR tree or a trace to illustrate your answer.

Question 10.3 The predicate repall/4 succeeds if replacing all occurrences of the element represented by the first argument, by the element represented by the second argument, in the list represented by the third argument, results in a list represented by the fourth argument.

```
e.g.    ?- repall(p,b,[a,p,p,l,e],A).
        A=[a,b,b,l,e]
        yes

        repall/4 is defined as:

        1. repall(El,Rel,[],[]).
        2. repall(El,Rel,[El|T],[Rel|NT]):-
                repall(El,Rel,T,NT).
        3. repall(El,Rel,[H|T],[H|NT]):-
                repall(El,Rel,T,NT).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
10.3a ?- repall(e,a,[f,e,a,t],Ans).

10.3b ?- repall(P,r,[s,o,u,p],Ans).
```

Question 10.4 The predicate `whowants/3` has three arguments. The first is intended to represent a type of food; the second a list of people; and the third another list, representing those people on the 2nd argument list who want the type of food specifed in the 1st argument (where want/2 is defined separately, with two arguments representing who wants what food).

```
e.g.    ?- whowants(beans,[jo,tom,ann],Who).
        Who = [jo,ann]
        yes

        1. whowants(Food,[Name|Rest],[Name|Others]):-
                wants(Name,Food),
                whowants(Food,Rest,Others).
```

```
2. whowants(Food,[Name|Rest],Others):-
        whowants(Food,Rest,Others).
3. whowants(Food,[],[]).

4. wants(jo,chips).
5. wants(jo,beans).
6. wants(jo,eggs).
7. wants(ann,beans).
8. wants(ann,bacon).
9. wants(tom,eggs).
10. wants(tom,chips).
11. wants(rick,bacon).
```

10.4a Give either the AND/OR tree or a trace which represents the execution of the query:

```
?- whowants(beans,[jo,tom,ann],Who).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

`10.4b ?- whowants(chips,[ann,rick],Ans).`

`10.4c ?- whowants(bacon,[tom,ann],A).`

`10.4d ?- whowants(eggs,Diners,Ans).`

10.4e Using `whowants/3` as a model, write a predicate `overage/3` that takes an age limit and a list of names, checks the age of each person named (using a predicate `age/2`) and returns a list of names of those people who are over the age limit.

For example, the query below should give the output shown:

```
        ?- overage(18,[sally,alice,bill],Ans).
        Ans=[alice,bill]
        yes
where:
        age(sally,15).
        age(mark,26).
        age(bill,20).
        age(alice,37).
```

10.4f If the predicate whowants/3 had been (incorrectly) defined as:

```
1. whowants(Food,[Name|Rest],[Name|Others]):-
        wants(Name,Food),
        whowants(Food,Rest,Others).
2. whowants(Food,[Name|Rest],Others):-
        whowants(Food,Rest,Others).
```

(i.e. no 3rd clause) predict the outcome of the query in 4c.

```
?- whowants(bacon,[tom,ann],A).
```

and explain why this is the case, using an AND/OR tree or a trace in your explanation.

Question 10.5 The predicate deletefirst/3 is supposed to succeed if deleting the element represented by the first argument, from the list represented by the second argument, results in a list represented by the third argument, as delete/3 defined as above (B.)

```
e.g. intended behaviour:
        ?- deletefirst(i,[b,i,b,s],A).
        A=[b,b,s]
        yes

Instead, it produces the following behaviour:
        ?- deletefirst(i,[b,i,b,s],A).
        A=[s]
        yes

        ?- deletefirst(a,[b,a,l,l],X).
        no

        deletefirst/3 is defined as:

        1. deletefirst(El,[El|T],T).
        2. deletefirst(El,[H|T],NT):-
                deletefirst(El,T,[H|NT]).
```

Explain why this predicate produces this behaviour (instead of the intended behaviour), using an AND/OR tree or a trace in your explanation.

10.6 Practical 4: More list processing

In this practical we will start off decomposing and constructing terms using arithmetic operators in a similar way to manipulating lists. To complete this practical you will first need some tips.

Terms can be constructed from individual constants or variables by joining them together using any arithmetic operator or alphanumeric character. They can then be instantiated to a variable using `=/2`. If `is/2` is used then the variable will instantiate to the result of the arithmetic sum.

```
| ?- X=5,Y=6,Z = X+Y.
X = 5,
Y = 6,
Z = 5+6 ?
yes

| ?- X=5,Y=6,Z is X+Y.
X = 5,
Y = 6,
Z = 11
yes
```

Terms can also be decomposed using arithmetic operators.

```
| ?- X = pro+log, Z+Y = X.
X = pro+log,
Y = log,
Z = pro ?
yes

| ?- X = pro+log+ged, Z+Y = X.
X = pro+log+ged,
Y = ged,
Z = pro+log ?
yes
```

However, it is important to note that the binding preference of arithmetic operators is different to the list deconstructor `'|'`. The list deconstructor searches from the far left of a list first, designating the first argument it finds as the head and the rest the tail. Arithmetic operators work in reverse, matching the right-most instance of the operator and using that to split the term. An example of this can be seen in the second example above.

10.6.1 Part 1: List and term processing

1. Write a predicate, `list_to_term/2`, which takes as its first argument a non-empty list of numbers and returns as its second argument a term composed of the numbers in the list joined by `'+'` operators. An example of the behaviour of this predicate is:

   ```
   | ?- list_to_term([1,2,3,4], X).

   X = 1+(2+(3+4)) ?
   ```

```
yes

| ?- list_to_term([9,9,9], X).

X = 9+(9+9) ?
yes
| ?- list_to_term([44], X).

X = 44 ?
yes
```

(Hint: careful choice of the base case will make things easier.)

```
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
```

2. Write a predicate `inv/2` which takes as its first argument a term composed from `'+'` operators, `'-'` and numbers and returns as its second argument the same term but with the `'+'` changed `'-'` and the `'-'` changed to to `'+'`. An example of the behaviour of the program is:

```
| ?- inv(1+2+3,X).
X = 1-2-3
yes

| ?- inv(1+(2-3)+4,X).
X = 1-(2+3)-4
yes
```

```
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
```

3. Write a predicate `numset/2`, which takes as its first argument a term composed from `'+'` operators and numbers, and returns as its second argument a list containing the **set** of numbers appearing in the term. The order in which the numbers appear in the list is **not** important. An example of the behaviour of this program is:

```
| ?- numset(1+2+3+2,X).
X = [3,2,1]
yes

| ?- numset(1+(2+(1+2)),X).
X = [2,1]
yes
```

In answering this question you will probably need to use the standard `member/2` predicate:

```
member(X,[X|_]).
member(X,[_|T]):-
member(X,T).
```

```
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
```

10.6.2 Part 2: The Sticks Problem

The following puzzle requires complex list manipulation and rudimentary search (first proposed by John Hallam).

The Sticks Problem: There are 8 sticks lined up in a row (as shown below). The goal is to move exactly four sticks and end up with four groups of crossed sticks. Each move consists of picking up a stick, jumping over exactly two sticks, and then putting it down onto a fourth stick.

```
Start state:
| | | | | | | |
a b c d e f g h

Goal state:
X X X X
(position of the pairs is unimportant)

From start state, stick 'a' can be moved onto stick 'd':
  | | x | | | |
a b c d e f g h

And then stick 'c' could be moved onto 'e':
```

```
  |   x x | | |
a b c d e f g h
```

We can think of this problem as a **State-Space Search** problem as from every state there are a number of possible moves that may, or may not lead us to the goal state.

1. Using the list notation choose a representation for the states in this problem. Use your representation to show the start state and goal state:

   ```
   Start state: _____________________________________
   Goal state:  _____________________________________
   ```

2. Each move consists of picking up a stick, jumping over exactly two sticks, and then putting it down onto a fourth stick. How many different ways can this rule by applied? Specify an operator for each of these moves, stating the required starting state of the problem space and the result of making the move.

   State the operators in pseudo-code first:

3. Look at the operators you have just specified. Do you need all of them? Can general operators be specified that can apply in more than one situation? Modify the above specification to reflect this.

4. Download the `sticks.pl` file from the course website. Open it in a text editor (e.g. Emacs) and try to figure out how it works. Ask the demonstrators if you have an difficulties.

5. The `sticks` program uses two different search strategies to find a solution. If the program is called using `sticks([i,i,i,i,i,i,i,i],breadth)` then a breadth-first search is performed. If it is called using `sticks([i,i,i,i,i,i,i,i],depth)` then a depth first is performed. The difference in terms of how it searches the problem space is large but the difference between how the two strategies are implemented in Prolog is very small. Look at the code and see if you can figure out how the two strategies differ and what effect it has?

6. `sticks.pl` is currently missing the three predicates that allow it to make the actual moves. See if you can add these new predicates to the file and get the program to solve the search problem. Use the pseudo-code operators you have previously specified.

11 How Programs Work

11.1 Tracing

We can follow through the execution of a goal in Prolog by **tracing** it. The trace will show:

- which goal is currently being called, with what arguments;
- whether the goal succeeds or has to be redone (i.e., whether another match has to be looked for);
- whether the goal fails; and
- if the goal succeeds, whether other clauses might also match.

For example, given the predicate `no_cons/2`, the goal:

```
| ?- no_cons([a,e,o]).
```

can be traced by calling the conjunction of the goal `trace` and the goal itself; the code for `no_cons/2` and the results of tracing are shown in Figure 30.

Similarly we can trace the goal:

```
| ?- trace, max([7,3,9,2,6], Max).
```

giving the code and output as shown in Figure 31.

Some things to note here:

1. The number to the left of the goal indicates the level in the tree from which the goal is being called (the top level goal being 0). This makes it easier to see which goals are subgoals of the same clause, and also which are recursive goals.
2. The leftmost number is the number of the goal that is being called: each new goal that is created is given a number which is incremented for the next new goal. This makes it easier to see exactly which goal is being called, being redone, failing, or succeeding.

11.2 The Byrd Box Model of Execution

The model on which the tracing is based is known as the 'Byrd Box model of Execution'.

A program trace:

```
| ?- listing(parent).

parent(a, b).
parent(c, d).
```

```
no_cons([]).

no_cons([H|T]):-
        vowel(H),
        no_cons(T).

| ?- trace, no_cons([a,e,o]).
{The debugger will first creep -- showing everything (trace)}
   1  1  Call: no_cons([a,e,o]) ?
   2  2  Call: vowel(a) ?
   2  2  Exit: vowel(a) ?
   3  2  Call: no_cons([e,o]) ?
   4  3  Call: vowel(e) ?
   4  3  Exit: vowel(e) ?
   5  3  Call: no_cons([o]) ?
   6  4  Call: vowel(o) ?
   6  4  Exit: vowel(o) ?
   7  4  Call: no_cons([]) ?
   7  4  Exit: no_cons([]) ?
   5  3  Exit: no_cons([o]) ?
   3  2  Exit: no_cons([e,o]) ?
   1  1  Exit: no_cons([a,e,o]) ?

yes
{trace}
| ?-
```

Figure 30: Tracing the behaviour of the predicate no_cons

```
max([H|T], Answer):-
        max(T, H, Answer).

max([], Answer, Answer).
max([H|T], Temp, Answer):-
        H > Temp,
        max(T, H, Answer).
max([H|T], Temp, Answer):-
        max(T, Temp, Answer).

| ?- trace, max([7,3,9,2,6], Max).
{The debugger will first creep -- showing everything (trace)}
   1  1  Call: max([7,3,9,2,6],_112) ?
   2  2  Call: max([3,9,2,6],7,_112) ?
   3  3  Call: 3>7 ?
   3  3  Fail: 3>7 ?
   3  3  Call: max([9,2,6],7,_112) ?
   4  4  Call: 9>7 ?
   4  4  Exit: 9>7 ?
   5  4  Call: max([2,6],9,_112) ?
   6  5  Call: 2>9 ?
   6  5  Fail: 2>9 ?
   6  5  Call: max([6],9,_112) ?
   7  6  Call: 6>9 ?
   7  6  Fail: 6>9 ?
   7  6  Call: max([],9,_112) ?
   7  6  Exit: max([],9,9) ?
   6  5  Exit: max([6],9,9) ?
   5  4  Exit: max([2,6],9,9) ?
   3  3  Exit: max([9,2,6],7,9) ?
   2  2  Exit: max([3,9,2,6],7,9) ?
   1  1  Exit: max([7,3,9,2,6],9) ?

