CS3510: Artificial Intelligence – Prolog Practical 2001 – 2002

Exercises

Peter Lucas

Department of Computing Science
University of Aberdeen, Aberdeen

Contents

1 Learning outcomes **1**

2 AI and programming languages **2**
- 2.1 Symbol manipulation 2
- 2.2 Characteristics of PROLOG 3

3 Introductory remarks about PROLOG **4**
- 3.1 PROLOG under Windows and Unix 4
- 3.2 PROLOG programs 5

4 PROLOG exercises **6**
- 4.1 Meet The Dutch Royal Family: facts, queries and rules 6
- 4.2 Lists and recursion 8
- 4.3 Unification and matching 10
- 4.4 Backtracking 11
- 4.5 Tracing and spying 12
- 4.6 Variation of input-output parameters 14
- 4.7 The order of clauses and conditions 14
- 4.8 Fail and cut 16
- 4.9 A finite automaton 17
- 4.10 Sorting 18
- 4.11 A relational database in PROLOG 18
- 4.12 Natural language processing 22

1 Learning outcomes

This Prolog practical offers you the opportunity to familiarise yourself with the basic principles of the programming language Prolog. In the practical you will learn:

- About the role and place of Prolog in the area of Artificial Intelligence (AI), and in programming-language research more in general (See Section 2);

- The basic principles of Prolog by doing a number of exercises, described in Section 4 – the results you obtain for the two ★-marked exercises should be discussed with the lecturer;
- To develop a Prolog program on your own, by doing the assignment which will be distributed to you in week four of the course.

You can either use SWI-Prolog, a good Prolog interpreter and compiler available in the public domain, or Sicstus Prolog.[1] SWI-Prolog is very close to standard Prolog; the examples in this practical manual are therefore based on SWI-Prolog. Sometimes small changes are necessary to get them running under Sicstus Prolog.

A good book to learn programming in Prolog in the context of AI is *I. Bratko, PROLOG Programming for Artificial Intelligence, 3rd ed, Addison-Wesley, Harlow, 2001* (earlier editions are also useful), but there are many other good books about Prolog. However, **we recommend you to obtain Bratko's book**.

2 AI and programming languages

2.1 Symbol manipulation

AI programs can be written in any decent, sufficiently expressive programming language, including typical imperative programming languages such as C, C++, Java and Pascal. However, many developers prefer to use programming languages such as Prolog and Lisp. Both languages have a basis which cannot be traced back to the early ideas of computing as state change of variables, as holds for the imperative languages, but instead to more abstract, *mathematical notions of computing*. It is this formal foundation which explains why these two languages are enormously expressive, much more than the imperative languages. As a consequence, it is straightforward to create and manipulate complex data structures in these languages, which is not only convenient but really essential for AI, as AI people tend to solve complicated problems. Although much of what will be said in this introduction also holds for Lisp, we will not pay further attention to this latter programming language.

Prolog is frequently used for the development of interpreters of particular (formal) languages, such as other programming languages, algebraic or logical languages, and knowledge-representation languages in general. Languages are defined in terms of preserved (key)words, i.e. symbolic names with a fixed meaning. In Pascal, for instance, the keywords `program`, `begin`, `end` are used. It is rather straightforward to develop a Prolog program that is able to identify such keywords, and to interpreter these keywords in their correct semantic context. Of course, this does not mean that it is not possible to develop such interpreters in imperative languages, but almost all necessary facilities to do so are already included in any Prolog system and really constitute the core of the language. Compare this to the situation in imperative programming languages, where one needs to resort to preprocessors and class libraries, making it often far from easy to understand what is happening. In other words, the level of *abstraction* offered by Prolog is much closer to the actual problem one is trying to solve than to the machine or software environment being used. Using abstraction is preferred by most people when developing a program for an actual problem, as it means: to ignore as

[1] See `http://www.swi.psy.uva.nl/projects/SWI-Prolog`, and `http://www.sics.se/ps/sicstus.html`; there are free SWI-Prolog systems available for Linux, Solaris and Windows.

many irrelevant details as possible, and to focus on the problem issues that really matter. When you are sufficiently versed in PROLOG, the languages allows you to develop significant programs with only limited effort.

Although PROLOG and AI are often mentioned in one phrase, this does not mean that PROLOG is only suitable for the development of AI programs. In fact, the language has and is being used as a vehicle for database, programming language, medical informatics and bioinformatics research. And finally, the language is used by many people simply to solve a problem they have to deal with, resulting in systems that from the user's point of view are indistinguishable from other systems, as in particular commercial PROLOG systems offer extra language facilities to develop attractive GUIs.[2]

2.2 Characteristics of PROLOG

There are a number of reasons why PROLOG is so suitable for the development of advanced software systems:

- *Logic programming.* PROLOG is a logical language; the meaning of a significant fraction of the PROLOG language can be completely described and understood in terms of the Horn subset of first-order predicate logica (Horn-clause logic). So, this is the mathematical foundation of the language, explaining its expressiveness.

- *A single data structure as the foundation of the language.* PROLOG offers the *term* as the basic data structure to implement any other data structure (lists, arrays, trees, records, queues, etc.). The entire language is tailored to the manipulation of terms.

