Prolog Step-by-Step

Graeme Ritchie

October, 2002

School of Informatics, University of Edinburgh.

Acknowledgements: Sections 1, 2 are taken entirely from *Quick Prolog* by Dave Robertson, Mandy Haggith, Louise Pryor & Geraint Wiggins.

Contents

1	Introduction 2 1.1 Software 2			
2	How Do You Learn A Programming Language?	2		
	2.1 The gist of Prolog	3		
3	Getting Started	4		
	3.1 Starting Prolog	4		
	 3.2 Simple goals	4		
4	Defining your own procedures	5		
	 4.1 Consulting files	5 5		
	4.2 Defining simple procedures	5 6		
	4.4 Adding comments to your program	6		
_		_		
5	Variables 5.1 Sharing information within a clause	7 7		
	5.1 Sharing information within a clause	7		
	5.2 Clauses in order	8		
	5.4 Re-trying goals	8		
	TT (0		
6	Tests 6.1 Success and conjunction	9 9		
	6.2 Selecting clauses using tests	10		
		10		
7	Returning results by instantiation11			
8	"Facts"	12		
	8.1 Clauses without bodies	12		
	 8.2 Unit clauses as "facts" 8.3 Facts and rules 	12 14		
	6.5 Facts and fulles	14		
9	Simple tracing 15			
10	Backtracking in more detail	16		
11	Summary so far	17		
12	Unification and Equality 18			
13	3 Arithmetic 19			
14	Terminology and notation 21			
	5 Lists 22			
10				
16	General structures	23		
	16.1 Clustering items	24		
	16.2 Arithmetic expressions as structures 16.3 Lists as structures	25 25		
		23		
17	Circularity and loops	26		

1 Introduction

These notes are not intended to be a complete introduction to the art of programming in Prolog. There are many good books available that cover the content of this module, and there is no point in duplicating the authors' efforts. The recommended text book is:

[Clocksin & Mellish 94] A good basic introductory text.

Other books that may be useful include:

- [Bratko 90] 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;
- [O'Keefe 90] A book for people who are well acquainted with Prolog; have done a fair chunk of programming; and were wondering what more there was to know;¹
- [Ross 89] A good book for advanced programmers. It contains lots of useful ideas about how to tackle substantial programming problems;
- [Shoham 94] Good on AI applications of Prolog for those with a knowledge of the basics;
- [Sterling & Shapiro 94] Possibly the best general Prolog book around, but definitely not an introduction, especially if you don't have much programming experience.

1.1 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.

2 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 Edinburgh² 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?

¹The answer is "A lot".

²In fact it was conceived in Marseilles, though not as a general purpose programming language

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;
- 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.

2.1 The gist of Prolog

Some facts about Prolog:

- Prolog is a *high-level logic programming* language (PROgramming in LOGic);
- 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 Getting Started

3.1 Starting Prolog

Prolog is an *interactive* programming language. This means that you can control the Prolog system by sitting at a terminal typing in commands which the system can execute more or less immediately, giving replies to you directly. In this respect it is like BASIC or LISP, but unlike C or Java.

On any department workstation, type

sicstus

This will start up the Prolog system, which will print a short message (stating the version of SICStus and the date the system was built), and then the Prolog "prompt" (the indication that the system expects some input) which looks like this:

| ?-

If you try just hitting the RETURN key (the normal end-of-line key), the prompt will be repeated, but nothing much else will happen.

3.2 Simple goals

Prolog is sent into action by giving it a *goal*, which we can think of for the moment just as a simple command. Simple Prolog commands are formed by a name (the *predicate* name), followed by brackets round the data item(s) involved (the *arguments*). The Prolog command for simple printing is write. Try typing in, directly at the prompt, this simple program, hitting RETURN at the end:

write('hello').

Prolog is extremely fussy about punctuation, so get it right – no spaces before the left-bracket, only singlequote marks (apostrophes) leaning as shown, and a full-stop at the end. In executing this goal, the system should print out

hello yes

followed by the system prompt again.

Prolog generally prints yes when it has successfully completed a command, and no if it fails to.

3.3 Conjoining commands

Several commands can be typed one after another, provided *commas* are put between them (and a *full stop* after the last one). For example

write('hello'), nl, write('master').