Max = 9 ?

yes
{trace}
| ?-
```

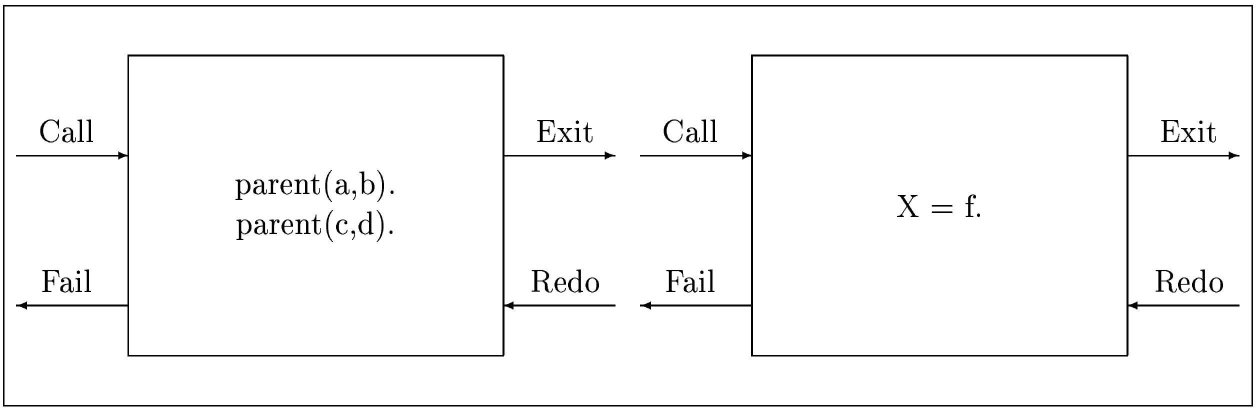
Figure 31: Tracing the behaviour of `max`

```
yes
| ?- trace, parent(X,Y), X = f.
{The debugger will first creep
-- showing everything (trace)}
   1  1  Call: parent(_44,_58) ?
   1  1  Exit: parent(a,b) ?
   2  1  Call: a=f ?
   2  1  Fail: a=f ?
   1  1  Redo: parent(a,b) ?
   1  1  Exit: parent(c,d) ?
   2  1  Call: c=f ?
   2  1  Fail: c=f ?
   1  1  Redo: parent(c,d) ?
   1  1  Fail: parent(_44,_58) ?

no
{trace}
| ?-
```

- Each box represents one invocation of a **procedure**.
- Use nested boxes for the bodies of rules.
- Each box has four ports:

  Call: used first time we look for a solution
  Exit: used if the procedure succeeds
  Redo: used when backtracking
  Fail: used if we can't satisfy the goal



Not all systems provide tracers that use all four ports, i.e. some tracers do not explicitly use the 'redo' port, but use 'call' again when backtracking.

11.3 Debugging using the Tracer

spy(predicate_name) Mark any clause with the given predicate_name as " spyable". Does not work for built-in predicates; can take a list of predicates as argument.

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.

Debugging options:

```
<cr>   creep              c       creep
 l     leap               s       skip
 r     retry              r <i>   retry i
 d     display            p       print
 w     write
 g     ancestors          g <n>   ancestors n
 n     nodebug            =       debugging
 +     spy this           -       nospy this
 a     abort              b       break
 @     command            u       unify
 <     reset printdepth   < <n>   set printdepth
 ^     reset subterm      ^ <n>   set subterm
 ?     help               h       help
```

Useful Debugging Options:

creep: This is the single stepping command. Use ⟨RETURN⟩ to **creep**. The tracer will move on to the next port.

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.

Tracing in SICStus Prolog, given the following program:

```
son(A, B) :-
        parent(B, A),
        male(A).

daughter(A, B) :-
```

```
        parent(B, A),
        female(A).

parent(A, B) :-
        father(A, B).
parent(A, B) :-
        mother(A, B).

male(tom).
male(jim).
male(cecil).
male(fred).

female(mary).
female(jane).
female(sue).

father(tom, jim).
father(tom, sue).
father(fred, cecil).

mother(mary, tom).
mother(mary, jane).
mother(mary, fred).

| ?- trace,daughter(X,Y).
{The debugger will first creep -- showing everything (trace)}
   1  1  Call: daughter(_51,_67) ?
   2  2  Call: parent(_67,_51) ? s
   2  2  Exit: parent(tom,jim) ?
   4  2  Call: female(jim) ? g
Ancestors:
   1  1  daughter(jim,tom)
   4  2  Call: female(jim) ?
   4  2  Fail: female(jim) ?
   2  2  Redo: parent(tom,jim) ? s
   2  2  Exit: parent(tom,sue) ?
   4  2  Call: female(sue) ? g
Ancestors:
   1  1  daughter(sue,tom)
   4  2  Call: female(sue) ? s
   4  2  Exit: female(sue) ?
   1  1  Exit: daughter(sue,tom) ?

X = sue,
Y = tom ?

yes
```

```
{trace}
| ?-
```

We have 'skipped' each time that we called `parent/2` so we do not see whether it is satisfied by using the `father/2` or `mother/2` rule: we only see the outcome of the subgoal. We have also asked to see the ancestors of the subgoal `female(jim)` i.e. what it was the immediate subgoal of.

We might choose to trace a particular predicate such as `father/2` by placing a spy point on it. Whenever we 'leap' in the tracing thereafter, the tracer will jump to the next call to this goal.

```
| ?- spy(father).
{Spypoint placed on user:father/2}
{The debugger will first leap -- showing spypoints (debug)}

yes
{debug}
| ?- daughter(X,Y).
 + 3  3  Call: father(_81,_65) ?
 + 3  3  Exit: father(tom,jim) ?
   2  2  Exit: parent(tom,jim) ?
   4  2  Call: female(jim) ? s
   4  2  Fail: female(jim) ?
   2  2  Redo: parent(tom,jim) ? l
 + 3  3  Redo: father(tom,jim) ?
 + 3  3  Exit: father(tom,sue) ?
   2  2  Exit: parent(tom,sue) ?
   4  2  Call: female(sue) ? l

X = sue,
Y = tom ?

yes
{debug}
```

11.4 Loading Files

Consult or [] can be used for loading files into the prolog interpreter. If the filename has punctuation characters in it, remember to enclose it in single quotes. More than one file may be consulted at once.

- ```
  | ?- consult(parents).
  {consulting parents...}
  {parents consulted, 10 msec 287 bytes}

  yes
  | ?-
  ```
  ```

- `| ?- [parents].`
- `| ?- consult('/u/ai/s2/ai2/aifoo/program').`
- `| ?- consult('foo.pl').`
- `| ?- consult([foo,baz,'foobaz.pl']).`
- `| ?- [foo,baz].`

Avoid splitting predicate definitions between files. For example, if you define part of your family program in one file and part in another (perhaps you decided to put different families in different files?) the following may happen:

```
| ?- consult(file1).
{consulting file1...}
{file1 consulted, 10 msec 287 bytes}

yes
| ?- consult(file2).
{consulting file2...}
The procedure parent/2 is being redefined.
    Old file: file1
    New file: file2
Do you really want to redefine it? (y, n, p, or ?) ?
       y    redefine this procedure
       n    don't redefine this procedure
       p    redefine this procedure and don't ask again
       ?    print this information

(y, n, p, or ?)
```

Files can also be consulted from inside another program file by writing either of the normal consult commands preceded by `'?-'` somewhere in the main file. Everytime the main program file is consulted it will also consult the other files. This saves you having to consult them all by hand.

11.5 Some common mistakes

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

** , | or ] expected in list **
member ( a , [ a , b , c
** here **
) .
Leaving off the closing list bracket causes a syntax error.
```

```
2. | ?- parent(a,b) X = f.

** variable follows expression **
parent ( a , b )
** here **
X = f .
Subgoals must be separated by commas.

3. | ?- parent(a b),X = f.

** atom follows expression **
parent ( a
** here **
b ) , X = f .
Arguments must also be seperated by commas.

4. | ?- parent(a, b, X = f.

** , or ) expected in arguments **
parent ( a , b , X = f
** here **
.
Closing predicate bracket omitted.

5. | ?-  parent (a,b).

** bracket follows expression **
parent
** here **
 ( a , b ) .
| ?-
Extra space between the predicate name and the opening bracket.
```

11.6 Summary

You should be able to:

- model the behaviour of Prolog programs using the Byrd Box Model
- be able to use the basic functionality of Prolog's debugging tools
- be able to identify common mistakes in typing Prolog queries

12 AI Applications of Prolog: State-Space Search

The rest of this document will sketch out a range of AI applications that Prolog is well suited for:

- State-space Search
- Definite Clause Grammars
- Morphological Processing
- Chat-Bot: Eliza
- Planning

These sections will place more emphasis on the practical aspects than the theory. The theory will be provided in the lectures. References will also be made to appropriate sections in the recommended text books. However, only material contained within this document, presented in the lectures, or used in the assignments is examinable.

12.1 The Missionaries and Cannibals Problem

Given the following ...

- Three missionaries and three cannibals are trying to cross a river: they want to get from the left bank to the right bank.
- There is one boat, which can take two people at most.
- If the cannibals outnumber the missionaries on a bank, then the missionaries get eaten. We'll assume that the boat being at a bank counts as it being on the bank.

... we want to find the schedule of crossings that will get all six across safely.

The Missionaries and Cannibals problem is a classic AI example of a **State-Space Search** problem.

- **States** are a simplified representation of the problem-world at a single point in time.
- The **State-Space** is the configuration of the possible states and how they connect to each other (i.e. the legal moves between states)

The state-space is best represented as a tree which grows down from the start state. Each branch on the tree denotes a transition from one state to another. To solve a state-space problem we need to search through the tree of possible states to find a path from our **start state** to the **goal state**.

From every problem state there may be multiple possible transitions, the number of transitions (know as the **branching factor**) increasing with problem complexity. It is the job of a **search strategy** to manage these possible transitions and choose the next one to pursue.

In the Missionaries and Cannibals problem there are a very small number of possible transitions so the search problem isn't too complex. However, even a small search problem like this can be made more efficient by correctly choosing the right search strategy.

For the theory behind search strategies see the lecture slides or Bratko Chapters 11 and 12. Further information on AI search strategies can be found in *Artificial Intelligence: A Modern Approach* by Russel and Norvig.

We'll now use the Missionaries and Cannibals problem to demonstrate the **top-down design** of a problem solution in Prolog.

12.2 Viewing the Problem as State Space Search

In abstract terms:

- We can represent the states as follows:

Initial State	MMMCCCB|—
Goal State	—|MMMCCCB

- We can represent the possible moves as follows:

M →	← M
MM →	← MM
MC →	← MC
CC →	← CC
C →	← C

A strategy for solving the problem, given a current state:

- Try each move in turn.
  - Check if it is a safe move to make.
  - If it is not, try another move.
- If there are no more moves to try, go back to where we had a choice of moves and try again.
- Once we've made a move, check to see if we are at the goal state; if not, repeat the process.

12.3 Practical 5: Missionaries and Cannibals in Prolog

(Read the beginning of this chapter before undertaking this practical.)

In this practical you will work through a solution to the Missionaries and Cannibals search problem. A partial version of the program needed to solve this problem is provided:

1. Download the file `mandc.pl` from the course website into your home directory.
2. Load mandc.pl in **Emacs**.
3. We will work through the code explaining each predicate.
4. If, at any time, you don't understand the explanation call over a demonstrator.
5. Your task will be to complete and modify certain sections.

12.3.1 A walk through the program

To implement a prolog program that solves the M&C problem we have to:

1. Choose a representation.
2. Decide on the algorithm to be used.
3. Choose predicates to represent each step.
4. Work out the detail in a top-down fashion.

Choosing a Representation: How will we represent the situation on each side of the river?

Some possibilities:

- Use a list of the entities that are on that side of the river:

```
[m,m,m,c,c,c,b]
```

- Use separate terms for each type of entity:

```
missionaries(3)
cannibals(3)
```

- Use one term with three arguments:

```
leftside(3,3,1)
```

- Use a list with three elements:

```
[3,3,1]
```

  where

  - missionaries = {0,1,2,3}
  - cannibals = {0,1,2,3}
  - boat = {0,1}

We'll use the last of these; so:

Initial state:	Left	=	`[3,3,1]`
	Right	=	`[0,0,0]`
Goal state:	Left	=	`[0,0,0]`
	Right	=	`[3,3,1]`

12.3.2 Problem Solution

The general idea is to make moves from one bank to the other until we reach the goal state.

To make a move from one bank to the other:

1. Make a move and get the new states of the banks.
2. Check if the new states of the banks are safe (i.e., make sure the missionaries won't get eaten); if they are not, repeat Step 1 for a different move.
3. Repeat the entire process until we reach the goal state.

We have to choose some predicates to represent these steps, then work top-down.

The Top Level Predicate: The main predicate `gofrom/2`:

- make a move and get the new states of the banks
- if `safe(Left)` and `safe(Right)` then `gofrom` the new states.

In Prolog:

```
gofrom(Left, Right):-
        applymove(Left,Right,NewLeft,NewRight),
        safe(NewLeft),
        safe(NewRight),
        gofrom(NewLeft, NewRight).
```

We need to check if the goal state has been reached:

```
gofrom([0,0,0],[3,3,1]).
```

Remember we have to put this before the recursive `gofrom/2` clause.

Applying a Move The predicate `applymove/4` gives us new states of the two banks by applying a move.

- If the boat is already on the left bank, the pattern when called will be:

  ```
  applymove([M1,C1,1], [M2,C2,0], ...)
  ```

- If the boat is on the right, the pattern will be:

  ```
  applymove([M1,C1,0], [M2,C2,1], ...)
  ```

To save having to do two sets of clauses, one for each side, we can check which side the boat is on then apply the same operator:

- If the boat is on the left:

```
applymove(Left, Right, '-->', Comment, NewLeft, NewRight):-
        boathere(Left),
        moveload(Left, Right, Comment, NewLeft, NewRight).
```

- If the boat is on the right:

```
applymove(Left, Right, '<--', Comment, NewLeft, NewRight):-
        boathere(Right),
        moveload(Right, Left, Comment, NewRight, NewLeft).
```

This uses the same `boathere/1` and `moveload/5`. `boathere/1` is defined very simply:

```
boathere([M,C,1]).
```

The arrow is a string which will be used later to denote in which direction the move occurs.

Moving a Load Now we need to define the predicate `moveload/5`.

The algorithm:

- select a move (`move/2`)
- check if it is possible (`possibletomove/2`)
- make the move (`performmove/5`)

So:

```
moveload(Source, Target, Comment, NewSource, NewTarget):-
        move(BoatLoad, Comment),
        possibletomove(Source, BoatLoad),
        performthemove(Source,Target,BoatLoad,NewSource,NewTarget).
```

Note that we use `Source` and `Target` since we may be moving from the left bank to the right, or from the right bank to the left.

Choosing a Move We have five possibilities for boatloads: one missionary, two missionaries, one missionary and one cannibal, two cannibals, or one cannibal.

How do we represent the moves?

- we could represent the possible moves as a list, and repeatedly select elements from this list; or
- we could represent each move as a separate clause and choose each clause in turn.

With the latter, we can use Prolog's backtracking: get a move, then if it fails, backtrack and get another move.

`move/2` is called with its arguments uninstantiated:

```
move(Boatload)
```

This matches to potential moves and instantiates `Boatload` in the process. For example:

```
move([1,1,1],'Move 1 Cannibal and 1 Missionary.').
```

means one missionary, one cannibal, and one boat. The second argument is a string describing the move. This matches the `Comment` variable in `moveload/5`.

You have to define the other moves yourself (write them here and in the designated part of mandc.pl:

Checking Whether a Move is Possible Defining `possibletomove/2`:

- Suppose the current state of the `Source` is `[2,0,1]`.
- Suppose the move selected is `move([1,1,1],'Move 1 Cannibal and 1 Missionary.')`.

This move is not possible, since there are no cannibals to move.

So, there must be, at the most, the number of missionaries, cannibals and boats in the move as in the `Source` state. Write the code for possibletomove/2 here:

We've selected a move and checked if it is possible. Now we have to perform the move.

Performing the Chosen Move We want to define `performthemove/5`.

What we have to do:

- we are given the current Source and Target banks
- we know the move to be made
- we want to find out the new states of the Source and Target banks after the move.