- *Simple syntax.* The syntax of PROLOG is much simpler than that of the imperative programming languages. A PROLOG program is actually from a syntactical point of view simply a sequence of terms. For example, the following PROLOG program

  ```
  p(X) :- q(X).
  ```

  corresponds to the term `:-(p(X),q(X))`. Whereas the definition of the formal syntax of a language such as Java or Pascal requires many pages, PROLOG's syntax can be described in a few lines. Finally, the syntax of data is exactly the same as the syntax of programs!

- *Program–data equivalence.* Since in PROLOG, programs and data conform to the same syntax, it is straightforward to interpret programs as data of other programs, and also to take data as programs. This feature is of course very handy when developing interpreters for languages.

- *Weak typing.* Types of variables in PROLOG do not have to be declared explicitly. Whereas in C or Java, the type of any object (for example `int`, `real`, `char`) needs to be known before the compiler is able to compile the program, the type of a variable in PROLOG only becomes relevant when particular operations are performed on the variable. This so-called weak typing of program objects has as a major advantage that program design decisions can be postponed to the very last moment. This eases the

[2] SWI-PROLOG offers a special object-oriented library for GUI development called XPCE. This is not covered in this practical, however.

programmer's task. However, weak typing not only comes with advantages, as it may make finding program bugs more difficult. Expert programmers are usually able to cope with these problems, while in addition most PROLOG systems come with tools for the analysis of programs which may be equally useful for finding bugs.

- *Incremental program development.* Normally, a C or Java program needs to be developed almost fully before it can be executed. A PROLOG program can be developed and tested incrementally. Instead of generating a new executable, only the new program fragments need to be interpreted or compiled. It is even possible to modify a program during its execution. This offers the programmer the possibility of incremental program development, a very powerful and flexible approach to program development. It is mostly used in research environments and in the context of prototyping.

- *Extensibility.* A PROLOG system can be extended and modified. It is even possible to modify the PROLOG language, and to adapt both its syntax and semantics to the needs. For example, although PROLOG is not an object-oriented language, it can be extended quite easily into an object-oriented language, including primitives for inheritance and message passing. Compare this to the task of modifying a Java interpreter into a PROLOG compiler, something which would require an almost complete redesign and re-implementation of the original Java system.

3 Introductory remarks about PROLOG

By means of a number of small PROLOG programs, which can be found in the directory `/home/student/courses/cs3510/prolog`, the most important features of the PROLOG language will be studied in this practical. The programs are available in the following files:

exercise0.pl	(*facts, queries and rules*)
exercise1.pl	(*lists and recursion*)
exercise2.pl	(*backtracking*)
exercise3.pl	(*parameters*)
exercise4.pl	(*order of clauses*)
exercise5.pl	(*fail and cut*)
exercise6.pl	(*finite automaton*)
exercise7.pl	(*sorting*)
exercise8.pl	(*relational database*)
exercise9.pl	(*natural language processing*)

Copy these files to your own directory.

3.1 PROLOG under Windows and Unix

Under Windows SWI-PROLOG is activated and reads in a PROLOG program by *clicking on the program* (which should have the `.pl` extension in order to be recognised). SICSTUS-PROLOG is also available under Windows.

One can quit the system by the command:

```
halt.
```

The SWI-Prolog system can be activated under Unix by typing in:

```
pl
```

The Sicstus-Prolog system can be activated under Unix by typing in:

```
sicstus
```

A program stored in a file, for example the file **exercise0.pl**, can be read into the Prolog interpreter by typing (assuming you are in the directory where the file resides):

```
?- [exercise0].
```

This is called *consulting* a file; the symbol `?-` is the system prompt. If a file has the extension `.pl`, the extension need not be typed in, as the file name is automatically completed. Hence, normally files with extension `.pl` indicate Prolog programs (this convention is unfortunately also used for Perl programs). If one prefers to use another extension than `.pl`, the full name of the file in between single quotes must be supplied. For example,

```
?- ['prog.pro'].
```

causes the program in the file `prog.pro` to be loaded. The above also works for the Windows environment.

It is also possible, although not very convenient, to type in a program in the Prolog system itself. This is accomplished by 'consulting' the file **user**.

```
?- [user].
|
```

The file **user** represents your keyboard. Its input can be terminated by `<cntrl>d`. In most cases, it is better to use an editor, such as emacs or some other editor, as these offer editing possibility far superior to the `user` interface. There is normally a special Prolog mode available for emacs users. Note that a program that has been typed in at `user` is lost when leaving the system.

In the next sections, it is assumed that the meaning of most Prolog predicates is known. The handout 'Introduction to Prolog' and any book about Prolog contain descriptions of the standard predicates.

3.2 PROLOG programs

A Prolog program consists of a sequence of *clauses*; a Prolog clause is either:

- a *fact*,
- a *query* or *goal*, or
- a *rule*.