Here, nl is a built-in procedure which causes a new line to be output. Another built-in command is tab, which puts out a given number of spaces. Try:

```
write('oh'), tab(3)
write('what'),tab(3)
write('a'),tab(3)
write('beautiful'),tab(3)
write('morning').
```

(For your own convenience, you can carry on typing on to another line if the list of commands is very long - nothing will happen until the full stop is read by the system.)

4 Defining your own procedures

4.1 Consulting files

Although typing in simple goals allows you to see Prolog in action, and is the way in which you get Prolog to execute your program once you have written it, it is not the normal way of building a program. To show you how to write, load and test programs, we first need to digress into the use of program files.

Suppose you have thought up some Prolog procedure definitions (we shall discuss that shortly). The next step is to write these in an ordinary text file using a suitable text editor (e.g. emacs). Let us assume that you have made such a file, called progl.pl. (It is normal to have Prolog program files end in ".pl".)

Then, inside Prolog, the goal

consult(prog1).

will cause the file to be loaded into the Prolog system ("consulted", in Prolog terminology). Then any definitions you have put in that file will be available, just as the basic Prolog definition of write is.

If what you wrote in the file is notationally correct Prolog, the system will respond with yes to indicate that the file has been loaded successfully (i.e. the goal consult(progl) has been achieved).

If you made mistakes in the file, there will be error messages, and you will have to go back and edit the file before trying the consult again.

4.2 Defining simple procedures

A simple procedure has two major parts - the *head* (sometimes called the "left hand side") and the *body* (sometimes called the "right hand side"). The head gives the name to the procedure, and the body is the sequence of commands (i.e. one or more commands, separated by commas and ending with a full stop) which constitute "obeying the procedure". The two parts are written on either side of the special symbol

:-

For example:

```
salute :-
write('salutations,'), tab(2), write('master').
```

This defines a procedure whose head is

salute

and whose body is

write('salutations,'), tab(2), write('master').

If we load this definition into the Prolog system using consult (see above for details), then we can then use salute as a Prolog command.

Create a file containing the above definition. Start up SICStus. Load the procedure in (consult). Try typing

salute.

(Remember the fullstop!). The system should carry out the body of the procedure.

Prolog procedures are often referred to as *predicates* or as *rules*. The latter name for them reflects the idea that the left-hand side specifies *how* to carry out the right-hand side. We will tend to treat the terms "procedure" and "predicate" as meaning much the same thing, but will try to avoid using "rule" as it sometimes is taken to mean a single clause within a procedure (see below), rather than the whole procedure.

4.3 Making choices

The salute example procedure has only one sequence of commands as its body. In Prolog, a rule can have *several* sequences of commands in its definition – each sequence for a separate situation. That is, the whole definition is made up of several *clauses*, each clause having its own head and body. When the rule (procedure) is used, the system selects one of these definitions and uses it. The system does this selection by searching through all the definitions (clauses) listed for that predicate until it finds one for which the head (left-hand side) matches the current goal. (The search is done in the order you have provided the clauses, from first to last. That may not be obvious in these very simple examples, but it soon becomes very important in more complex predicates.)

For example, suppose we want to have a rule which greets different people in a different way. We can do this by specifying actual values in the heads of the clauses. The system will, when trying to greet, select the clause corresponding to the name used.

```
greet(hamish):-
    write('How are you doing, pal?').
greet(amelia):-
    write('How awfully nice to see you!').
greet(mike):-
    write('Hi there').
```

Try typing this into a file, loading it into Prolog, and invoking the rule, using different names as the argument; for the moment, stick to lower-case letters for names (no capitals!), with no punctuation in the names. What if the greet rule is used for someone not named in any of these three clauses? Try it. The system will say

no

meaning that it was unable to carry out your task, as greet(anastasia) (or whatever you typed in) did not match any of the clauses for greet. This is an illustration of an important point about Prolog – there are *two* ways of Prolog not doing what you asked. If what you have typed in causes no Prolog errors, then the system will simply print no if it is unable to fulfil the goal. If, on the other hand, your goal triggers an error, you will get an error message about this. To see the latter, try typing in

say('hello').