```
performthemove(Source, Target,
               Move, NewSource, NewTarget)
```

Define performthemore/5 here:

Safe States We still have to define `safe/1`, which determines whether a state is safe.

A state is safe provided there are not more cannibals than missionaries, or if there are no missionaries in that state (then it doesn't matter how many cannibals there are). So:

```
safe([2,3,1])
```

should be false, since there are two missionaries and three cannibals.

Define this predicate below. `safe/1` should succeed if the state is safe, and fail if it is not.

The Problem of Looping We don't want to waste effort visiting states already tried.

- How do we know if we've tried a state before? We keep a list of all the states visited.
- Do we need to remember the whole state? No: The left (or right) bank alone will do.

So:

1. Each time we make a move, add the old left side state to a list of previous left side states.
2. Before committing ourselves to the next move, after checking the resulting state is safe, we check if we are looping, i.e., if the new left state is a member of the list of previously visited left states. If it is not, then we are okay.
3. This check is done using the `member/2` predicate. `member` is negated using `\+` so that it will fail if the state is found to be in the list of previous states.

So the new `gofrom/3` predicate incorporating the looping check is:

```
gofrom(Left, Right, PreviousLeftStates):-
        applymove(Left,Right,Direction,Comment,NewLeft,NewRight),
        safe(NewLeft),
        safe(NewRight),
        \+member(NewLeft, PreviousLeftStates),
write(NewLeft), write(Direction),
write(NewRight), write(' '), write(Comment), nl,
        gofrom(NewLeft,NewRight,[NewLeft|PreviousLeftStates]).
```

The `write/1` commands use the Direction and Comment variables to construct a description of each chosen move. This will become clear once you run the program.

The code should now be complete and you will be able to get your code to generate a solution by typing `go`.

12.3.3 Improving the search strategy

Running the program will generate a solution to the problem. Do you think this is the only solution? Does the program generate any alternative solutions when you re-run it?

Can you think of how you would make it generate multiple answers?

The reason it only produces one solution is because it uses the same search strategy everytime you run it. Which search strategy does the program currently use? ________________________________

If you can't find the search strategy trace a run of the program using `trace.` Find the calls to `move/2`. These equate to the nodes on the search tree. Using these you can draw a search tree and see how the program traverses it. This should give you an idea of which search strategy it uses.

Can you think of how you would make the problem perform a different search strategy: depth-first, iterative deepening, breadth-first (depending on which one you thought it was already doing)?

[Optional] Try and implement a different search strategy. See if your new search strategy reaches a solution in less steps than before.

```
go:-
        gofrom([3,3,1], [0,0,0], [[3,3,1]]).

gofrom([0,0,0], [3,3,1],_).

gofrom(Left, Right, PreviousLeftStates):-
        applymove(Left,Right,Direction,Comment,NewLeft,NewRight),
        safe(NewLeft),
        safe(NewRight),
        \+member(NewLeft, PreviousLeftStates),
write(NewLeft), write(Direction),
write(NewRight), write(' '), write(Comment), nl,
        gofrom(NewLeft, NewRight,
               [NewLeft|PreviousLeftStates]).

applymove(Left, Right, '-->', Comment, NewLeft, NewRight):-
        boathere(Left),
        moveload(Left, Right, Comment, NewLeft, NewRight).

applymove(Left, Right, '<--', Comment, NewLeft, NewRight):-
        boathere(Right),
        moveload(Right, Left, Comment, NewRight, NewLeft).

boathere([_,_,1]).

moveload(Source, Target, Comment, NewSource, NewTarget):-
        move(BoatLoad,Comment),
        possibletomove(Source, BoatLoad),
        performthemove(Source, Target,
               BoatLoad, NewSource, NewTarget).

member(X,[X|_]).
member(X,[_|Z]):-member(X,Z).
```

Figure 32: The Code Provided

13 Parsing in Prolog

13.1 Introduction

In this section, 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.

We describe Prolog's inbuilt mechanism for encoding 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.

13.2 Simple English Syntax

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: so, *The ball runs fast* is syntactically correct while *The man goes pub* is not, since 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.

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

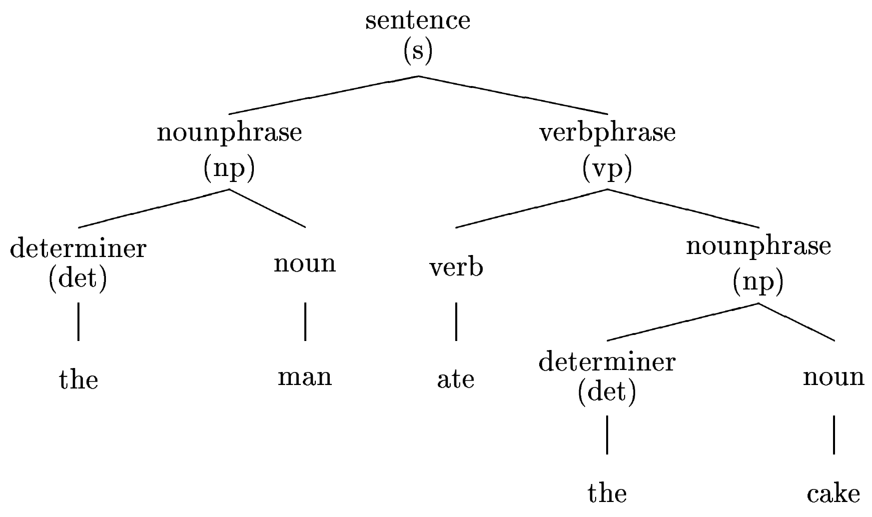


Figure 33: A parse tree

13.3 The Parse Tree

Figure 33 shows the parse tree for the sentence *the man ate the cake* with some common abbreviations in brackets.

We must remember that many sentences are ambiguous—i.e., they result in different parse trees.

13.4 Prolog Grammar Rules

In the earlier section, we saw the beginnings of a parser which used list processing techniques to build up a string of words corresponding to a sentence. In this section, we describe Prolog's inbuilt mechanism for building grammar rules.

Prolog, as a convenience, will do most of the tedious work for you. We can take advantage of a simplified representation that Prolog converts into normal Prolog syntax for us.

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 we won't pursue this here.

We could have interpreted these rules in English as follows:

```
Predicate noun_phrase/1
    noun_phrase(X) means that:  X is a sequence of words
    forming a noun_phrase
```

In which case, we could have used the predicate `append/3` to implement this:

```
noun_phrase(X):-
    append(Y,Z,X),
    det(Y),
    noun(Z).

det([the]).
noun([cat]).
noun([dog]).

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

?- noun_phrase([the,cat]).
yes
```

Here `append/3` is used in reverse to decompose a list rather than build it up. Once decomposed the individual elements of the list (sentence) can be checked to see if they are the correct parts of speech.

Instead, our definition for `noun_phrase/2` is:

```
    noun_phrase(X,Y) is true if:
        there is a noun phrase at the beginning of sequence X
        and the part of the sequence left after the nounphrase is Y.
```

In Prolog we could write this as:

```
noun_phrase(X,Y):-
       det(X,Z),
       noun(Z,Y).

det([the|S],S).
noun([dog|S],S).

?- noun_phrase([the,dog,bit,the,cat],[bit,the,cat]).
yes
```

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).
```

`'C'/3` is a built in predicate that is used by other built in predicates.It is rarely used by programmers as the same effect can be achieved using head and tail decomposition in the head of a clause. Hence the obscure predicate name.

13.5 Using the Grammar Rules

Set a goal of the form

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

and the Prolog interpreter will check the sentence against the grammar you have defined and tell you if it belongs to the language. A system like this is known as a *recogniser*. It is of limited usefulness.

13.6 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* and we refer to its name 'foo' as a *functor*.

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([...
```

13.7 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., '{' and '}'.

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 "=<".

Further details on the use of Definite Clause Grammars will be provided in the lectures. If you want to learn more about the use of DCGs in Prolog (which is outside the scope of this course) see Chapter 9 in Clocksin and Mellish or Chapter 21 in Bratko.

13.8 Practical 6: Definite Clause Grammars

13.8.1 Introduction

This practical is intended to give you some experience in writing Definite Clause Grammars. You are asked to extend a basic grammar in various ways.

13.8.2 The Basic Grammar

1. Download the file `grammar1.pl` from the course website, the contents of which are shown in Figure 34, to your area. Get into Prolog and consult your copy of the file.

   Note here that the symbols `vt` and `vi` are being used to mean **transitive verb** and **intransitive verb** respectively. A transitive verbs is one which takes an object noun phrase; an intransitive verb is one which does not take an object noun phrase.

2. You might find it interesting to do a `listing` of the grammar; you will see from this that Prolog adds extra arguments in the conversion from the definite clause grammar formalism to normal Prolog clauses.

```
| ?- [grammar1].
{consulting grammar1.pl...}
{grammar consulted, 50 msec 2650 bytes}

yes
| ?- listing.

n(A, B) :-
        'C'(A, dog, B).
n(A, B) :-
        'C'(A, cat, B).

s(A, B) :-
        np(A, C),
        vp(C, B).

...

yes
| ?-
```

3. To test the grammar, you have to provide calls to `s` with two arguments, since you are calling the translated form. The first argument is the list of words whose sentencehood you want to check, and the second argument is the empty list:

```
| ?-  s([a,cat,chases,a,dog],[]).

yes
| ?-
```

```
s --> np,vp.

np --> det, n.
np --> pn.

vp --> vt, np.
vp --> vi.

det --> [every].
det --> [a].
det --> [the].

vt --> [chases].
vi --> [miaows].

n --> [dog].
n --> [cat].

pn --> [fido].
pn --> [tigger].
```

Figure 34: A simple Definite Clause Grammar

Note that you can also use this grammar to randomly generate sentences by providing a variable as the first argument to `s`:

```
| ?- s(Sentence, []).

Sentence = [every,dog,chases,every,dog] ? ;

Sentence = [every,dog,chases,every,cat] ? ;

Sentence = [every,dog,chases,a,dog] ? ;

...
```

Test the grammar with the following sentences and make sure you understand its behaviour; before trying each sentence, try to work out what Prolog will do and why.

the dogs miaow
the cat miaows
fido chases a cat
every dog chases
tigger miaows a dog
a tigger miaows

Try other random sentences.

13.8.3 Structure Building

A string of words is in the language defined by a grammar if a tree corresponding to the structure of the string can be built. Parsing a sentence in Prolog consists in building such a tree implicitly. We can make this tree explicit using another facility provided by the DCG notation: grammar symbols may be given arguments, in exactly the same way as Prolog goals. To build the tree, we associate with each non-terminal symbol an argument which represents its structure, as in the following example:

```
s([s,[NP,VP]]) --> np(NP), vp(VP).
```

For this exercise, you should augment the grammar given in Figure 34 with structure-building arguments. You should achieve the following result.

```
| ?- s(ParseTree,[tigger,miaows],[]).

ParseTree = [s,[[np,[[pn,[tigger]]]],[vp,[[verb,[miaows]]]]]] ? ;

no
| ?-
```

So, for each node n in the tree, the result should contain a list whose first element is the name of the node, and whose second element is a list of elements that correspond to the nodes that are daughters of n. Each terminal node should be represented by the word that lies at that terminal node.

Note that, since we are adding an additional arguement to each predicate, calls to `s` must now contain this new argument; it appears *before* the arguments required by the DCG translation process.

Make the alterations to `grammar1.pl`. Perform a few tests and then show it to the demonstrator.

Also notice that, in the example above, we have typed a `;` to see if Prolog can offer any more parses. When more than one parse is available, we say that the string is **syntactically ambiguous**.

Will this grammar accept any syntactically ambiguous strings? Why?

13.8.4 Adding Number Agreement

Above, we saw how an extra argument was added to the non-terminal symbols of a grammar to build a syntactic structure. Any number of arguments may be added in this way. The DCG translator just passes through the argument structure and adds the two string handling arguments at the end.

This aspect of DCGs can be used to improve the coverage of a grammar. For instance, we can add an argument whose values range over `{singular, plural}` to represent the feature **number**. By introducing the appropriate values for this feature in the lexicon and then percolating them up and identifying the values on say, subject and verb phrase, we can guarantee number agreement. Here's an example:

```
n([n,[dog]],singular) --> [dog].
n([n,[dogs]],plural) --> [dogs].
np([np,[DET,N]],Num) --> det(DET,Num), n(N,Num).
s([s,[NP,VP]]) --> np(NP,Num),vp(VP,Num).
```

For this part of the exercise, you should augment all of the appropriate rules in the grammar to take account of number agreement information, so that your grammar should **rule out** sentences like the following:

1. the dogs miaows
2. the cat miaow
3. fido chase a cat
4. every dogs chases a cat
5. tigger chases a dogs

To make this work, you will have to attend to the following:

- Note that, in English, the subject noun phrase and the verb phrase in a sentence must agree in number; similarly, in a noun phrase, the determiner and noun must agree in number. However, the number of the object noun phrase in a sentence is not relevant to the grammaticality of the sentence as a whole.
- Each **lexical rule** (the rules that have words as their right hand sides) will need to be augmented with number information, which will then percolate up via unification to higher level nodes in the trees.

Make the alterations to `grammar1.pl`. Test that the example sentences above are rejected by your new code and then show it to the demonstrator.

13.8.5 Extending the Coverage of the Grammar

You should now try to extend the coverage of the grammar to include the following phenomena. In each case you will have to add at least one new grammar rule and some lexical rules.

Relative clauses as in *a cat that chases tigger miaows.* You should incorporate number agreement arguments to rule out sentences like **a cat that chase tigger miaows.*

Note:

- The relative clause (*that chases tigger*) modifies the noun within the initial Noun Phrase (*a cat*) so it should be placed within the NP grammar rule.
- A relative clause is quite a complex grammatical structure but for this practical just limit yourself to generating the example sentence. Look at the example sentence and see which rules and extra lexical entries (e.g. 'that' is a pronoun) you will need to generate it.

Write your added entries here:

Prepositional phrases Now try adding rules for prepositional phrases such as

A cat with a hangup chases Tigger.
A cat with a hangover miaows in discomfort.

Note here that prepositional phrases may appear to be attached to either noun phrases, as in the first case, or to verb phrases, as in the second case.

Your grammar should provide two parses for the sentence *a cat chases the dog with a mouse-trap*.

Ultimately your grammar should build structure for these syntactic constructions, but you may find it easier in the first instance to add the additional grammar rules required to the version of the grammar that doesn't build structure.

Write your added entries here:

13.8.6 Optional extension of DCG

If you still have time in the tutorial try and think of other ways you can extend the scope of your DCG so thatit can recognise and generate more sentences. Try adding more words to the lexicon and getting your program to generate sentences. If some of them appear grammatically incorrect think of why they are incorrect and see if you can fix them.

14 Input/Output

This section will describe the built in predicates that allow us to pass data into and get data out of a running Prolog program. This data can be in the form of single lines of input/output or whole files. Rudimentary processing of that data is also discussed.

14.1 Basic input/output facilities

Prolog provides the system predicates `read`ing and `write`ing for reading and writing atoms:

```
| ?- read(X).
|: 2.