Either or all of these may be empty (so, the simplest Prolog program is nothing!). Let us see what response we get from Prolog when trying to execute the empty program:

```
knuth-1 > pl
Welcome to SWI-Prolog (Version 3.2.9)
Copyright (c) 1993-1999 University of Amsterdam.  All rights reserved.
For help, use ?- help(Topic).  or ?- apropos(Word).
?- nothing.
[WARNING: Undefined predicate: ‘nothing/0’]
No
?-
```

The PROLOG *prompt* `?-` basically asks the user to type in a query. What is typed in, is shown in italised, bold face, and *includes the terminating dot.* PROLOG issues a warning, as it does not recogise the query `?- nothing`. However, this warning is not part of PROLOG, but a feature of SWI-PROLOG to help in program debugging. The response of PROLOG to this empty program is `No`, which means that this program has *failed.*[3]

Above, we have only employed the *declarative* terminology of PROLOG, which views PROLOG as a logical language. There also exists a *procedural* terminology, where facts and rules are called *procedure* (entries), and queries or goals are called (procedure) *calls.* In addition, we speak of the *head* and the *body* of a clause, which again is terminology that derives from the procedural interpretation of PROLOG. Hence, using this terminology, a PROLOG clause is defined as:

clause `::=` *head* `:-` *body*.

where body is defined as:

body `::=` *disjunction*`,`...`,`*disjunction*
disjunction `::=` *condition* | `(`*condition*`;` ⋯ `;`*condition*`)`

However, even when using the procedural terminology, it is difficult to ignore the logical basis of PROLOG, as the colon ‘`,`’ has the meaning of conjunction $\wedge$ (AND), and the semicolon ‘`;`’ has the meaning of disjunction $\vee$ (OR). We will use both terminologies in this practical to conform to the practice in the PROLOG and logic programming community.

4 PROLOG exercises

It is assumed you have copied the files to your directory as indicated in Section 3.

4.1 Meet The Dutch Royal Family: facts, queries and rules

Consider the following *facts* concerning the Dutch Royal Family. This family is so large, that I declare myself unable to unravel the details of the relationships among the members of the family, hence only the most obvious facts have been represented (but you are invited to exend the program!):

[3] Deep Philosophers will note that we have not really studied the effects of the *empty* program, as the tested program actually consisted of a single clause: the query `?- nothing.` When we type in nothing at all we get the real response to the empty program from PROLOG, which is `?-`. So, this is the real *nothing*!

```
% The facts about the Dutch Royal Family

mother(wilhelmina,juliana).
mother(juliana,beatrix).
mother(juliana,christina).
mother(juliana,irene).
mother(juliana,margriet).
mother(beatrix,friso).
mother(beatrix,alexander).
mother(beatrix,constantijn).
mother(emma,wilhelmina).

father(hendrik,juliana).
father(bernard,beatrix).
father(bernard,christina).
father(bernard,irene).
father(bernard,margriet).
father(claus,friso).
father(claus,alexander).
father(claus,constantijn).
father(willem,wilhelmina).

queen(beatrix).
queen(juliana).
queen(wilhelmina).
queen(emma).
king(willem).
```

In these facts, `mother` and `father` are called *predicates* or *functors*; elements such as `beatrix` are called *constants* or *atoms* (an atom is a constant that is not a number). The meaning of the fact `mother(juliana, beatrix)` is that Princess Juliana, the previous queen, *is the mother of* Queen Beatrix, the present queen. The meanings of the other `mother` and `father` facts are similar.

Now, if you are either a mother or father of a person, you are also seen as the *parent* of that person. This is a general rule, which even applies to members of the Royal Family. A rule, however, which uniquely applies to the Royal family is the one which defines kings and queens as rulers. These two principles can be formulated as four PROLOG *rules*:

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

ruler(X) :- queen(X).
ruler(X) :- king(X).
```

Note that the bodies of these rules consist of a single condition. `X` and `Y` in these rules are *variables*. They seem similar to atoms, but are required to start with an upper-case letter or underscore. Variables can be *bound* or *instantiated* to constants or other variables by a process called *matching* or *unification* (see below, Section 4.3).

Surely, one of the most popular questions asked by American tourists visiting the Netherlands is who predecesses who.[4] Now, this can be easily formulated into another two rules:

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

Note that these rules have multiple conditions constituting the body that must be satisfied in order for a rule to be satisfied. Also note the indentation of the body in comparison to the head; it makes PROLOG programs more readable.

The facts and rules above constitute a PROLOG program. The clauses of a PROLOG program are loaded into a part of the PROLOG system called the PROLOG *database*, which is rather confusing terminology, as a PROLOG system is not a database system. It can be invoked by means of *queries* or *goals*.

► *The complete* PROLOG *program is included in the file* **exercise0.pl**. *Consult this file, and query the program. If the* PROLOG *system returns with a response, you can enter a semicolon '*;*', and* PROLOG *will give an alternative solution. If you simply enter* `<return>` PROLOG *stops looking for alternatives. For example, enter the following queries:*

```
?- parent(beatrix,alexander).

?- parent(beatrix,X).

?- parent(beatrix,emma).

?- parent(X,Y), ruler(Y).

?- predecessor(X,beatrix).
```

Try to understand the answers returned by PROLOG by studying the program. (But be careful not to turn into a royalist!)

4.2 Lists and recursion

The list is one of the most popular data structures in PROLOG. A *list* is simply a sequence of elements, separated by commas and embraced by brackets:

$[e_1,\ e_2, \ldots,\ e_n]$

Each element e_i, $1 \leq i \leq n$, $n \geq 0$, can be an arbitrary PROLOG term, including a list.

Consider now the following list:

[4] Another one is how much their possessions are worth.