In this case, you will get an error message:

{EXISTENCE ERROR: say(hello): procedure user:say/1 does not exist}

This is because there is no Prolog command say, and trying to use a non-existent predicate is an error.

4.4 Adding comments to your program

The Prolog comment characters are

%

and the pair of "brackets"

/* */

At the end of any line, % means the rest of the line is a comment (ignored by the Prolog system), thus:

% This defines a simple predicate

```
salute :-
write('Hi,'), tab(2), write('pal!').
```

Multiple line comments can be enclosed between /* and */, thus:

```
/* Predicate name : salute
   Number of arguments: NONE
   Effects: prints a message
*/
salute :-
   write('Hi,'), tab(2), write('pal!').
```

5 Variables

As in most programming languages, there can be named variables which take on values. Variables in Prolog behave slightly differently from variables in many other languages, although this will not be obvious in our simpler examples.

Any symbol which starts with a capital (upper case) letter is treated by Prolog as a variable.

5.1 Sharing information within a clause

Suppose we wanted to generalise our greet procedure so that it would work for any name, and would repeat the name of the person in the printed greeting. The following would achieve this:

```
greet(Name):-
    write('Hullo, '), write(Name).
```

Here, Name is a variable. If we try out this clause on its own (i.e. without the earlier clauses for greet), we get this effect:

```
?- greet(alphonse).
Hullo, alphonse
yes
```

What happens is that when Prolog tries to match the goal greet(alphonse) against the stored definitions, it compares greet(alphonse) with the head of the only available clause, the definition of greet(Name). Two terms such as this match if they have the same predicate name (here, greet) and the same number of arguments (here, one) and if the items in those argument positions match (here, alphonse and Name). A variable such as Name can, if it has not already matched something, match *any* item. So Name matches alphonse and, as a result of this, the value alphonse becomes associated with Name – the variable is *bound* to that value.

When Prolog (having successfully matched the head of the clause against the current goal, and therefore chosen that clause) goes on to carry out the body of the clause, any occurrences of Name are now deemed to be alphonse.

5.2 Clauses in order

Suppose we now construct a file (say, prog2.pl) which contains all the clauses for greet, in this order:

```
greet(hamish):-
    write('How are you doing, pal?').
greet(amelia):-
    write('How awfully nice to see you!').
greet(mike):-
    write('Hi there').
greet(Name):-
    write('Hullo, '), write(Name).
```

Once we have consulted this file, we then get the following behaviour:

```
?- greet(hamish).
How are you doing, pal?
yes
?- greet(daisy).
Hullo, daisy
yes
?- greet(amelia).
How awfully nice to see you!
yes
```

Since Prolog searches from the top of the clauses when trying to find a match, the specific clauses at the start are found when they are applicable, and the general clause at the end (with the variable) is reached only when nothing else matches.

5.3 Variables in goals

In the above example, the variable was in the predicate-definition, and the goal we supplied contained a non-variable (a *constant*). It is also possible to put a variable in a goal, and have it matched against whatever can be found in the predicate definitions. Suppose we have loaded (consulted) the file progl above, which has the three clauses for greet but not the clause with the variable. Then if we type a goal which has a variable as its argument, this is what happens:

```
?- greet(Person).
How are you doing, pal?
Person = hamish ?
```

The system has tried to find a clause whose head matches the goal, and the first clause fulfils this requirement: greet(Person) matches greet(hamish), with the variable Person getting bound to hamish. Hence the first clause is selected and its body (right hand side) is performed; you can see that this has happened, because the write message appears.

Since the variable was in a goal which came from the user (you), the Prolog interface supplies you with information about the successful binding of the variable.

This does not happen every time a variable is bound, only when the variable is in the goal you supplied.