X = 2 ?

yes
| ?- read(X).
|: 'This is just a few words between single quotes'.

X = This is just a few words between single quotes ?

yes
| ?- write(foo).
foo
yes
| ?- write('Where *did* you buy that jacket?').
Where *did* you buy that jacket?
yes
| ?-
```

We might start with a basic program that computes the cube of a number:

```
cube(N,C) :-
        C is N * N * N.
```

Using the program:

```
| ?- cube(1, X).

X = 1 ?

yes
| ?- cube(5, Y).

Y = 125 ?
```

```
yes
| ?- cube(12, Z).

Z = 1728 ?

yes
| ?-
```

We can them make modification so that the program will read the numbers itself:

```
cube :-
        read(X),
        process(X).

process(stop):-  !.

process(N) :-
        C is N * N * N,
        write(C), ttyflush,
        cube.


| ?- cube.
|: 2.
8
|: 5.
125
|: 1728.
864813056
|: stop.

yes
| ?-
```

(n.b. `ttyflush/0` is a built in predicate that ensures that the prolog prompt appears on the screen)

We might think that this could be simplified:

```
cube :-
        read(stop).
cube :-
        read(N),
        C is N * N * N,
        write(C), ttyflush,
        cube.
```

However, we get the following behaviour:

```
| ?- cube.
|: 34.
|: 2.
8
|: stop.
```

What happens is that `read` cannot be redone, so when the first `read` in the first `cube` rule is called, the number '34' is input. It failes to match to 'stop', so this rule fails and the next `cube` rule is tried. This calls `read` and waits for a further input, '2', which it then 'cubes' and writes out: the first value input has been lost. We can illustrate this by tracing the program.

```
| ?- trace, cube.
   1  1  Call: cube ?
   2  2  Call: read(stop) ?
|: 34.
   2  2  Fail: read(stop) ?
   2  2  Call: read(_165) ?
|: 2.
   2  2  Exit: read(2) ?
   3  2  Call: _170 is 2*2*2 ?
   3  2  Exit: 8 is 2*2*2 ?
   4  2  Call: write(8) ?
8    4  2  Exit: write(8) ?
   5  2  Call: ttyflush ?
   5  2  Exit: ttyflush ?
   6  2  Call: cube ?
   7  3  Call: read(stop) ?
|: stop.
   7  3  Exit: read(stop) ?
   6  2  Exit: cube ?
   1  1  Exit: cube ?

yes
```

- `get(X)` unifies `X` with next non blank printable character (in ASCII code) from current input stream
- `get0(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

For example:

```
| ?- get0(C).
|:    <--nothing input; identifies new line command
```

```
C = 10 ?

yes
| ?- get0(C).
|: f

C = 102 ?
yes
| ?-  get0(C).
|:     <-- Space

C = 32 ?
yes
| ?-

| ?- get(C).
|: <-- Return pressed; can't match so tries again.
|:  <-- Return pressed; can't match so tries again.
|:   <-- Return pressed; can't match so tries again.
|: .    <-- identifies full-stop

C = 46 ?
yes
| ?- get(C).
|: y

C = 121 ?
yes
| ?- put(87).
W
yes
| ?- put(32).

yes
| ?- put(10).


yes
| ?-
```

If you want to know which characters have which ASCII characters use this on-line ASCII table: `http://www.asciitable.com/`.

14.2 File input/output

A wee program:

```
double(X, Y):-
        Y is 2*X.

test:-
        read(X),
        double(X,Y),
        write(Y),
        nl.


| ?- test.
|: 2.
4
yes
| ?-
```

We might want to put this into a test loop, for testing a number of values.

```
double(X, Y):-
        Y is 2*X.

test:-
        read(X),
        \+(X = -1),
        double(X,Y),
        write(Y),
        nl,
        test.

test.
```

Rather than keep having to type in the test values, we might want to store then in a file, and read them into the program each time an input is called for.

```
go:-
        see(datafile),
        test,
        seen,
        write('Okay, all done.').

double(X, Y):-
        Y is 2*X.

test:-
        read(X),
        \+(X = -1),
        double(X,Y),
```

```
          write(Y),
          nl,
          test.

test.
```

So if the contents of the file were:

```
3.
54.
-1.
```

and we ran the program, we would get:

```
| ?- go.
6
108
Okay, all done.
yes
| ?-
```

The useful predicates here for reading in from a file are:

- `see/1`
- `seen/0`
- `seeing/1`

where `see/1` specifies which file to read from ('datafile' in the above example); `seen/0` closes this file and returns to the default input from the terminal, and `seeing/1` enables the user to find out where the input is coming from at any time.

We might also want to consider writing the output from our testing to a file:

```
go:-
          tell(resultsfile),
          see(datafile),
          test,
          seen,
          told,
          write('Okay, all done.').

double(X, Y):-
          Y is 2*X.

test:-
          read(X),
```

```
        \+(X = -1),
        double(X,Y),
        write(Y),
        nl,
        test.

test.
```

The system predicates provided here are:

- `tell/1`
- `told/0`
- `telling/1`

where `tell/1` specifies the file to write to; `told/0` closes this file and returns to the default keyboard output, and `telling/1` allows the user to query where the output is being sent to at any time.

Once an input file is identified with `see/1` and an output file identified with `tell/1` `read/1` and `write/1` commands read from and write to these datafiles rather than the terminal window. These commands should be used carefully as this kind of casual manipulation of datafiles can lead to confused program behaviour.

Instead of defining an arbitrary character to indicate when we have read all the values we need from a file, we can use the pre-defined `end_of_file` marker.

```
go:-
        tell(resultsfile),
        see(datafile),
        test,
        seen,
        told,
        write('Okay, all done.').

double(X, Y):-
        Y is 2*X.

test:-
        read(X),
        \+(X = end_of_file),
        double(X,Y),
        write(Y),
        nl,
        test.

test.
```

14.3 Translating atoms and strings

The `name` predicate defines the relation that holds between an atom and the list of characters that make it up.

```
| ?- name(cat,Y).

Y = [99,97,116] ?

yes
| ?- name(X,[98,101,114,116]).

X = bert ?

yes
| ?-
```

At least one of `name`'s arguments must be instantiated. It can be used to avoid ASCII codes:

```
| ?- [98,101,114,116] = "bert".

yes
| ?- [X] ="a".

X = 97 ?

yes
| ?-
```

This will be used in the later section on Morphology.

14.4 Practical 7: Input/Output

14.4.1 Introduction

This practical is intended to give you some experience in using Prolog's built-in input/output predicates. From the notes above you should have some understanding of the operation of each of the following predicates:

Predicate	Behaviour
`read/1`	read a term from the current input stream
`write/1`	write a term to the current output stream
`nl/0`	write a newline to the current output stream
`tab/1`	write a specified number of spaces to the current output stream
`put/1`	write a specified ASCII character to the current output stream
`get/1`	read a printable ASCII character from the current input stream
`get0/1`	read an ASCII character from the current input stream
`see/1`	make the specified file be the current input stream
`seeing/1`	determine the current input stream
`seen/0`	close the current input stream and reset it to `user`
`ttyflush/0`	flush the output buffer
`tell/1`	make the specified file be the current output stream
`telling/1`	determine the current output stream
`told/0`	close the current output stream and reset it to `user`
`name/2`	arg 1 (an atom) is made of the ASCII characters listed in arg 2

In this practical, you will use some of these predicates to write a number of programs that make use of input and output. More information on each of these predicates can be found in the SICStus manual.

This practical contains a lot of material. You are not expected to finish it but see how far you can get. You do not need to write your answers on the worksheet and show them to the demonstrator (although writing out code on paper before typing it into the computer is always advisable). You will be asked to incrementally develop five predicates: `echo, limit, numberfile, and findnumber`. When you have a final version of each predicate (you will continually add to them throughout the practical) show it to the demonstrator who will log it as complete. If you can't complete a predicate then call the demonstrator over and show them your problems.

14.4.2 Basic Terminal Input/Output

1. Write a predicate `echo/0` that reads a number from the terminal and writes it out again. Your program should keep going until it reads the number `0`. So, for example, you should see behaviour like the following:

```
| ?- echo.
```

```
|: 3.
3
|: 4.
4
|: 5.
5
|: 1278.
1278
|: 0.

yes
| ?-
```

Note that each number typed in must be terminated by a full stop.

2. Modify this program so that it prints a prompt for input, and prints a response message, along the following lines:

```
| ?- echo.
Give me a number:  4.
Your number was 4.
Give me a number:  129.
Your number was 129.
Give me a number:  0.
Okay, see you later.
yes
| ?-
```

   Note that you will have to use `ttyflush` to print the prompt in the way it is done here; alternatively, you might look at what the manual has to say about the built-in predicate `prompt/2`.

3. Write a predicate `limit/1` which takes a number as argument, and then reads numbers typed at the terminal, for each number saying whether it is greater than, equal to, or less than the number provided as argument. The program should stop when it reads a `0`. The program's behaviour should be something like the following:

```
| ?- limit(41).
Give me a number: 28.
Your number was below the limit.
Give me a number: 41.
Your number was the same as the limit.
Give me a number: 71891.
Your number was above the limit.
Give me a number: 0.
Okay, see you later.
yes
| ?-
```

14.4.3 File Input/Output

1. Write a predicate called `numberfile/1` that takes as argument the name of a file. The named file should be one you have already constructed, containing a number terminated by a full stop on each line, like the following:

   ```
   1.
   1898.
   34.
   1891238912398.
   12.
   18.
   0.
   ```

   The last number in the file should be a `0`; this is how you will terminate the reading process.

   The predicate should open the file and read numbers from the file one at a time, writing them out to the screen, until it reads the `0`. It should then close this input stream.

2. Augment the program above so that it will write out *any* number it finds, including `0`. To do this, you should modify the program so that it terminates on reaching the `end_of_file` marker.

3. Write a predicate `findnumber/2` that takes as arguments a number and the name of a file. The predicate should open the file and read numbers from the file one at a time, doing nothing until it finds a number that matches the first argument; it should then announce this fact and continue with the rest of the file.

4. Modify the `echo` predicate you defined earlier so that it now takes two arguments, the first the name of an input file and the second the name of an output file.

   The predicate should open the input file and read numbers from the file one at a time, writing them out to the output file. Use the `end_of_file` marker to terminate the process, at which point you should make sure you reset both the input and output streams to `user`.

5. Modify `echo` again, this time to read the names of the input and output files from the terminal. You should print appropriate prompts in each case.

6. Modify the predicate `limit` you defined earlier, to read input from a file and to write the numbers out to different places as follows:

   - if the number is the same as the limit, announce this on the terminal;
   - if the number is less than the limit, write it to a file called `lower`; and
   - if the number is greater than the limit, write it to a file called `higher`.

   Once done, check the contents of the two files to make sure your program did what it should have done.

Remember to show each completed predicate to the demonstrator so that they have a record of your work.

15 Morphology: A List Processing Application

Most natural languages show a regularity in the way words decompose into units of meaning: for example, in English most plurals are formed by adding the letter *s*. This suggests a possible economy for NLP systems: instead of storing every form (singular, plural, etc.) of each word, we can store just the base form plus some rules for building the derived forms. Question: how would we do this in Prolog?

We are going to look at:

- Turning atoms into strings and back again.
- Using `append` to concatenate strings.
- Adding exceptions to general rules.

Just to remind you about lists:

`[1]`	=	`[1|[]]`
`[1, 2]`	=	`[1|[2]]`
`[1, 2, 3]`	=	`[1|[2,3]]`
	=	`[1|[2|[3]]]`
	=	`[1,2|[3]]`

and about using the predicate `append`:

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

Given the goal:

```
?- append([a,b], [c,d], L).
```

the following trace results:

```
| ?- trace, append([a,b],[c,d],L).
{The debugger will first creep
-- showing everything (trace)}
   1  1  Call: append([a,b],[c,d],_84) ?
   2  2  Call: append([b],[c,d],_238) ?
   3  3  Call: append([],[c,d],_345) ?
   3  3  Exit: append([],[c,d],[c,d]) ?
   2  2  Exit: append([b],[c,d],[b,c,d]) ?
   1  1  Exit: append([a,b],[c,d],[a,b,c,d]) ?

L = [a,b,c,d] ?

yes
{trace}
| ?-
```

So now we have `name` (see previous chapter) which gives us a way of converting an atom into a list of characters, and `append` which gives us a way of concatenating lists. So, we have a way of concatenating atoms:

```
| ?- name(black,L1), name(bird,L2),
     append(L1,L2,L3), name(Word,L3).

L1 = [98,108,97,99,107],
L2 = [98,105,114,100],
L3 = [98,108,97,99,107,98,105,114,100],
Word = blackbird ?

yes
| ?-
```

If we then think about the simple plural rule in English:

> To make the plural form of a singular noun, add an *s*.

for example:

Singular Noun	Plural Noun
terminal	terminals
tree	trees
cube	cubes

How can we perform this mapping in Prolog? We could define a predicate

```
plural(SingularNoun, PluralNoun)
```

with the following behaviour:

```
| ?- plural(cube, Plural).

Plural = cubes ?

yes
| ?- plural(keyboard, Plural).
Plural = keyboards ?

yes
| ?-
```

How do we do this?

```
% plural(Sing, Plu)
```

```
plural(Sing, Plu):-
        name(Sing, SingChs),
        name(s, EndChs),
        append(SingChs, EndChs, PluChs),
        name(Plu, PluChs).
```

Tracing an example:

```
| ?- trace, plural(cube, Plural).
Call: plural(cube,_58) ?
Call: name(cube,_210) ?
Exit: name(cube,[99,117,98,101]) ?
Call: name(s,_216) ?
Exit: name(s,[115]) ?
Call: append([99,117,98,101],[115],_223) ?
Call: append([117,98,101],[115],_706) ?
Call: append([98,101],[115],_814) ?
Call: append([101],[115],_921) ?
Call: append([],[115],_1027) ?
Exit: append([],[115],[115]) ?
Exit: append([101],[115],[101,115]) ?
Exit: append([98,101],[115],[98,101,115]) ?
Exit: append([117,98,101],[115],[117,98,101,115]) ?
Exit: append([99,117,98,101],[115],[99,117,98,101,115]) ?
Call: name(_58,[99,117,98,101,115]) ?
Exit: name(cubes,[99,117,98,101,115]) ?
Exit: plural(cube,cubes) ?