```
[a,b,c,d]
```

It consists of four elements, where each element in this case is an *atom* (recall that this is a constant symbol that is not a number).

The following list consists of three elements,

[e_1,e_2,e_3]

but is actually defined as

[e_1|[e_2|[e_3|[]]]]

where e_1 is called the *head* of the list, and [e_2|[e_3|[]]] is called its *tail*. Thus, as you already suspected, a list is a term of which the functor (from a logical point of view, the functor is now not a predicate, but a function symbol) has two arguments: the head and the tail. A list containing a single element [e] is really the following term: [e|[]]. A simplified notation as [e_1,e_2,e_3] is called *syntactic sugar*, as it makes dealing with lists easier. (Syntactic sugar makes your programs more tasty!)

Note that the following five lists

```
[a,b,c]
[a|[b,c]]
[a|[b|[c]]]
[a|[b|[c|[]]]]
[a,b|[c]]
```

are different representations of what essentially is the same list.

The handout 'Introduction to PROLOG' includes an example of a simple PROLOG program. This program identifies whether a particular element is a member of a set of elements, represented as a list. The following clauses are given (the predicate '`member`' was replaced by '`element`', as '`member`' is already included in most PROLOG systems):

```
/*1*/ element(X,[X|_]).
/*2*/ element(X,[_|Y]) :-
          element(X,Y).
```

Examples of queries to this program are:

```
?- element(d,[a,b,c,d,e,f]).
```

Note that the predicate `element` both occurs in the body of clause 2 and also in its head. Hence, this is an example of a *recursive rule*. In fact, you have seen recursion before when we considered Dutch royalty.

► *The two* PROLOG *clauses are included in the file* **exercise1.pl**. *Consult this file, and query the system. For example, try:*

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

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

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

Again, try to understand what is happening.

4.3 Unification and matching

An important mechanism underlying every PROLOG-interpreter is the *unification* of terms. Unification basically attempts to make two terms equal by substituting appropriate terms for the variables occurring in the two terms. In all cases, the so called *most general unifier* (mgu) is determined. A term can both be a condition and a head of a clause, but also a structure deeply nested into a condition or head.

To answer a query, the PROLOG interpreter starts with the first condition in the query, taking it as a procedure call. The PROLOG database is subsequently searched for a suitable entry to the called procedure; the search starts with the first clause in the database, and continues until a clause has been found which has a head that can be matched with the procedure call. A *match* between a head and a procedure call is obtained, if there exists a substitution for the variables occurring both in the head and in the procedure call, such that the two become (syntactically) equal after the substitution has been applied to them. Such a match exists:

- if the head and the procedure call contain the *same predicate*, and
- if the terms in *corresponding* argument positions after substitution of the variables are equal; one then also speaks of a match for argument positions.

Applying a substitution to a variable is called *instantiating* or *binding* the variable to a term. For example, the query:

```
?- parent(X,Y).
```

yields the following instantiations given the program in the file `exercise0.pl` discussed in Section 4.1:

```
X = wilhelmina
Y = juliana
```

A user can force unifying two terms by means of the infix predicate (functor) '`=`'. As an example, consider the following two terms:

`p(X,a,f(Y,g(b)))` and
`p(f(b),a,f(c,g(b)))`

The following query

```
?- p(X,a,f(Y,g(b))) = p(f(b),a,f(c,g(b))).
```

leads to the following substitutions:

```
X = f(b)
Y = c
```

because:

- the two functors of the two terms are equal (both `p`);
- to make the first arguments syntactically equal, the term `f(b)` has to be substituted for the variable `X`;

- the second arguments are already equal;
- in order to make the third arguments equal, we have to unify the terms `f(Y,g(b))` and `f(c,g(b))`. The functors of these two arguments are already equal, as are the second arguments of the terms. If we substitute the term `c` for the variable `Y`, the second arguments will also have become equal.

If two terms have been successfully unified, it is said that a *match* has been found; otherwise, there is no match.

▶ *Investigate whether the following pairs of terms match, and, if they do, which substitutions do make them equal?*

`p(f(X,g(b)),Y)` and `p(f(h(q),Y),g(b))`
`p(X,a,g(X))` and `p(f(b),a,g(b))`
`[[a]|[b]]` and `[X,Y]`
`p(X,f(X))` and `p(Y,Y)`

It is not possible to find a match for the last pair of terms. Yet, PROLOG is not able to recognise this fact. First, the matching algorithm substitutes `Y` for `X`:

```
X = Y
```

The second arguments of `p` would become equal if

```
Y = f(X)
```

This creates a cyclic binding between the variables `X` and `Y`; hence, it is not possible to unify these two terms. Many PROLOG interpreters, however, are not able to detect this fact, and get stuck in an infinite loop. What is missing in the implementation of the unification algorithm is known as the *occur-check*. It is often left out for efficiency reasons. To make a distinction between unification that includes the occur-check, and unification without, the latter is often called *matching*.

4.4 Backtracking

Backtracking is a mechanism in PROLOG that is offered to systematically search for solutions of a problem specified in terms of PROLOG clauses. A particular PROLOG specification may have more than one solution, and PROLOG normally tries to find one of those. When it is not possible to prove a subgoal given an already partially determined solution to a problem, PROLOG does undo all substitutions which were made to reach this subgoal, and alternative substitutions are tried.

The search space that is examined by the PROLOG system has the form of a tree. More insight into the structure of this space is obtained by a PROLOG program that itself defines a tree data structure:

```
/*1*/ branch(a,b).
/*2*/ branch(a,c).
/*3*/ branch(c,d).
/*4*/ branch(c,e).
```

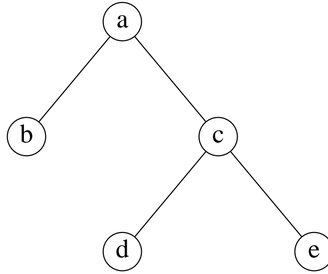

Figure 1: A binary tree.

```
/*5*/ path(X,X).
/*6*/ path(X,Y) :-
        branch(X,Z),
        path(Z,Y).
```

► *Load the file* **exercise2.pl** *into the* PROLOG *interpreter, and pose the following queries to the* PROLOG *database. Check the answers returned by* PROLOG.

```
?- path(a,d).
```

Figure 1 shows the tree corresponding to the program. This figure is useful for understanding the execution steps carried out by the PROLOG interpreter. In the next section, we shall consider a number of facilities offered by most PROLOG systems which assist in debugging programs.

4.5 Tracing and spying

PROLOG systems offer a number of facilities to 'debug' a program. Two of those techniques will be considered in this section. They are probably quite useful when working through the remainder of the exercises. A program can be debugged by using:

- the `trace` predicate, which makes sure that every execution step is shown;
- selective tracing of a program, which is made possible by so-called *spy points*. This is accomplished by the `spy` predicate.

By means of an example, we illustrate below how debugging with those two facilities works in practice.

After consulting a program, the trace facility can be switched on by the query:

```
?- trace.
```

For example, if after loading the file **exercise2.pl** as described in Section 4.4 into PROLOG, and after switching on the trace facility, and querying PROLOG:

```
| ?- trace.

Yes
| ?- path(a,e).
```

the following is shown:

```
   Call:  (  7) path(a,e) ?
```

The question marks indicates that the user may enter an option, to change the behaviour of the interpreter. A list of all possible options is obtained by entering the letter '*h*'. Each next execution step is shown by entering `<return>`:

```
?- path(a,e).
   Call:  (  7) path(a, e) ? creep
   Call:  (  8) branch(a, _L155) ? creep
   Exit:  (  8) branch(a, b) ? creep
   Call:  (  8) path(b, e) ? creep
   Call:  (  9) branch(b, _L178) ? creep
   Fail:  (  9) branch(b, _L178) ? creep
   Fail:  (  8) path(b, e) ? creep
   Redo:  (  8) branch(a, _L155) ? creep
   Exit:  (  8) branch(a, c) ? creep
   Call:  (  8) path(c, e) ? creep
   Call:  (  9) branch(c, _L167) ? creep
   Exit:  (  9) branch(c, d) ? creep
   Call:  (  9) path(d, e) ? creep
   Call:  ( 10) branch(d, _L190) ? creep
   Fail:  ( 10) branch(d, _L190) ? creep
   Fail:  (  9) path(d, e) ? creep
   Redo:  (  9) branch(c, _L167) ? creep
   Exit:  (  9) branch(c, e) ? creep
   Call:  (  9) path(e, e) ? creep
   Exit:  (  9) path(e, e) ? creep
   Exit:  (  8) path(c, e) ? creep
   Exit:  (  7) path(a, e) ? creep

Yes
?-
```

This information can be interpreted as follows. The number between parentheses represents the recursion level of the call (7 as a starting level); `Call` is followed by the subgoal that has to be proven; `Exit` indicates that the subgoal has been proven, en `Fail` indicates that proving the subgoal failed. If the program, as a consequence of a failed subgoal, backtracks to a previously considered subgoal, then this is indicated by `Redo`. Note that variables are renamed to unique internal names, such as `_L155`.

► *The above program is available in the file* **exercise2.pl**. *Now, experiment with the trace facility so that you understand it thoroughly.*

Switch off debug mode by `nodebug`.

Selective information of the execution is obtained by means of spy points. A spy point is indicated as follows:

```
spy(<specification>)
```

where <*specification*> is the name of a predicate or a name follows by the arity (number of arguments) of the predicate. For example:

```
spy(path).
spy(branch/2).
```

A spy point can be removed by the predicate: **nospy**:

```
nospy(path).
nospy(branch/2).
```

The structure of the information generated by the PROLOG interpreter in the case of spy points is almost identical to the information that is generated by using `trace`.

► *Experiment with spy points using the tree search algorithm.*

4.6 Variation of input-output parameters

An argument of a query can both function as an input and an output parameter, depending on the context of the query. This ensures that a program can be used for sometimes completely different, even opposite purposes, like construction and analysis. Consider the the following program, which is available in the file **exercise3.pl**:[5]

```
conc([],L,L).
conc([X|L1],L2,[X|L3]) :-
     conc(L1,L2,L3).
```

This program can be used to concatenate two lists, but, in fact, it can be used as well to split up a list into its parts.

► *Check the statement above by consulting the file* **exercise3.pl**,*; ask the following questions to the* PROLOG *interpreter (when the interpreter returns with a response, enter '*;*' in order to enforce backtracking, so that* PROLOG *will attempt to find alternative solutions.*

```
?- conc(X,Y,[a,b,c,d,e]).