5.4 Re-trying goals

In the situation just described, notice that Prolog has not printed out either yes (indicating success) or no (for failure) – it is waiting for you to respond. If you hit RETURN, it will respond with yes to indicate successful completion. However, the interface allows you to ask the system to see if any other bindings for the variable are possible. To do this, type a semi-colon (that is, ";") instead of RETURN at this stage. The printout should now look like this:

```
?- greet(Person).
How are you doing, pal?
Person = hamish ? ;
How awfully nice to see you!
Person = amelia ?
```

The system has discarded the binding it had for Person, and moved on to see if some other clause can be found which matches. It does succeed, and so returns to the user to ask if this is sufficient (if so, press RETURN, and get the yes message) or whether more options should be tried. If you keep pressing the semi-colon in response to these prompts, the complete sequence will look like this:

```
?- greet(Person).
How are you doing, pal?
Person = hamish ? ;
How awfully nice to see you!
Person = amelia ?;
Hi there
Person = mike ? ;
```

no

Prolog tries all clauses in turn, and when the third semi-colon asks for more options, it returns the failure message no – there are no more clauses to try. This re-trying on failure is known as *backtracking*, and is a very important concept in Prolog. What we have illustrated here is an extremely simple example of backtracking – much more complex examples will appear shortly.

Notice that we said that the system *discarded* the first binding it had for the variable before going on to look for other options. This is because a variable can be bound to at most one constant (specific value) at a time, so the variable has to become *unbound* before further bindings were possible.

We will sometimes use the term *instantiated* to describe a variable which is bound to a non-variable (specific) and *uninstantiated* if it is not.

6 Tests

6.1 Success and conjunction

We have talked of a goal 'failing' or 'succeeding'. This distinction is routinely used to get an effect which in a conventional language would be regarded as computing the truth or falsity of some condition. For example, Prolog has the built-in operator < which tests if one number is less than another. Try typing in some goals using this symbol:

```
?- 5 < 7.
yes
?- 9 < 7.
no
?- 7 < 7.
no</pre>
```

Here the yes/no answer (indicating success or failure) communicates what is effectively the truth of the condition stated in the goal. If you put several tests joined by commas, Prolog will treat this as a sequence of goals to be carried out (just as demonstrated already).

?-3 < 7, 2 < 4, 10 < 12.

yes

Because all of these goals succeeded, the sequence as a whole succeeded. If one of the goals fails, as here:

?-3 < 7, 8 < 4, 10 < 15.

no

then the whole sequence fails. In this way, the normal left-to-right meaning of the comma between goals achieves the effect of a logical 'AND' (conjunction): the whole sequence is true (successful) if and only if every item in the sequence is true (successful). In fact, Prolog stops working through the sequence when it reaches a failure, so in this example the goal 10 < 15 would not have been attempted. This makes it slightly less like a real logical operator, as the goals which are skipped might have errors embedded in them, but this would not come to light.

6.2 Selecting clauses using tests

So far, the system has dealt with a multi-clause rule by simply taking the first clause for which the head matches its current task. Either a head matched (in which case the body of that clause was used, complete with any variable settings established by the match) or it did not match (in which case that entire clause was skipped). Further subtleties of clause selection are possible. In particular, it is possible for the system to find a matching clause, embark upon executing the body, and then find a goal which fails. In this case, it abandons the clause in mid-execution and resumes the search for another clause whose head matches the current task. This arrangement allows us to introduce a finer selection process for clauses by putting a test of some kind at the front of the body. If the test "succeeds" (i.e. turns out to be true) then the execution proceeds; if it "fails" (i.e. turns out to be false), then the execution of that whole clause is abandoned and the system searches for a later matching clause within the predicate definition. Suppose we want a rule which takes two parameters (arguments) which are numbers, and prints out the bigger of the two. Using the Prolog "less-than" symbol "<", we can write the rule in two clauses thus:

There are several points worth a second glance here. You might think the 3rd clause should really have the test "N=M" at the start of the body (there is an "equal" sign in Prolog, though it does not behave as simple equality – see later). We could add that check, but it would not alter the behaviour of the rule. Remember that the system works down through the clauses until the first one which succeeds (more accurately, the first one where the head matches and the body doesn't fail). The third clause will therefore be considered by the system *only if* it has skipped the first two; that is, only if N is not less than M, and M is not less than N. That leaves only the possibility that M=N. (Assuming you have invoked the procedure with two numbers as arguments - if you use non-numbers, anything might happen!). So if the third clause is being considered, N must be the same as M.