Plural = cubes ?

yes
```

There are other morphological transformations in English: for example, we generally add *ed* to get the past tense of verbs:

Present Tense	Past Tense
collect	collected
screen	screened
attend	attended

We can build a more general procedure

```
% generate_morph(BaseForm, Suffix, DerivedForm)

generate_morph(BaseForm, Suffix, DerivedForm):-
        name(BaseForm, BaseFormChs),
        name(Suffix, SuffixChs),
```

```
        append(BaseFormChs, SuffixChs, DerivedFormChs),
        name(DerivedForm, DerivedFormChs).
```

A problem: there are exceptions to the morphological rules we have seen.

For example:

- To make the plural of a word like *knife*, we not only add an *s* but we change the *f* to a *v*.
- To make the past tense of a word like *create*, we add only a *d*, instead of *ed*.

How do we deal with this? First, we define a predicate `morph/3`: this is just `append/3` under another name.

```
morph([], Suffix, Suffix).
morph([H|T], Suffix, [H|L2]):-
        morph(T, Suffix, L2).
```

We call this inside `generate_morph`:

```
% generate_morph(BaseForm, Suffix, DerivedForm)

generate_morph(BaseForm, Suffix, DerivedForm):-
        name(BaseForm, BaseFormChs),
        name(Suffix, SuffixChs),
        morph(BaseFormChs, SuffixChs, DerivedFormChs),
        name(DerivedForm, DerivedFormChs).
```

A few other things you should remember about strings:

```
| ?- X = "fe".

X = [102,101] ?
yes
| ?- X = "s".

X = [115] ?
yes
| ?- X = "ves".

X = [118,101,115] ?
yes
| ?- X = "e".
```

```
X = [101] ?
yes
| ?- X = "ed".

X = [101,100] ?
yes
| ?-
```

Note that the exceptions we indicated are also **rules**: there are classes of words that have this behaviour (in other words, they are *not* irregular forms).

So, we can add special base cases for these more specific rules:

```
morph([102,101], [115], [118,101,115]).
morph([101], [101,100], [101,100]).
morph([], Suffix, Suffix).

morph([H|T], Suffix, [H|L2]):-
        morph(T, Suffix, L2).
```

But we don't need to use ASCII codes explicitly:

```
morph("fe", "s", "ves").
morph("e", "ed", "ed").
morph([], Suffix, Suffix).

morph([H|T], Suffix, [H|L2]):-
        morph(T, Suffix, L2).
```

15.1 Summary

- Understand how to use `name` to turn atoms into strings and back again.
- Understand how to use `append` to concatenate strings.
- Understand how to add exceptions to general rules.

15.2 Prolog Practical 8: Morphology

15.2.1 Introduction

This practical is intended to give you some practice in using list processing to perform morphological manipulations of strings. The practical is built around the use of the built-in predicate `name/2` and the `generate_morph/3` predicate explained above.

Write your prolog code in the spaces provided and show them to the demonstrator.

15.2.2 Basic Operations on Atoms and Strings

To do these problems, you will need to make use of the built-in predicate `name/2`.

1. Define a predicate `starts/2` that takes as arguments an atom and a character, and checks whether the atom starts with the character. So, your predicate should behave in the following manner:

   ```
   | ?- starts(foo,f).

   yes
   | ?- starts(baz,g).

   no
   | ?-
   ```

   Write your code here:

   ```
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ```

2. Define a predicate called `plural/2` which will convert nouns into their plural forms: for example

   ```
   | ?- plural(cake,X).

   X = cakes
   yes
   | ?- plural(xylophone, X).

   X = xylophones
   yes
   | ?-
   ```

3. Modify `plural` so that it reads words from a file and writes the output to the screen.

Write your code here:

```
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
```

15.2.3 Morphological Processing

Figure **??** shows the basic elements required for morphological processing.

This is used as follows:

```
| ?- generate_morph(knife,s,W).

W = knives ?

yes
| ?- generate_morph(terminal,s,W).

W = terminals ?

yes

| ?- generate_morph(attempt,ed,W).

W = attempted ?

yes
| ?- generate_morph(inundate,ed,W).

W = inundated ?

yes
| ?-
```

For this part of the practical, you have to extend this program in various ways.

1. First, make sure you understand thoroughly how the program works. To do this you should type it in and trace its behaviour.

2. The program as it stands includes, effectively, a rule that pluralises words ending in *-fe* by changing the *f* to a *v* before adding the *s*. There are other rules required for other plurals. Add additional clauses to obtain the following behaviour from your program:

```
| ?- generate_morph(box,s,W).

W = boxes ?

yes
| ?- generate_morph(fox,s,W).

W = foxes ?

yes
```

Write your code here:

```
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
```

```
| ?- generate_morph(spy,s,W).

W = spies ?

yes
| ?- generate_morph(fly,s,W).

W = flies ?

yes
| ?-
```

Write your code here:

```
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
```

3. Think about how would you extend the program above so it *also* deals with the following case:

```
| ?- generate_morph(ox,s,W).

W = oxen ?

yes
| ?-
```

You don't have to implement this: the objective is to understand the nature of the problem.

4. Finally, add clauses to your program so that it produces the following behaviour:

```
| ?- generate_morph(slow,ly,W).

W = slowly ?
```

```
yes
| ?- generate_morph(full,ly,W).

W = fully ?

yes
| ?-
```

Write your code here:

```
------------------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------
```

15.2.4 Implementing Eliza in Prolog

ELIZA was the very first Chat-Bot. It was developed by Weizenbaum in 1966 to simulate (or rather parody) conversation with a human typing input at a terminal. It works by recognizing simple word patterns and replying by using preprogrammed response patterns. This allows it to imply depth and understanding without actually having any. So much so that early users of ELIZA believed it to be the first step to passing the Turing Test (the gold standard for AI). This claim was passionately dismissed by Weizenbaum (for reasons that I am sure will become apparent).

ELIZA uses similar pattern matching techniques to those you have been developing in this practical. Your task is to examine a rudimentary version of ELIZA and try to improve it.

Download `eliza.pl` from the course website and examine the code. A large portion of the code is concerned with Input/Output. The interesting parts of the code are printed below.

The general algorithm:

- Read a sentence.
- Perform some transformation on the sentence to produce a response.
- Write out the response.

A Rudimentary Eliza

```
go:-
        read_in(Input),
        process(Input).

process([W1,W2]):-
        member(W1, [bye, goodbye, quit, halt, chow]),
        write('I hope I was of some help.'), nl.
```

```
process(Input):-
        build_response(Input, Output),
        reply(Output), nl,
        go.

build_response(Input, Output):-
        rule(Input, Output).

rule([Greeting|Rest], [hi|Rest]):-
        member(Greeting, [hi, hello]).
rule([i,hate,X,'.'],
     [do, you, really, hate, X, ?]).
rule([X,you,Y,me,'?'],
     [why, do, you, ask, if, 'I', Y, you, ?]).
rule([i,like,X,'.'],
     [does, anyone, else, in, your, family, like, X, ?]).
rule([are, you, X,'?'],
     [what, makes, you, think, 'I', am, X, ?]).
rule([you, are, X, '.'],
     [does, it, please, you, to, believe, 'I', am, X, ?]).
rule(X,
     [please, go, on, '.']).
```

A Consultation

```
| ?- go.
|: Hello there.
Hi there.

|: How are you?
Please go on.

|: Can you help me?
Why do you ask if I help you?

|: Because you're supposed to.
Please go on.

|: Don't you like me?
Why do you ask if I like you?

|: You're not being very helpful.
Please go on.

|: Are you deaf?
What makes you think I am deaf?
```

```
|: You don't help much.
Please go on.

|: Oh, I give up.
Please go on.

|: Bye.
I hope I was of some help.
```

Improving ELIZA The current version of ELIZA is very limited in the patterns it can recognize. Examine the code carefully. `read_in/1` takes user input from the command prompt and turns it into a list of words. These are then sent to `rule/2` which pattern matches the input and produces a response. Make sure you understand how this works.

1. The range of responses ELIZA can produce is limited by the number of rules it has. Try writing some new clauses for `rule/2`.

   - Make sure you respect the ordering of clauses that are already in the program. The default rule (the one with a variable in place of the input) must always be at the bottom.
   - Your rules do not have to match the entie input. You can use `[H|T]` to split the input and process specific parts of it.
   - The first clause of `rule/2` already in the code is actually a rule rather than a fact. Can you think of any other patterns that might need a similar degree of processing?

   Write samples of your new rules along with example consultations here:

   ```
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ------------------------------------------------------------
   ```

2. When generating a response ELIZA currently outputs sentences starting with a lower-case letter. This is because any word beginning with an upper-case letter stored in the response lists would be identified as a variable. One solution to this would be to use strings instead of constants in the response lists (i.e. enclose all words in single quotes: `'What'`). Another more elegant way would be to change the first letter of the response to a capital before writing it to the terminal.

   - Look at `eliza.pl`. Beneath `reply/1` there is another version commented out. This version contains a command `capitalize/2` which takes a word starting with a lower-case letter and converts the first character into upper-case. See if you can define this predicate.

- For an idea of how to change letter case look at the ASCII table (`www.asciitable.com`). Lower-case letters have an ASCII number 32 points higher than their upper-case equivalents.
-

Write `capitalize/2` along with example consultations here:

```
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
```

3. Currently the pattern matching rules have to handle agreement of pronouns. For example, if you were to ask the question *Why do you hate me?* the *'you'* only gets transformed to *'me'* in the response because this transformation is explicitly represented in the response pattern. Write a predicate `you2me/2` which takes a list of words and converts all instances of *me* and *'I'* to *you*.

Write the predicate here:

```
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
----------------------------------------------------------
```

[This section is optional] Now try and integrate it into ELIZA. Where do you think would be the best place to put it, before or after pattern matching? Will you have to change any clauses of `rule/2`? What do the new reponse patterns look like? Is it better or worse than before?

16 Planning in Prolog: The Monkey and the Bananas

Here we take a standard example AI problem: modelling problem-solving behaviour. We look at representing the world as symbolic descriptions, using predicates and arguments for relations and objects and also for actions. We use operators to represent actions in the world—a good approach where the search space is potentially infinite, but a finite set of potential actions and outcomes can be identified.

16.1 The Problem

> A hungry monkey is in a cage. Suspended from the roof, just out of his reach, is a bunch of bananas. In the corner of the cage is a box. After several unsuccessful attempts to reach the bananas, the monkey walks to the box, pushes it under the bananas, climbs on to it, picks the bananas and eats them.

How do we write a program that can build a plan which, if executed, would model the behaviour of the monkey?

16.2 The General Approach to a Solution

We express the actions that can be performed as **operators** which act on the world and change its state.

For example:

- the monkey can move objects;
- the monkey can move from place to place;
- the monkey can stand on the floor or climb on the box;
- the monkey can grab bananas.

Operators: the basic techniques we'll use come from an early AI planning language called STRIPS.

- Each operator has outcomes: it affects the state of the world.
- Each operator can only be applied in certain circumstances: these circumstances are the **preconditions** of the operator.

16.3 Representational Considerations

An informal solution would be:

The monkey pushes the box under the bananas, climbs on it and grabs the bananas.

Questions:

- How do we represent the state of the world?
- How do we represent operators?
- Does our representation make it easy to:
  - check preconditions;
  - alter the state of the world after performing actions; and
  - recognise the goal state?

Representing the World: we have:

- a monkey, a box, the bananas, a floor;
- places in the room—call them a, b, and c;
- relations of objects to places:
  - the monkey being at location a;
  - the monkey being on the floor;
  - the bananas hanging;
  - the box being at the same place as the bananas.

We use appropriately chosen predicates and arguments:

```
at(monkey, a)

on(monkey, box)

status(bananas, hanging)

at(box,X), at(bananas,X)
```

The Initial State of the World:

```
on(monkey, floor),
on(box, floor),
at(monkey, a),
at(box, b),
at(bananas, c),
status(bananas, hanging)
```

The Goal State:

```
on(monkey, box),
on(box, floor),
at(monkey, c),
at(box, c),
at(bananas, c),
status(bananas, grabbed)
```

Representing Operators: We make some assumptions about the use of the operators and what needs to be stated explicitly:

Moving around in the world: only the monkey can move: `go(X,Y)`—e.g., `go(a,b)`

Pushing things around: Again, only the monkey does this: `push(B,X,Y)`—e.g., `push(box,a,b)`

Climbing on objects: `climb_on(X)` Note: we don't allow the monkey to get off the box!

Grabbing objects: `grab(X)`

Each operator has preconditions and effects on the world:

Operator	Preconditions	Effects	
		Delete	Add
go(X,Y)	at(m,X) on(m,fl)	at(m,X)	at(m,Y)
push(B,X,Y)	at(m,X) at(B,X) on(m,fl) on(B,fl)	at(m,X) at(B,X)	at(m,Y) at(B,Y)
climb_on(B)	at(m,X) at(B,X) on(m,fl) on(B,fl)	on(m,fl)	on(m,B)
grab(B)	on(m,box) at(box,X) at(B,X) status(B,h)	status(B,h)	status(B,g)

where:

m	=	monkey
fl	=	floor
h	=	hanging
g	=	grabbed

The General Solution

1. Look at the state of the world:
   - Is it the goal state? If so, the list of operators so far is the plan to be applied.
   - If not, go to Step 2.
2. Pick an operator:
   - Check it has not already been applied (i.e., check for looping).
   - Check if it can be applied (ie that the preconditions are satisfied).

   If either of these checks fails, backtrack to get another operator.
3. Apply the operator:
   - Make changes to the world: delete from and add to the world state.
   - Add the operator to the list of operators to be applied.
   - Go to Step 1.

How the World Changes

```
on(m,fl), on(box,fl), at(m,a), at(box,b),
at(bananas,c), status(bananas, hanging)
        ↓
     [go(a,b)]
        ↓
on(m,fl), on(box,fl), at(m,b), at(box,b),
at(bananas,c), status(bananas, hanging)
        ↓
   [push(box,b,c)]
        ↓
on(m,fl), on(box,fl), at(m,c), at(box,c),
at(bananas,c), status(bananas, hanging)
        ↓
   [climb_on(box)]
        ↓
on(m,box), on(box,fl), at(m,c), at(box,c),
at(bananas,c), status(bananas, hanging)
        ↓
   [grab(bananas)]
        ↓
on(m,box), on(box,fl), at(m,c), at(box,c),
at(bananas,c), status(bananas, grabbed)
```

16.4 Doing All This in Prolog

The top level predicate will be `solve/3`, whose arguments are the initial state, the goal state, and the eventual plan.