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

The fact that it is possible to use the same program for such different purposes follows from the fact that PROLOG is essentially a declarative language based on a mathematical (logical) notion of computing.

4.7 The order of clauses and conditions

According to the principle of logic programming, neither the order of PROLOG clauses nor the order of conditions in the bodies of PROLOG clauses really matters. However, as a consequence of its fixed backtracking scheme this is not entirely the case for PROLOG. For example, it is possible that there does exist a solution to a problem according to the logical interpretation of

[5]The predicate `conc` is known in PROLOG as `append`, but as this is often a built-in, we have renamed `append` to `conc`.

a Prolog program, whereas Prolog is not able to find it. As a consequence, any Prolog programmer requires a good understanding of where and when order in and among clauses matters, and when not.

The program we are going to study in order to get a better understanding of the order issues in Prolog is available in the file **exercise4.pl**.

The following facts concern the relation `parent(X,Y)`, which expresses that X is a parent of Y:

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

▶ *Pose a number of queries to the program. For example, how do you find out what the names are of Tom's children? And, who are Bob's parents?*

The following two clauses recursively define that `X` is a predecessor of `Z`, `predecessor1(X,Z)`, if `X` is a parent of `Z` or if there is a person `Y`, such that `X` is a parent of `Y` and `Y` is a predecessor of `Z`.

```
/*1*/ predecessor1(X,Z) :-
          parent(X,Z).
/*2*/ predecessor1(X,Z) :-
          parent(X,Y),
          predecessor1(Y,Z).
```

▶ *Query this program, and find out how* Prolog *is able to give answers to your queries using* Prolog*'s debug facilities.*

There are three other declaratively correct specifications of the problem possible, which only differ with respect to order of clauses and conditions. Firstly, the order of the two clauses can be reversed. This results in the following program:

```
/*3*/ predecessor2(X,Z) :-
          parent(X,Y),
          predecessor2(Y,Z).
/*4*/ predecessor2(X,Z) :-
          parent(X,Z).
```

▶ *Ask again a number of queries, and compare the results with those obtained for the first program. Can you explain the differences?*

It is also possible to interchange the two conditions in the second clause, yielding the following program:

```
/*5*/ predecessor3(X,Z) :-
          parent(X,Z).
/*6*/ predecessor3(X,Z) :-
          predecessor3(X,Y),
          parent(Y,Z).
```

▶ *Which queries will this program be unable to answer?*

Finally, we can interchange clause 1 as well interchange both conditions in clause 2¿ The following result is then obtained:

```
/*7*/ predecessor4(X,Z) :-
          predecessor4(X,Y),
          parent(Y,Z).
/*8*/ predecessor4(X,Z) :-
          parent(X,Z).
```

▶ *Investigate which queries cannot be answered by this program, and try to explain why they cannot.*

4.8 Fail and cut

Some of the predicates which PROLOG offers can be used to control PROLOG's execution method in particular ways. Two important ones are the *fail* and the *cut* predicate. If the fail predicate is used as a condition, then this condition will always fail, regardless of anything else (hence, of course, the name of this predicate). Usually, the `fail` predicate is used to enforce backtracking in order to obtain or consider alternative solutions for a problem. Now, consider the program given in the file **exercise5.pl**:

```
element1(X,[X|_]).
element1(X,[_|Y]) :-
     element1(X,Y).

all_elements1(L) :-
     element1(X,L),
     write(X),
     nl,
     fail.
all_elements1(_).
```

By means of the clauses `all_elements1`, the elements of a list `L` are printed one by one. Note that the `element1` predicate is called in `all_elements1`.

▶ *Experiment with the* `all_elements1` *program. Do you understand why there is also a second* `all_elements1` *clause included?*

Next, the effects of the *cut* predicate on program execution are considered. Whereas the `fail` predicate *enforces* backtracking, the cut tries to prevent it. So, these two predicates are more or less complementary. Consider the following use of the cut:

```
element2(X,[X|_]) :- !.
element2(X,[_|Y]) :-
     element2(X,Y).

all_elements2(L) :-
     element2(X,L),
```

```
    write(X),
    nl,
    fail.
all_elements2(_).
```

▶ *Study the difference in behaviour between* `all_elements1` *and* `all_elements2`*, and try to explain this difference.*

4.9 A finite automaton

A finite automaton is a mathematical object, which is used to determine whether a particular string of symbols has the right structure or syntax. Finite automata are for example used to build compilers of programming languages, but also to build simulators.

An automaton is always in a one of a given set of *states*. One of those states is the *initial state*, another one is the *final state*. By examining one of the symbols in the given string, the automaton may go to another state, as specified by means of a *transition function*. If the automaton is in the final state after processing the string, it is said that the string is *accepted*; otherwise, it is said that the string has been *rejected*.