Just to reassure yourself that the system really does embark upon the earlier clauses and give up (in the M = N situation), try altering the first two clauses so that they contain small printing commands to indicate what is happening. That is:

```
bigger(N, M):-
    write('Trying clause 1'), nl,
    N < M, write('Bigger number is '),tab(1), write(M).
bigger(N, M) :-
    write('Trying clause 2'), nl,
    M < N, write('Bigger number is '), tab(1), write(N).
bigger(N, M) :-
    write('Numbers are the same'), nl.</pre>
```

If you load up this version of the rule instead of the old one, and type in

```
?- bigger(8,8).
you should get the output
Trying clause 1
Trying clause 2
Numbers are the same
```

yes

7 Returning results by instantiation

Although we have been using illustrative predicates which print out messages, as a simple way of showing what was happening inside the procedures, printing is not the normal route for information to be passed back from a procedure. Printing is useful in certain situation (warning messages, debugging, etc.) but generally results are passed back *as the values bound to variables*. To demonstrate this, consider a slightly more realistic (non-printing) version of the bigger predicate, which we will call larger.

In larger, the chosen value becomes the value of the 3rd argument. It works as follows:

```
?- larger(8,3,Result).
Result = 8
yes
?- larger(2,5,Result).
Result = 5
yes
?- larger(6,6,Result).
Result = 6
yes
```

What happens here is that what we think of as the "input" or "initial" values are supplied as actual values (8, 3), and the available place for putting the result is an uninstantiated (unbound) variable, Result. The matching of this goal against the first clause causes N to be bound to 8, M to 3, and Result to M. That clause then fails (as N < M fails for N bound to 8 and M bound to 3), so the second clause is tried. This makes the bindings as follows: N to 8, M to 3, and Result to N. The body then succeeds, and the goal is therefore completed. Since Result is bound to N, and N is bound to 8, Result is treated as being bound to 8, which the Prolog user-interface reports as usual.

There are various relevant points about variable-matching in Prolog:

- a variable can be bound to another variable;
- if two variables are bound to each other, and one of them is (or becomes) bound to another value, then both variables are regarded as bound to that other value;
- hence, if two variables are bound to each other, they cannot then become bound to different values (except for other variables).
- binding between variables is symmetric, in the sense that if A is bound to B then B is bound to A.

• any number of variables can be bound to each other, forming a whole set of variables which share whatever one of them may become bound to;

The pattern demonstrated in larger is extremely typical of the structure of a Prolog procedure, with one (or more) of the arguments acting as the result(s). Ensuring that two variables share a binding, either by the matching of the head or otherwise, is also very common.

8 "Facts"

(See Clocksin & Mellish 1.1, 1.2, 1.3)

8.1 Clauses without bodies

It is possible to have a Prolog clause where the right hand side (body) is empty, indicating that nothing needs to be done. This is not as useless or as unusual as it sounds; in fact, it is an extremely common type of clause.

Let us go back to the greet program that we developed earlier. Suppose that for some argument, nasty, we wanted no action to be taken at all. You might think that we could write a clause as follows:

```
greet(nasty):-
```

That is, with an empty right-hand side. Unfortunately, this is not permitted, notationally, in Prolog. The good news is that there is an even shorter way of writing an empty right-hand side – simply put a full-stop right after the head (left-hand side), omitting the :- symbol:

greet(nasty).

This then allows the following interactions:

```
?- greet(nasty).
```

yes

```
?- greet(Anyone).
```

```
Anyone = nasty ?
```

yes

That is, when the system is trying to satisfy a goal, if it finds a clause whose head matches the goal, and there is no right hand side to that clause, that counts as success. **Clauses like this are known as** *unit clauses***.**

8.2 Unit clauses as "facts"

Unit clauses are widely used in Prolog to represent basic information, including special cases, or information that might in other languages be represented in a table. These are often known informally as *facts*, for reasons which should become clear.

Although we have used simple numeric examples above, Prolog is most typically used for manipulating *symbolic* data: letters, words, structures such as lists which contain other symbolic data, etc.