Operators will be represented by the predicate `opn/4` whose arguments are

- the operator name and arguments;
- the list of preconditions;
- what to delete from the world state;
- what to add to the world state

The Top Level in Prolog:

```
% solve(+State, +Goal, -Plan).
% Given starting State and final Goal state,
% returns a Plan consisting of a list of
% operations for transforming State into
% Goal (N.B. operators in plan are in reverse
% order of application).

solve(State, Goal, Plan):-
    solve(State, Goal, [], Plan).
```

The Operators in Prolog:

```
opn(go(X,Y),
    [at(monkey,X), on(monkey,floor)],
    [at(monkey,X)],
    [at(monkey,Y)]).

opn(push(B,X,Y),
    [at(monkey,X), at(B,X),
     on(monkey,floor), on(B,floor)],
    [at(monkey,X), at(B,X)],
    [at(monkey,Y), at(B,Y)]).

opn(climbon(B),
    [at(monkey,X), at(B,X),
     on(monkey,floor), on(B,floor)],
    [on(monkey,floor)],
    [on(monkey,B)]).

opn(grab(B),
    [on(monkey,box), at(box,X),
     at(B,X), status(B,hanging)],
    [status(B,hanging)],
    [status(B,grabbed)]).
```

The Main Predicate: The work is done by the `solve/4` predicate. There are two cases:

- if we've reached the goal state; and
- if we haven't reached the goal state.

The arguments to `solve/4` are the current state, the goal state, the sequence of operations so far, and the final plan.

- If at the goal state, the plan is the sequence of operators so far;
- If not at the goal state:
  - select an operator: (match to `opn/4`)
  - check if not a member of list so far (use `member`)
  - check if preconditions hold in world (i.e., preconditions list should be a subset of world state)
  - delete from world state what is no longer true (use `dellist`)
  - add to world state what is now true (use `append`)
  - recurse on `solve` to get the next state

The Prolog Code for `solve/4` is:

```
solve(State, Goal, Plan, Plan):-
    is_subset(Goal, State).

solve(State, Goal, Sofar, Plan):-
    opn(Op, Preconditions, Delete, Add),
    \+ member(Op, Sofar),
    is_subset(Preconditions, State),
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState),
    solve(NewState, Goal, [Op|Sofar], Plan).
```

Utility Predicates are:

```
is_subset([H|T], Set):-
    member(H, Set),
    is_subset(T, Set).
is_subset([], _).

delete_list([H|T], List, Final):-
    remove(H, List, Remainder),
    delete_list(T, Remainder, Final).
delete_list([], List, List).

remove(X, [X|T], T).
remove(X, [H|T], [H|R]):-
    remove(X, T, R).
```

16.5 Prolog Practical 9: A Simple Version of STRIPS

Read the beginning of this chapter before attempting this practical.

16.5.1 Basics

The file `/home/infteach/prolog/code/simstrips.pl` contains a simple version of a STRIPS-type means-ends analysis program, coded to solve the monkey and bananas problem. This is the program discussed above.

Copy the file to your area. Have a look at the file and make sure you understand the structure of the program. Think back to how the program is supposed to work (operators with preconditions, add and delete lists applied to some world state); try and match this to the program. Run the program by calling the top level goal

```
test(P).
```

The `test/1` predicate has a subgoal `solve/3`. The first argument of `solve` represents the initial state of the world; the second represents the goal state to be achieved; and the third will become instantiated to the plan that, when applied to the initial state, will achieve the goal state.

16.5.2 Making a Blocksworld Planner

Consider the blocksworld scenario shown in Figure 35. The configuration of objects in Figure 35 is the initial state, and the configuration in Figure 36 is the desired goal state. These states can be represented using a similar representation as the Monkey and Bananas problem. By simply modifying the representation of the problem and the operators we can modify our Monkey and Bananas planner to solve this blocksworld problem. (n.b. ignore the shapes of the blocks: just think of them as coloured blocks that can be stacked on top of each other in any configuration.)

1. Choose a representation for the blocksworld and use it to represent the initial state and goal states as depicted. The closer you stick to the representation used in the Monkey and Bananas problem the easier it will be to write the operators.
   Initial State:
   ____
   ____
   ____
   Goal State:
   ____
   ____
   ____

2. Compare the initial state and goal states. What operators do we need to move the blocks so that their configuration matches the goal state? Write the new operators in the space below, listing their preconditions, delete lists, and add lists.

```
[Operator]    [Preconditions]  [Delete list]    [Add list]

----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
```

3. Use the operators to write out a plan using the format shown in the `How the World Changes` section at the start of this chapter. Start from the initial state of the blocksworld. Apply operators one at a time, making sure that you only apply operators with preconditions that match the current state. Update the state of the world using the add and delete lists. Start again.

   Write your plan out below:

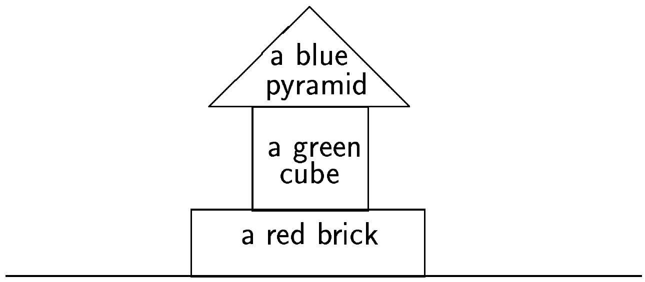


Figure 35: The initial blocksworld state

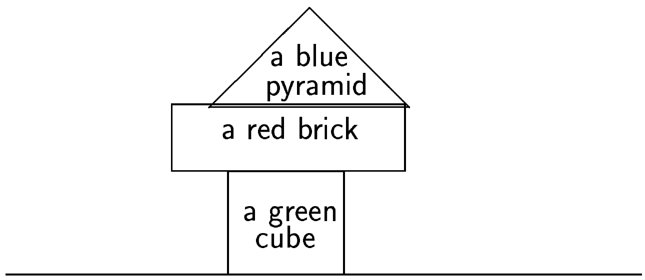


Figure 36: The goal state

4. Now add your operators to the Prolog program. Use the format previously used by the Monkey and Bananas operators. You do not need to change `test/1` or `solve/3` if you retain the same format for the operator definitions. You should be able to run your program and generate a plan.

17 Answers

17.1 Chapter 3

Question 3.1 The predicate =/2 takes 2 arguments and tries to unify them. For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
3.1a     ?- Pear = apple.
        Pear = apple    yes

3.1b     ?- car = beetle.
        no

3.1c     ?- likes(beer(murphys),john) = likes(Who,What).
        What = john,    Who = beer(murphys)     yes

3.1d     ?- f(1) = F.
        f(1) = F        yes

3.1e     ?- name(Family) = smith.
        no

3.1f     ?- times(2,2) = Four.
        Four = times(2,2)       yes

3.1g     ?- 5*3 = 15.
        no

3.1h     ?- f(X,Y) = f(P,P).
        X = P  Y = P            yes

3.1i     ?- a(X,y) = a(y,Z).
        X = y Z = y             yes

3.1j     ?- a(X,y) = a(z,X).
        no
```

Question 3.2 The following Prolog program is consulted by the Prolog interpreter.

```
vertical(seg(point(X,Y),point(X,Y1))).
horizontal(seg(point(X,Y),point(X1,Y))).
```

What will be the outcome of each of the following queries?

```
3.2a     ?- vertical(seg(point(1,1),point(1,2))).
        yes
```

Note that whilst the constants 1 and 2 match to variables in the program, the Prolog interpreter only returns the values of any variables in the query – none in this case.

```
3.2b     ?- vertical(seg(point(1,1),point(2,Y))).
        no
```

The Y in the query is independent of the variable Y in the program. In fact, when the program is interpreted the variables are renamed anyway to avoid any confusion.

```
3.2c     ?- horizontal(seg(point(1,1),point(2,Y))).
        Y = 1   yes

3.2d     ?- vertical(seg(point(2,3),P)).
        P = point(2,_A) yes
```

The second argument of `point/2`, in the second argument of `seg`, has not been instantiated by the match, so is returned as a variable.

```
3.2e     ?- vertical(S), horizontal(S).
        S = seg(point(_B,_A),point(_B,_A))      yes
```

Here is an example of renaming of variables: what were `X`, `Y`, and `Y1` in the program are renamed `_A` and `_B`

Question 3.3 The following Prolog program is consulted by the Prolog interpreter.

```
parent(pat,jim).
parent(pam,bob).
parent(bob,ann).
parent(bob,pat).
parent(tom,liz).
parent(tom,bob).
```

What will be the outcome of each of the following queries?

```
3.3a     ?- parent(bob,pat).
        yes

3.3b     ?- parent(liz,pat).
        no
```

```
3.3c     ?- parent(tom,ben).
        no

3.3d     ?- parent(Pam,Liz).
        Liz = jim  Pam = pat    yes

3.3e     ?- parent(P,C),parent(P,C2).
        C = jim  C2 = jim  P = pat  yes
```

Question 3.4 The following Prolog program is consulted by the Prolog interpreter.

```
colour(b1,red).
colour(b2,blue).
colour(b3,yellow).
shape(b1,square).
shape(b2,circle).
shape(b3,square).
size(b1,small).
size(b2,small).
size(b3,large).
```

What will be the outcome of each of the following queries?

```
3.4a     ?- shape(b3,S).
        S=square  yes

3.4b     ?- size(W,small).
        W=b1    yes

3.4c     ?- colour(R,blue).
        R=b2    yes

3.4d     ?- shape(Y,square),colour(Y,blue).
        no

3.4e     ?- size(X,large),colour(X,yellow).
        X=c3  yes

3.4f     ?- shape(BlockA,square),shape(BlockB,square).
        BlockA=b1  BlockB=b1 yes

3.4g     ?- size(b2,S),shape(b2,S).
        no

3.4h     ?- colour(b1,Shape),size(X,small),shape(Y,circle).
        Shape=red  X=b1  Y=b2   yes
```

Question 3.5 The following Prolog program is consulted by the Prolog interpreter.

```
film(res_dogs,dir(tarantino),stars(keitel,roth),1992).
film(sleepless,dir(ephron),stars(ryan,hanks),1993).
film(bambi,dir(disney),stars(bambi,thumper),1942).
film(jur_park,dir(spielberg),stars(neill,dern),1993).
```

What will be the outcome of each of the following queries?

```
3.5a     ?- film(res_dogs,D,S,1992).
        D=dir(tarantino) S=stars(keitel,roth)  yes

3.5b     ?- film(F,dir(D),stars(Who,hanks),Y).
        F=sleepless D=ephron Who=ryan Y=1993  yes

3.5c     ?- film(What,Who,stars(thumper),1942).
        no
```

3.5d Write the query that would answer the question:

"Who directed Jurassic Park (jur_park)?"

and give the outcome of the query.

```
?- film(jur_park,dir(Director),stars(X,Y),Z).
Director=spielberg   X=neill  Y=dern  Z=1993  yes
```

3.5e Write the query that would answer the question:

"What film did hanks appear in in 1993 and who was the other star?"

and give the outcome of the query.

```
        ?- film(Film,D,stars(Other,hanks),1993).
        Film=sleepless  D=dir(ephron)  Other=ryan
[might also try ?- film(Film,D,stars(hanks,Other),1993).
with outcome   no    if did not know order of stars]
```

17.2 Chapter 5

Question 5.1 The following Prolog program is consulted by the Prolog interpreter.

```
big(bear).
big(elephant).
small(cat).
```

```
brown(bear).
black(cat).
grey(elephant).
dark(Animal):- black(Animal).
dark(Animal):- brown(Animal).
```

What will be the outcome of each of the following queries?

```
5.1a     ?- dark(X), big(X).
        X = bear        yes

5.1b     ?- big(X), grey(Y).
        X = bear   Y = elephant         yes

5.1c     ?-  dark(D), small(D).
        D = cat         yes

5.1d     ?- big(Animal), black(Animal).
        no

5.1e     ?- small(P), black(P), dark(P).
        P = cat         yes
```

Question 5.2 The following Prolog program is consulted by the Prolog interpreter.

```
knows(A,B):-
        friends(A, B).

knows(A,B):-
        friends(A, C),
        knows(C, B).

friends(john, alice).
friends(alice, tom).
friends(sue, john).
friends(sue, clive).
friends(fred, tom).
friends(tom, sue).
```

State whether the following queries succeed or fail. If a query fails, explain why.

```
5.2a     ?- knows(alice, john).
        yes

5.2b     ?- knows(clive, sue).
        yes
```

```
5.2c     ?- knows(alice, fred).
        no, loops

5.2d     ?- knows(sue, john).
        no, relationship not defined
        (and cannot change order of arguments).
```

17.3 Chapter 6

For each of the following programs, say if the query given fails or succeeds. Give any bindings made as a consequence.

Question 6.1

```
        a:-b,c.
        b.
        c:-d.
        d:-e.

        ?- a.

fails (a if b and c.
        b succeeds.
        c if d.
            d if e.
                e fails.
            so d fails. so
        c fails.
      so a fails.)
```

Question 6.2

```
        a:-b,c.
        c:-e.
        b:-f,g.
        b:-n.
        e.
        f.
        n.