For the automaton that is considered here, the following set is defined:

$$S = \{s_1, s_2, s_3, s_4\}$$

Assume that s_1 is the initial state and that s_3 is the final state. The transition function is defined by means of the following productions:

$s_1 \rightarrow \mathrm{a}s_1$
$s_1 \rightarrow \mathrm{a}s_2$
$s_1 \rightarrow \mathrm{b}s_1$
$s_2 \rightarrow \mathrm{b}s_3$
$s_3 \rightarrow \mathrm{b}s_4$
$s_4 \rightarrow s_3$

The last transition is a silent transition: the automaton simply moves to another state without reading a symbol.

The transition function can be easily specified in PROLOG; for example the predicate `transition(s1,c,s2)` says that the automaton changes from state s_1 to s_2 after reading the symbol 'c'. The condition `silent(s1,s2)` represents a silent transition. The fact that s_3 is a silent transition is indicated by the predicate `end`. The transition function as defined above can therefore be defined in PROLOG as follows:

```
end(s3).

transition(s1,a,s1).
transition(s1,a,s2).
transition(s1,b,s1).
transition(s2,b,s3).
transition(s3,b,s4).

silent(s4,s3).
```

The following rules investigate whether a string is either or not accepted:

```
accept(S,[]) :-
    end(S).
accept(S,[X|Rest]) :-
    transition(S,X,S1),
    accept(S1,Rest).
accept(S,String) :-
    silent(S,S1),
    accept(S1,String).
```

★ (**P.1**) *Investigate using this program which input string with 3 and 4 symbols are accepted by the finite automaton. The program is included in the file* `exercise6.pl`*. The first argument of* `accept` *is the initial state; the second argument is a string, represented as a list of symbols.*

4.10 Sorting

As you already know, there are quite a number of different algorithms for sorting elements of a set (or multiset), assuming that a total order is defined on these elements. One of the simplest is the well-known *bubble sort* algorithm. In the bubble sort, two successive elements in a sequence of (here) numbers are examined. If they are in the right order, the algorithm proceeds to the next pair; otherwise, the two numbers are interchanged, and the algorithm proceeds by considering the next pair. This goes on until there are no elements left which need to be interchanged. Both worst and average case time complexity of the bubble sort is quadratic.

It is quite straightforward to express this algorithm in terms of a PROLOG program. Assuming that a set of elements is represented as a list, one possible program is:

```
bubsort(L,S) :-
     conc(X,[A,B|Y],L),
     B < A,
     conc(X,[B,A|Y],M),
     !,
     bubsort(M,S).
bubsort(L,L).

conc([],L,L).
conc([H|T],L,[H|V]) :-
     conc(T,L,V).
```

► *Investigate the workings of this program; it is available in file* **exercise7.pl**.

4.11 A relational database in PROLOG

Basically, PROLOG programs are about defining relation among objects in a domain. As storing data as instances of relations is the essential feature, you are probably not surprised to learn the PROLOG is very good in performing the sort of operations on data, which many people normally associate with a relational database management system.

For example, consider `salary(scale,amount)` which is the sort of relation defined in a relational data model. Recall that `scale` and `amount` are referred to as *attributes*, whereas specific instances of this relation are called *tuples*. A relational database contains tuples defined according to the data model. Retrieving data stored in the database according to particular constraints on the attributes is called *querying* the database.

The file **exercise8.pl** includes an example of a database, with the following data model (Note that the data model is really implicit in this case, which is because we simply make use of standard PROLOG facilities without implementing anything extra):

- employee(*name*,*department_number*,*scale*)
- department(*department_number*,*name_department*)
- salary(*scale*,*amount*)

The database (PROLOG database in this case) included the following tuples (facts), again included in **exercise8.pl**:

```
employee(mcardon,1,5).
employee(treeman,2,3).
employee(chapman,1,2).
employee(claessen,4,1).
employee(petersen,5,8).
employee(cohn,1,7).
employee(duffy,1,9).

department(1,board).
department(2,human_resources).
department(3,production).
department(4,technical_services).
department(5,administration).

salary(1,1000).
salary(2,1500).
salary(3,2000).
salary(4,2500).
salary(5,3000).
salary(6,3500).
salary(7,4000).
salary(8,4500).
salary(9,5000).
```

Relational database theory offers a number of useful operations which can be carried out to extract information from a given database. The result of these operations is a new relation, i.e. a new set of tuples. Typical examples of such database operations are *selection*, *projection* and the *join*. Below we shall discuss simple PROLOG implementations of the selection, projection and join, just to give you an impression of how such programs may look like. The implementation of a more complete relational database package would, of course, be more involving than would be realistic for this practical. We will make use of an SQL-like notation (as you know, SQL is the most popular query language in relational databases).