We will now show a simple program in which data about an imaginary family is stored, allowing goals which ask about family relationships to be fulfilled.

```
man(paul).
man(david).
man(peter).
woman(louise).
woman(helen).
woman(mandy).
wifeof(paul, louise).
wifeof(peter, helen).
sonof(paul, peter).
daughterof(peter, mandy).
We can then "query" this "database":
?- man(peter).
yes
?- man(louise).
no
?- woman(Someone).
Someone = louise;
Someone = helen;
Someone = mandy;
no
?- wifeof(paul, Hiswife).
Hiswife = louise
yes
?- wifeof(Herhusband, louise).
Herhusband = paul
yes
?- daughterof(Father, mandy).
Father = peter
yes
```

Notice what is happening in the two wifeof queries. We are chosing in the first query to treat the first

argument as the "input", by supplying an actual value, and leaving the second argument free to be the result, by leaving it uninstantiated. In the last of the examples, we do it the other way round. This shows that there is nothing explicit in the predicate definitions to say what is to be an input argument and which is to be an output result – it all depends on the information supplied in the goals, and how this affects the matching. The Prolog user-interface chooses to print out the values for the variables that were uninstantiated in the original goal, which gives the impression that these have some special status, but that is misleading.

Notice also that queries involving these predicates, but non-existent clauses, will give failure, as usual:

```
?- sonof(david, Son).
```

no

This demonstrates what is sometimes referred to as the *closed world assumption* in Prolog – if the system doesn't know something, it gives the same answer as if that item were untrue. This contrasts with queries about unknown predicates:

```
?- cousin(david, Son).
{EXISTENCE ERROR: cousin(david,_40): procedure user:cousin/2 does not exist}
```

Notice that when the Prolog interface has to print out a variable, such as Son here, it uses its own internal form, not the name in the program. These Prolog-internal variable names consist of an underscore followed by a number. You will need to be able to keep track of this notation when you use the debugger (below).

8.3 Facts and rules

The unit clauses ("facts") given above can be combined with non-unit clauses such as the following:

```
parentof(Person1, Person2):-
    daughterof(Person1, Person2).
```

```
parentof(Person1, Person2):-
        sonof(Person1, Person2).
```

This allows queries like this:

```
?- parentof(peter, Child).
Child = mandy
yes
?- parentof(paul, peter).
yes
?- parentof(louise, peter).
no
?- parentof(Parent, peter).
Parent = paul
```

yes

Again, we can choose to treat either argument as the "result" by leaving it uninstantiated. Or we can leave both uninstantiated:

```
?- parentof(Parent, Child).
Child = mandy
Parent = peter
```

yes

Here, the peter-mandy link has been picked up first (rather than the equally valid paul-peter connection), because the first clause of parent-of checks the daughterof data; the fact that the sonof clauses precede the daughterof ones is irrelevant. If we had requested backtracking (by typing a semi-colon in response to the first result), the peter-mandy values would have been found.

9 Simple tracing

In finding out why your program does not act as you expect (a common situation, unfortunately), the spy mechanism can be handy. If you want to watch what happens inside a particular predicate, type "spy" with the predicate name as an argument. This will cause the system to pause when it tries this predicate, allowing you various options to find out more.

The simplest option is to hit RETURN, which means "go on to the next step then pause there". We will use this in the example below.

Suppose you have loaded the larger predicate given earlier. Then the following is possible:

```
?- spy(larger).
{The debugger will first leap -- showing spypoints (debug)}
{Spypoint placed on user:larger/3}
yes
{debug}
?- larger(8,9, Output).
+ 1 1 Call: larger(8,9,_195) ?
  2 2 Call: 8<9 ?
     2 Exit: 8<9 ?
  2
 + 1 1 Exit: larger(8,9,9) ?
Output = 9 ?
ves
{debug}
?- larger(9,8, Result).
 + 1 1 Call: larger(9,8,_195) ?
  2 2 Call: 9<8 ?
  2 2 Fail: 9<8 ?
  2 2 Call: 8<9 ?
  2 2 Exit: 8<9 ?
 + 1 1 Exit: larger(9,8,9) ?
Result = 9 ?
```

yes

To turn this off, type "nospy" with the predicate name as argument (e.g. nospy(larger)).

10 Backtracking in more detail

(See Clocksin & Mellish 1.4)

We mentioned earlier that Prolog, on failure, re-tries previous goals to see if success can be found by choosing a different clause for some procedure. It can be useful to work through a more complicated example of this happening, with variable bindings happening and being discarded along the way.