        ?- a.

succeeds (a if b and c.
            b if f and g.
```

```
            f succeeds. g fails.
            can't redo f so f fails.
        redo b. b if n.
            n succeeds.
        b succeeds.
        c if e.
            e succeeds.
        c succeeds.
    a succeeds.)
```

[Unfortunately sicstus prolog gives the same answer to both 5.1 and 5.2 here:

{EXISTENCE ERROR: g: procedure user:g/0 does not exist}

on the assumption that if you have predicates called with no facts for them then maybe you made an error, and you probably meant to have facts for e in 5.1 and g in 5.2. Particularly this is unfortunate because the sicstus interpreter does not go on to prove b if n, but stops to tell you that g does not exist. I think this is a bad design decision: you should have got a warning here, as with singleton variables, not an "existence error".]

Question 6.3

```
do(X):-a(X),b(X).
 a(X):-c(X),d(X).
 a(X):-e(X).
 b(X):-f(X).
 b(X):-c(X).
 b(X):-d(X).
 c(1).
 c(3).
 d(3).
 d(2).
 e(2).
 f(1).
```

```
6.3a        ?- do(1).       no
        (do(1) if a(1) and b(1)
                a(1) if c(1) and d(1)
                        c(1) succeeds
                        d(1) fails
                redo a(1) if e(1)
                        e(1) fails
                a(1) fails
        do(1) fails)
```

```
6.3b        ?- do(2).       yes
       (do(2) if a(2) and b(2)
              a(2) if c(2) and d(2)
                     c(2) fails
              redo a(2) if e(2)
                     e(2) succeeds
              a(2) succeeds
              b(2) if f(2)
                     f(2) fails
              redo b(2) if c(2)
                     c(2) fails
              redo b(2) if d(2)
                     d(2) succeeds
              b succeeds
       do(2) succeeds)

6.3c        ?- do(3).       yes
       (do(3) if a(3) and b(3)
              a(3) if c(3) and d(3)
                     c(3) succeeds
                     d(3) succeeds
              a(3) succeeds
              b(3) if f(3)
                     f(3) fails
              redo b(3) if c(3)
                     c(3) succeeds
              b(3) succeeds
       do(3) succeeds)

6.3d        ?- do(A).       A = 3  yes
       (do(A) if a(A) and b(A)
              a(A) if c(A) and d(A)
                     c(1) succeeds
                     d(1) fails
              redo c(A)
                     c(3) succeeds
                     d(3) succeeds
              a(3) succeeds
              b(3) if f(3)
                     f(3) fails
              redo b(3) if c(3)
                     c(3) succeeds
              b(3) succeeds
       do(3) succeeds)
```

17.4 Chapter 8

Question 8.1 How many elements are there in each of the following list structures?

```
8.1a    [a,  [a,[a,[a]]]]       = 2

8.1b    [1,2,3,1,2,3,1,2,3]     = 9

8.1c    [a(X),  b(Y,Z),  c,  X] = 4

8.1d    [[sum(1,2)],  [sum(3,4)],  [sum(4,6)]]  = 3

8.1e    [c,  [d,[x]],  [f(s)],  [r,h,a(t)],  [[[a]]]]   = 5
```

[Note: extra spaces used to illustrate each element]

Question 8.2 The predicate =/2 takes 2 arguments and tries to unify them. For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables;
- if it fails, explain why it fails.

```
8.2a   ?- [A,B,C,D] = [a,b,c].          no      lists are different length.

8.2b   ?- bar([1,2,3]) = bar(A).        yes     A = [1,2,3].

8.2c   ?- [X,[]] = [X].                 no      lists are different length.

8.2d   ?- [X,Y|Z] = [a,b,c,d].          yes     X=a, Y=b, Z=[c,d].

8.2e   ?- [1,X,X] = [A,A,2].            no      A bound to 1, cannot then
                                                 be bound to 2.

8.2f   ?- likes(Y,a) = likes(X,Y).      yes     X=a, Y=a.
                                                 (X and Y share)

8.2g   ?- [foo(a,b),a,b] = [X|Y].       yes     X=foo(a,b), Y=[a,b].

8.2h   ?- [b,Y] = [Y,a].                no      Y cannot be bound to b and a.

8.2i   ?- [H|T] = [red,blue,b(X,Y)].    yes     H=red, T=[blue,b(X,Y)].

8.2j   ?- [1,[a,b],2] = [[A,B],X,Y].    no      lists cannot unify with atoms.
```

```
8.2k   ?- test(a,L) = test(E,[b,c,d]). yes     E = a, L = [b,c,d]

8.2l   ?- [[a,[b]],C] = [C,D].         yes     C = [a,[b]], D =[a,[b]]

8.2m   ?- [fred|T]=[H|[sue,john]].     yes     H = fred T = [sue,john]
```

Question 8.3 Imagine that this program is consulted by the Prolog interpreter:

```
foo([],[]).

foo([H|T], [X|Y]):-
        H = X,
        foo(T,Y).
```

[Note: this program tests if two lists unify by testing if the heads unify then recursing on the tails]

What will be the outcome of each of the following queries?

```
8.3a   ?- foo([a,b,c], A).             yes     A=[a,b,c].

8.3b   ?- foo([c,a,t], [c,u,t]).       no

8.3c   ?- foo(X, [b,o,o]).             yes     X=[b,o,o].

8.3d   ?- foo([p|L], [F|[a,b]]).       yes     F=p, L=[a,b].

8.3e   ?- foo([X,Y], [d,o,g]).         no
```

17.5 Chapter 9

Question 9.1 The predicate member/2 succeeds if the first argument matches an element of the list represented by the second argument.

```
e.g.    ?- member(1,[2,3,1,4]).          yes

member/2 is defined as:         1. member(El,[El|T]).
                                2. member(El,[H|T]):-
                                        member(El,T).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;

- if it succeeds, specify what is assigned to any variables.

```
9.1a ?- member(a,[c,a,b]).                        yes

9.1b ?- member(a,[d,o,g]).                        no

9.1c ?- member(one,[one,three,one,four]).         yes

9.1d ?- member(X,[c,a,t]).                        X=c  yes

9.1e ?- member(tom,[[jo,alan],[tom,anne]]).       no
```

9.1f Complete the AND/OR tree below which represents the execution of the query:

```
Tree:
        ?- member(a,[b,r,e,a,d]).                yes
        /        \
      1/          2 \
  a=b /           member(a,[r,e,a,d])
 fails              /     \
              1 /          \ 2
              a=r       member(a,[e,a,d])
             fails        /       \
                        1/          \2
                       a=e      member(a,[a,d])
                                   /
                                 1 /
                             succeeds

Trace:
   1  1  Call: member(a,[b,r,e,a,d]) ?
   2  2  Call: member(a,[r,e,a,d]) ?
   3  3  Call: member(a,[e,a,d]) ?
   4  4  Call: member(a,[a,d]) ?
   4  4  Exit: member(a,[a,d]) ?
   3  3  Exit: member(a,[e,a,d]) ?
   2  2  Exit: member(a,[r,e,a,d]) ?
   1  1  Exit: member(a,[b,r,e,a,d]) ?
yes
```

Question 9.2 The predicate no_cons/1 succeeds if all elements of the list represented by the one argument are vowels (as specified by vowel/1).

```
e.g.    ?- no_cons([a,e,i]).
        yes
```

```
? no_cons([a,b,c]).
no

no_cons/1 is defined as:
1. no_cons([]).
2. no_cons([H|T]):-
        vowel(H),
        no_cons(T).

3. vowel(a).
4. vowel(e).
5. vowel(i).
6. vowel(o).
7. vowel(u).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

9.2a `?- no_cons([a,a,e,i]).` `yes`

9.2b `?- no_cons([A,e,e]).` `A=a  yes`

9.2c Complete the AND/OR tree below which represents the execution of the query:

```
        ?- no_cons([a,e,i]).
          /      |\
       1/      2|__\                  H/a     T/[e,i]
      /          |    \
  a=[]       vowel(a) no_cons([e,i])
 fails           |        |\
                3|      2|__\                 H1/e    T1/[i]
                 |        |    \
           succeeds  vowel(e) no_cons([i])
                          |         |\
                         4|       2|__\                 H2/i    T2/[]
                          |         |    \
                    succeeds   vowel(i) no_cons([])
                                    |       |
                                   5|      1|
                                    |       |
                              succeeds  succeeds

yes
```

Question 9.3 The predicate max/2 succeeds if the first argument is a list of numbers and second argument unifies with the maximum value in that list.

```
e.g.    ?- max([2,4,6,8],M).
        M = 8
        yes

max/2 is defined as:

        0. max([H|T],Max):-
                max(T,H,Max).

        1. max([H|T],Temp,Max):-
                H>Temp,
                max(T,H,Max).
        2. max([H|T],Temp,Max):-
                max(T,Temp,Max).
        3. max([],Finalmax,Finalmax).
```

9.3 Complete the AND/OR tree below which represents the execution of the query:

```
?- max([2,1,4,3],Ans).
             |
            0|            H/2     T/[1,4,3]
             |
   max([1,4,3],2,Ans)
      /     \
   1/        2\                H1/1    Temp/2  T1/[4,3]
  /            \
1>2        max([4,3],2,Ans)
fails          |\
              1|--\                    H2/4    Temp1/2   T2/[3]
               |    \
              4>2  max([3],4,Ans)
           succeeds     |\
                       1| 2\           H3/3    Temp2/4   T3/[]
                        |    \
                      3>4   max([],4,Ans)
                     fails     /|\
                             1/ 2| 3\              Finalmax/4
                             /    |    \
                         fails  fails  succeeds
                                        Ans=4
```

```
Ans = 4
yes
```

Question 9.4 The predicate prlist/1 is supposed to write out the elements of a list structure, regardless of the levels of embedding that are present in the list.

```
 e.g. intended behaviour:
        ?- prlist([a,b,[c,d,e],f,[g]]).
        abcdefg
        yes
```

Instead, the predicate as defined below has the following behaviour:

```
?- prlist([a,b]).
ab[]
yes

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

prlist/2 is defined as:

1. prlist([H|T]):-
```

```
                prlist(H),
                prlist(T).
        2. prlist(X):-
                write(X).
```

Explain why this predicate produces this behaviour (instead of the intended behaviour), using an AND/OR tree or a trace in your explanation.

```
Tree:
                ?- prlist([a,b]).
                    /---\
                 1 /     \
             prlist(a)    prlist([b])
             /   |           /---\
          1 /    |2       1 /     \
       fails  write(a)  prlist(b)  prlist([])
                         / \          / \
                      1 /   \2       /1  \2
                   fails  write(b) fails write([])

ab[]
```

The problem here is that whatever its value, the head and tail of the list on each recursion are prlisted, and when the head is no longer a list it is written. So the empty list also gets written. An extra clause, prlist([]) is needed to prevent this.

Question 9.5 The predicate checkvowels takes a list representing a word, checks each letter to see whether it is a vowel, and if it is it writes out the vowel.

```
        checkvowels([H|T]):-
                vowel(H),
                write(H), nl,
                checkvowels(T).

        checkvowels([H|T]):-
                checkvowels(T).

        checkvowels([]).

        vowel(a).
        vowel(e).
        vowel(i).
        vowel(o).
        vowel(u).
```

```
e.g.    ?- checkvowels([c,a,t,i]).
        a
        i
        yes
```

9.5a Using this as a model, write a predicate results/1 that takes a list of names, checks whether each is a pass (using a predicate that you also must define, pass/1) and writes out the names if they pass.

For example, the following query:

```
?- results([tom,bob,sue,jane]).
```

should give the output:

```
bob
sue
yes
```

Solution:

```
pass(bob).
pass(sue).
results([H|T]):-
       pass(H), write(H), nl,
       results(T).
results([H|T]):-
       results(T).

results([]).
```

9.5b Modify this program so that instead of 'writing out' the name of each person who passes it should output a list of them.

```
?- results2([tom,bob,sue,jane],Passlist).
Passlist = [bob,sue]
yes

results2([H|T],[H|Pl]):-
        pass(H),
        results2(T,Pl).
results2([H|T],Pl):-
        results2(T,Pl).
results2([],[]).
```

17.6 Chapter 10

Question 10.1 The predicate delete/3 succeeds if deleting the element represented by the first argument, from the list represented by the second argument, results in a list represented by the third argument.

```
e.g.    ?- delete(a,[a,p,p,l,e],A).
        A=[p,p,l,e]
        yes

        delete/3 is defined as:

        1. delete(El,[El|T],T).
        2. delete(El,[H|T],[H|NT]):-
                delete(El,T,NT).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
10.1a ?- delete(X,[pat,john,paul],Ans).   X = pat   Ans = [john,paul]  yes

10.1b ?- delete(A,L,[t,o,p]).              A = _A    L = [_A,t,o,p]  yes
                                         or L = [A,t,o,p]

10.1c ?- delete(a,[c,a,b],Ans).                     Ans = [c,b]  yes

10.1d ?- delete(e,[d,o,g],P).                       no

10.1e ?- delete(e,[f,e,e,t],Ans).                   Ans = [f,e,t] yes
```

Question 10.2 The predicate deleteall/3 succeeds if deleting all occurrences of the element represented by the first argument from the list represented by the second argument results in a list represented by the third argument.

```
e.g.    ?- delete(p,[a,p,p,l,e],A).     A=[a,l,e]       yes

deleteall/3 is defined as:      1. deleteall(El,[],[]).
                                2. deleteall(El,[El|T],NT):-
                                        deleteall(El,T,NT).
                                3. deleteall(El,[H|T],[H|NT]):-
                                        deleteall(El,T,NT).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
10.2a  ?- deleteall(e,[f,e,e,t],Ans).              Ans = [f,t]  yes

10.2b  ?- deleteall(p,[d,o,g],X).                  X = [d,o,g] yes
```

10.2c Complete the AND/OR tree below which represents the execution of the query:

```
        ?- deleteall(e,[f,e,e,t],Ans).
          /     |         \
        1/      |2       3 \ Ans/[f|NT]
        /       |           \
[]=[f,e,e,t]   e=f    deleteall(e,[e,e,t],NT)
  fails       fails    /    \
                      /      \
                  1 /       2\ NT/NT2
                   /          \
             []=[e,e,t]    deleteall(e,[e,t],NT2)
              fails
```

Solution:

```
Tree:
        ?- deleteall(e,[f,e,e,t],Ans).                   Ans= [f,t] yes
           /    |      \
        1 /     2       \ 3 Ans/[f|NT]                           = [f,t]
[]=[f,e,e,t]   e=f    deleteall(e,[e,e,t],NT)
  fails       fails   /    |
                     /     |
                 1 /     2| NT/NT2                               = [t]
            []=[e,e,t]  deleteall(e,[e,t],NT2)
             fails       /     \
                        /       \
                       /1        \ 2  NT2/NT3                    = [t]
                   []=[e,t]   deleteall(e,[t],NT3)
                    fails     /   |   \
                             /    |    \
                        1 /   2 |     \ 3 NT3/[t|NT4]         = [t]
                      []=[t]    e=t  deleteall(e,[],NT4)
                      fails    fails      |
                                          |
                                          |1  NT4/[]
                                       succeeds
```

Trace:

```
   1  1  Call: deleteall(e,[f,e,e,t],_115) ?
   2  2  Call: deleteall(e,[e,e,t],_288) ?
   3  3  Call: deleteall(e,[e,t],_288) ?
   4  4  Call: deleteall(e,[t],_288) ?
   5  5  Call: deleteall(e,[],_626) ?
   5  5  Exit: deleteall(e,[],[]) ?
   4  4  Exit: deleteall(e,[t],[t]) ?
   3  3  Exit: deleteall(e,[e,t],[t]) ?
   2  2  Exit: deleteall(e,[e,e,t],[t]) ?
   1  1  Exit: deleteall(e,[f,e,e,t],[f,t]) ?
Ans = [f,t] ?
yes
```

10.2d Suppose that the predicate deleteall/3 is defined incorrectly as:

```
        1. delall(E,[],[]).
        2. delall(E,[E|T],Y):-
                delall(E,T,Y).
        3. delall(E,[H|T],Y):-
                delall(E,T,[H|Y]).
```

resulting in the behaviour:

```
i.  ?- delall(e,[f,e,e,t],Ans).
   no
```

However, the following query succeeds, as intended:

```
ii. ?- delall(a,[a,a,a],Ans).
   Ans=[]
   yes
```

Explain why the program does not give the intended answer to query i. using an AND/OR tree or a trace to illustrate your answer.