The purpose of a *selection* is to request tuples which fulfil certain conditions with respect to their attribute values. Consider, for example, the query: 'Select all employees from department 1 with a salary higher than scale 2.' In SQL-like syntax, this could be stated as follows:

```
SELECT * FROM employee WHERE department_number = 1 AND
         scale > 2.
```

Expressed as a PROLOG query, the purpose is to select tuples satisfying the following condition:

```
?- employee(Name,Department_N,Scale),
   Department_N = 1,Scale > 2.
```

▶ *Consult the file* `exercise8.pl` *and enter the* PROLOG *query given above.*

The following output is produced:

```
Name = mcardon
Department_N = 1
Scale = 5.
```

By entering the '`;`' operator (the logical $\vee$ connective) is it again possible to find out whether there are any alternative solutions, as PROLOG will then backtrack, and will check whether there are any other employees satisfying the conditions. Note that, in fact, we have not yet produced a functionally complete implementation of the selection operator, as only a single tuple is selected from the database, but we are nearly there. Recall the trick we considered before, where we made use of the `fail` predicate in order to enforce backtracking. Here we use this trick again. This gives us the following implementation of the `selection` operator:

```
selection(X,Y) :-
     call(X),
     call(Y),
     write(X),
     nl,
     fail.
selection(_,_).
```

The predicate `call` generates a subgoal determined by the binding of its variable argument. It is assumed that the first argument of `selection` represents the relation from which tuples are to be selected; the second argument contains the set of selection conditions which should be satisfied by the selected tuples. For example, all employees in department 1 with a salary scale higher than 2 are selected by the following query:

```
?- selection(employee(Name, Department_N, Scale),
   (Departments_N = 1,Scale > 2)).
```

▶ *Design a* PROLOG *query to select those employees who earn between £3,000 and £4,500 per month? Also select all employees who are in salary scale 1, and working in department 1.*

The *projection* operator in relational database theory is used to select some attributes (or columns in the corresponding tables) from a relation. For example, the query 'give for all employees only the name and scale', in SQL-like notation:

```
SELECT name,scale FROM employee.
```

However, the PROLOG query does again give rise to the selection of a single tuple:

```
?- employee(Name,_,Scale).
```

But, as you already know, by using the '`;`' operator the other tuples can be selected as well.

▶ *Type in the above-given* PROLOG *query. Can you explain why we use here the anonymous variable _ ?*

As you may have guessed, the resulting PROLOG implementation of the `projection` operator is quite similar to the implementation of the `selection` operator discussed above.

```
projection(X,Y) :-
     call(X),
     write(Y),
     nl,
     fail.
projection(_,_).
```

The first argument of `projection` should be the relation that is being projected; the second argument lists the attributes on which the relationship must be projected. For example, name and scale of employees are obtained as follows:

```
?- projection(employee(Name, Department_N, Scale),
   (Name, Scale)).
```

Note that this is still not a complete implementation of the `projection`, as the resulting relation may not include doubles. As incorporating this in the program as well would make it quite a bit more complicated, hardly worth the effort for small databases, we leave the program as it is.

▶ *Experiment with the projection program until you fully understand how it works.*

It is now quite straightforward to combine selection and projection. This would imply not only specifying the attributes used in projecting a relation, but also giving the conditions on the attributes which need to be satisfied by the selected tuples. Consider, for example, the following query 'Print the name and scale of those employees in department 1, with a salary higher than scale 2'. In SQL-like notation:

```
SELECT name,scale FROM employee WHERE department_number = 1
       AND scale > 2.
```

▶ *Define yourself the predicate* `sel_pro` *which is meant to combine selection and projection.*

The final database operation which we will consider is the *join*. The join simply merges tuples in relations based on a *join condition*. Suppose that one would like to obtain a list of employees, with for every employee the salary included. For this purpose, we need information from two different relations: `employee` and `salary`. For every employee we need to find the salary corresponding to the salary scale. Hence, the join condition in this case is equality of tuples in the two relations concerning the attribute `scale`. In SQL-like notation:

```
JOIN employee WITH salary WHERE
     employee.scale = salary.scale
```

The following two clauses offer an implementation of the `join` operator:

```
join(X,Y,Z) :-
     call(X),
     call(Y),
     call(Z),
     write(X),
     write(Y),
     nl,
     fail.
join(_,_,_).
```

The first and second arguments represent relations; the third argument is used to specify join conditions. For example:

```
?- join(employee(Name,Department_N,Scale1),
   salary(Scale2,Amount),
   (Scale1 = Scale2)).
```

will yield the requested result.

★ (**P.2**) *Express in* PROLOG *a join between the relations* `employee` *and* `department`*, and inspect the result.*

4.12 Natural language processing

From its inception, PROLOG has been enormously popular with researchers in the field of natural language processing. Due to its declarative nature, the language is both suitable for the development of programs that analyse sentences as well as for language generation. In this case, we only consider a very simplistic example of a program, as natural language processing is a field in its own right, involving many special issues.

The following program is given in the file **exercise9.pl**:

```
sentence(X) :- noun_phrase(X,Y),
               verb_phrase(Y,[]).

noun_phrase([X|Y],Y) :-
          noun(X).
noun_phrase([X1,X2,X3|Y],Y) :-
          article(X1),
```

```
            adjective(X2),
            noun(X3).

verb_phrase([X|Y],Z) :-
            verb(X),
            noun_phrase(Y,Z).
```

- ► *Develop a small dictionary that can be used to generate sentences.*
- ► *Why are the predicate* `noun_phrase` *and* `verb_phrase` *supplied with a second argument? Is this program also capable of generating sentences? We could also have made use of the* `conc` *predicate in developing this program (see* **exercise3.pl***). How would this have affected the program?*