```
| ?-  delall(e,[f,e,e,t],Ans).
 + 1  1  Call: delall(e,[f,e,e,t],_95) ?
 + 2  2  Call: delall(e,[e,e,t],[f|_95]) ?
 + 3  3  Call: delall(e,[e,t],[f|_95]) ?
 + 4  4  Call: delall(e,[t],[f|_95]) ?
 + 5  5  Call: delall(e,[],[t,f|_95]) ?
 + 5  5  Fail: delall(e,[],[t,f|_95]) ?
 + 4  4  Fail: delall(e,[t],[f|_95]) ?
 + 4  4  Call: delall(e,[t],[e,f|_95]) ?
 + 5  5  Call: delall(e,[],[t,e,f|_95]) ?
 + 5  5  Fail: delall(e,[],[t,e,f|_95]) ?
 + 4  4  Fail: delall(e,[t],[e,f|_95]) ?
```

```
 + 3  3  Fail: delall(e,[e,t],[f|_95]) ?
 + 3  3  Call: delall(e,[e,t],[e,f|_95]) ?
 + 4  4  Call: delall(e,[t],[e,f|_95]) ?
 + 5  5  Call: delall(e,[],[t,e,f|_95]) ?
 + 5  5  Fail: delall(e,[],[t,e,f|_95]) ?
 + 4  4  Fail: delall(e,[t],[e,f|_95]) ?
 + 4  4  Call: delall(e,[t],[e,e,f|_95]) ?
 + 5  5  Call: delall(e,[],[t,e,e,f|_95]) ?
 + 5  5  Fail: delall(e,[],[t,e,e,f|_95]) ?
 + 4  4  Fail: delall(e,[t],[e,e,f|_95]) ?
 + 3  3  Fail: delall(e,[e,t],[e,f|_95]) ?
 + 2  2  Fail: delall(e,[e,e,t],[f|_95]) ?
 + 1  1  Fail: delall(e,[f,e,e,t],_95) ?
no


{trace}
| ?- delall(a,[a,a,a],Ans).
 + 1  1  Call: delall(a,[a,a,a],_89) ?
 + 2  2  Call: delall(a,[a,a],_89) ?
 + 3  3  Call: delall(a,[a],_89) ?
 + 4  4  Call: delall(a,[],_89) ?
 + 4  4  Exit: delall(a,[],[]) ?
 + 3  3  Exit: delall(a,[a],[]) ?
 + 2  2  Exit: delall(a,[a,a],[]) ?
 + 1  1  Exit: delall(a,[a,a,a],[]) ?

Ans = [] ?

yes

Builds in the body rather than the head of the clause.
```

Question 10.3 The predicate repall/4 succeeds if replacing all occurrences of the element represented by the first argument, by the element represented by the second argument, in the list represented by the third argument, results in a list represented by the fourth argument.

```
e.g.    ?- repall(p,b,[a,p,p,l,e],A).
        A=[a,b,b,l,e]
        yes

        repall/4 is defined as:

        1. repall(El,Rel,[],[]).
        2. repall(El,Rel,[El|T],[Rel|NT]):-
                 repall(El,Rel,T,NT).
        3. repall(El,Rel,[H|T],[H|NT]):-
```

```
        repall(El,Rel,T,NT).
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
10.3a ?- repall(e,a,[f,e,a,t],Ans).       Ans = [f,a,a,t] yes

10.3b ?- repall(P,r,[s,o,u,p],Ans).       Ans = [r,o,u,p] P = s  yes
```

Question 10.4 The predicate whowants/3 has three arguments. The first is intended to represent a type of food; the second a list of people; and the third another list, representing those people on the 2nd argument list who want the type of food specifed in the 1st argument (where want/2 is defined separately, with two arguments representing who wants what food).

```
e.g.    ?- whowants(beans,[jo,tom,ann],Who).
        Who = [jo,ann]
        yes

        1. whowants(Food,[Name|Rest],[Name|Others]):-
                wants(Name,Food),
                whowants(Food,Rest,Others).
        2. whowants(Food,[Name|Rest],Others):-
                whowants(Food,Rest,Others).
        3. whowants(Food,[],[]).

        4. wants(jo,chips).
        5. wants(jo,beans).
        6. wants(jo,eggs).
        7. wants(ann,beans).
        8. wants(ann,bacon).
        9. wants(tom,eggs).
        10. wants(tom,chips).
        11. wants(rick,bacon).
```

10.4a Give either the AND/OR tree or a trace which represents the execution of the query:

```
        ?- whowants(beans,[jo,tom,ann],Who).
             /\
          1/----\        Food/beans Name/jo Rest/[tom,ann] Who/[jo|Others]
          /      \
```

```
wants(jo,beans) whowants(beans,[tom,ann],Others)
    |              /  \
   5|           1/     2\       Name1/tom Rest1/[ann]
    |          /          \
succeeds wants(tom,beans) whowants(beans,[ann],Others)
              |              /\
           fails        1/----\    Name2/ann  Rest2/[] Others/[ann|Others1]
                       /        \
              wants(ann,beans) whowants(beans,[],Others1)
                   |              /|\
                  7|          1/ 2| 3\         Others1/[]
                   |          /    |    \
               succeeds    fails fails succeeds
                                       Others1=[]

Who=[jo,ann]
yes

| ?- trace, whowants(beans,[jo,tom,ann],Who).
 + 1  1  Call: whowants(beans,[jo,tom,ann],_75) ?
 + 2  2  Call: wants(jo,beans) ?
 + 2  2  Exit: wants(jo,beans) ?
 + 3  2  Call: whowants(beans,[tom,ann],_681) ?
 + 4  3  Call: wants(tom,beans) ?
 + 4  3  Fail: wants(tom,beans) ?
 + 4  3  Call: whowants(beans,[ann],_681) ?
 + 5  4  Call: wants(ann,beans) ?
 + 5  4  Exit: wants(ann,beans) ?
 + 6  4  Call: whowants(beans,[],_1438) ?
 + 6  4  Exit: whowants(beans,[],[]) ?
 + 4  3  Exit: whowants(beans,[ann],[ann]) ?
 + 3  2  Exit: whowants(beans,[tom,ann],[ann]) ?
 + 1  1  Exit: whowants(beans,[jo,tom,ann],[jo,ann]) ?
Who = [jo,ann] ?
yes
```

For each of the following:

- specify whether the query submitted to Prolog succeeds or fails;
- if it succeeds, specify what is assigned to any variables.

```
10.4b ?- whowants(chips,[ann,rick],Ans).          Ans = [] yes

10.4c ?- whowants(bacon,[tom,ann],A).             A = [ann].

10.4d ?- whowants(eggs,Diners,Ans).               fails loops
```

```
| ?- trace,whowants(eggs,Diners,Ans).
 + 1  1  Call: whowants(eggs,_53,_79) ?
 + 2  2  Call: wants(_677,eggs) ?
 + 2  2  Exit: wants(jo,eggs) ?
 + 3  2  Call: whowants(eggs,_678,_676) ?
 + 4  3  Call: wants(_1232,eggs) ?
 + 4  3  Exit: wants(jo,eggs) ?
 + 5  3  Call: whowants(eggs,_1233,_1231) ?
 + 6  4  Call: wants(_1787,eggs) ?
 + 6  4  Exit: wants(jo,eggs) ?
 + 7  4  Call: whowants(eggs,_1788,_1786) ?
 + 8  5  Call: wants(_2342,eggs) ?
 + 8  5  Exit: wants(jo,eggs) ?
...........
```

10.4e Using whowants/3 as a model, write a predicate overage/3 that takes an age limit and a list of names, checks the age of each person named (using a predicate age/2) and returns a list of names of those people who are over the age limit.

For example, the query below should give the output shown:

```
        ?- overage(18,[sally,alice,bill],Ans).
        Ans=[alice,bill]
        yes
where:
        age(sally,15).
        age(mark,26).
        age(bill,20).
        age(alice,37).
```

Solution:

```
        overage(L,[H|T],[H|Y]):-
                age(H,X),
                X>L,
                overage(L,T,Y).
        overage(L,[H|T],Y):-
                overage(L,T,Y).
        overage(L,[],[]).
```

10.4f If the predicate whowants/3 had been (incorrectly) defined as:

```
        1. whowants(Food,[Name|Rest],[Name|Others]):-
                wants(Name,Food),
                whowants(Food,Rest,Others).
```

```
2. whowants(Food,[Name|Rest],Others):-
       whowants(Food,Rest,Others).
```

(i.e. no 3rd clause) predict the outcome of the query in 10.4c.

```
?- whowants(bacon,[tom,ann],A).
```

and explain why this is the case, using an AND/OR tree or a trace in your explanation.

```
| ?- trace, whowants(bacon,[tom,ann],A).
 + 1  1  Call: whowants(bacon,[tom,ann],_69) ?
 + 2  2  Call: wants(tom,bacon) ?
 + 2  2  Fail: wants(tom,bacon) ?
 + 2  2  Call: whowants(bacon,[ann],_69) ?
 + 3  3  Call: wants(ann,bacon) ?
 + 3  3  Exit: wants(ann,bacon) ?
 + 4  3  Call: whowants(bacon,[],_868) ?
 + 4  3  Fail: whowants(bacon,[],_868) ?
 + 3  3  Redo: wants(ann,bacon) ?
 + 3  3  Fail: wants(ann,bacon) ?
 + 3  3  Call: whowants(bacon,[],_69) ?
 + 3  3  Fail: whowants(bacon,[],_69) ?
 + 2  2  Fail: whowants(bacon,[ann],_69) ?
 + 1  1  Fail: whowants(bacon,[tom,ann],_69) ?
no
```

Fails because there is now no clause to match the empty list.

Question 10.5 The predicate deletefirst/3 is supposed to succeed if deleting the element represented by the first argument, from the list represented by the second argument, results in a list represented by the third argument, as delete/3 defined as above (B.)

```
e.g. intended behaviour:
        ?- deletefirst(i,[b,i,b,s],A).
        A=[b,b,s]
        yes

Instead, it produces the following behaviour:
        ?- deletefirst(i,[b,i,b,s],A).
        A=[s]
        yes

        ?- deletefirst(a,[b,a,l,l],X).
        no
```

```
deletefirst/3 is defined as:

1. deletefirst(El,[El|T],T).
2. deletefirst(El,[H|T],NT):-
      deletefirst(El,T,[H|NT]).
```

Explain why this predicate produces this behaviour (instead of the intended behaviour), using an AND/OR tree or a trace in your explanation.

Solution:

```
Tree:
       ?- deletefirst(i,[b,i,b,s],A).                 A=[s] yes
             /       |
            /1       |2                               El/i
T/[i,b,s]
           /         |                                NT/A    H/b
         i=b     deletefirst(i,[i,b,s],[b|A]).
        fails        |                                        T/[b,s]
                     | 1                                      A/[s]
                    i=i
                 [b,s]=[b|A]
                  succeeds
```

```
Trace:
| ?- deletefirst(i,[b,i,b,s],A).
   1  1  Call: deletefirst(i,[b,i,b,s],_103) ?
   2  2  Call: deletefirst(i,[i,b,s],[b|_103]) ?
   2  2  Exit: deletefirst(i,[i,b,s],[b,s]) ?
   1  1  Exit: deletefirst(i,[b,i,b,s],[s]) ?

A = [s] ?       yes
```

Matches [b—X] from first match of [b—[i,b,s,]] to [b,s] left after matching i to [i,b,s]. - so only works by coincidence of 2 occurences here.

```
{trace}
| ?- deletefirst(a,[b,a,l,l],X).
   1  1  Call: deletefirst(a,[b,a,l,l],_103) ?
   2  2  Call: deletefirst(a,[a,l,l],[b|_103]) ?
   3  3  Call: deletefirst(a,[l,l],[a,b|_103]) ?
   4  4  Call: deletefirst(a,[l],[l,a,b|_103]) ?
   5  5  Call: deletefirst(a,[],[l,l,a,b|_103]) ?
   5  5  Fail: deletefirst(a,[],[l,l,a,b|_103]) ?
   4  4  Fail: deletefirst(a,[l],[l,a,b|_103]) ?
   3  3  Fail: deletefirst(a,[l,l],[a,b|_103]) ?
```

```
   2  2  Fail: deletefirst(a,[a,l,l],[b|_103]) ?
   1  1  Fail: deletefirst(a,[b,a,l,l],_103) ?
no
No match here.
```

The problem is that the head of the list that is supposed to be constructed is copied onto the front of the list that is called in the third argument of the recursive call (in the body of the clause), rather than in the query itself (in the head of the clause). So it gets built up as the program recurses, the program then fails when it cannot split the empty list. The variable represented by Ans does not get instantiated normally in this version, but with the call of:

```
?- deletefirst(i,[b,i,b,s],Ans)
```

co-incidentally at one point the program matches [b,s] to [b—Ans], causing Ans to be instantiated to [s].

References

[Bratko 01] I. Bratko. *Prolog Programming for Artificial Intelligence (3rd edition)*. Addison Wesley, 2001. ISBN 0-201-41606-9.

[Clocksin & Mellish 03] W.F. Clocksin and C.S. Mellish. *Programming in Prolog: Using the ISO Standard (5th edition)*. Springer-Verlag, 2003. ISBN 0-387-58350-5.

[Sterling & Shapiro 94] L. Sterling and E. Shapiro. *The Art of Prolog (Second edition)*. MIT Press, 1994. ISBN 0-262-19338-8.