If we add the various "family" definitions above the following procedure:

we can have queries such as:

```
?- grandparent(paul, GrandChild).
GrandChild = mandy
```

yes

```
?- grandparent(GrandDad, mandy).
```

Grandad = paul

yes

In the grandparent procedure, a variable appears which is neither in the original goal nor in final result, Another. We do not care what its value is, as long as some value can be found which allows the two parent goals to succeed. Suppose we give Prolog the goal grandparent(Oldone, Youngone). It will try to satisfy its goals as follows:

```
try to find GRANDPARENT values:
  try to find PARENT values, with PERSON1, PERSON2 unbound
     try to find DAUGHTEROF values with PERSON1, PERSON2 unbound
     succeed with PETER, MANDY
  succeed with PERSON1 = PETER, PERSON2 = MANDY
       (hence OLDPERSON = PETER, ANOTHER = MANDY)
  try to find PARENT values for ANOTHER = MANDY, YOUNGERPERSON unbound
     try to find DAUGHTEROF values for PERSON1 = MANDY, PERSON2 unbound
     FAIL!
     try other clause of PARENT, looking for SONOF values with
             PERSON1 = MANDY, PERSON2 unbound
     FAIL!
     BACKTRACK -
     reconsider previous goal (DAUGHTEROF with PERSON1, PERSON2 unbound)
     FAIL! (no other clauses for DAUGHTEROF)
  try later clauses for previous goal
              (PARENTOF with with PERSON1, PERSON2 unbound)
     try to find SONOF values with PERSON1, PERSON2 unbound
     succeed with PAUL, PETER
  succeed with PERSON1 = PAUL, PERSON2 = PETER
           (hence OLDPERSON = PAUL, ANOTHER = PETER)
  try to find PARENT values for ANOTHER = PETER, YOUNGERPERSON unbound
     try to find DAUGHTEROF values for PERSON1 = PETER, PERSON2 unbound
     succeed with PETER, MANDY
  succeed with PERSON1 = PETER, PERSON2 = MANDY
       (hence ANOTHER = PETER, YOUNGERPERSON = MANDY)
```

```
succeed with OLDPERSON = PAUL, YOUNGERPERSON = MANDY
Hence OLDONE = PAUL, YOUNGONE = MANDY
```

For the data we have here, there is only one possibility that allows *both* parentof goals to work. The same scenario would be depicted by the tracer as follows, where we have just hit RETURN in response to each pause.

```
?- spy(grandparent).
{The debugger will first leap -- showing spypoints (debug)}
{Spypoint placed on user:grandparent/2}
ves
{debug}
?- grandparent(Oldone, Youngone).
     1 Call: grandparent(_193,_221) ?
 + 1
   2
      2
        Call: parentof(_193,_485) ?
   3
     3
        Call: daughterof(_193,_485) ?
   3
      3
        Exit: daughterof(peter,mandy) ?
   2
      2
        Exit: parentof(peter,mandy) ?
         Call: parentof(mandy,_221) ?
   4
      2
         Call: daughterof(mandy, 221) ?
   5
      3
   5
      3
         Fail: daughterof(mandy,_221) ?
   5
      3
         Call: sonof(mandy,_221) ?
   5
      3
         Fail: sonof(mandy,_221) ?
   4
      2
         Fail: parentof(mandy,_221) ?
   2
         Redo: parentof(peter,mandy) ?
      2
   3
      3
         Redo: daughterof(peter,mandy) ?
   3
      3
         Fail: daughterof(_193,_485) ?
   3
      3
         Call: sonof(_193,_485) ?
   3
      3
         Exit: sonof(paul,peter) ?
   2
      2
         Exit: parentof(paul,peter) ?
   4
      2
         Call: parentof(peter,_221) ?
   5
      3
         Call: daughterof(peter,_221) ?
   5
      3
         Exit: daughterof(peter,mandy) ?
   4
      2
         Exit: parentof(peter,mandy) ?
  1
        Exit: grandparent(paul,mandy) ?
 +
      1
Oldone = paul,
Youngone = mandy ?
```

yes

11 Summary so far

A Prolog procedure (predicate) consists of one or more clauses. A clause has a head and a body; if the body is empty, then this is a unit clause. A Prolog program is initiated by giving it a goal (query), which may contain unbound variables. A goal is carried out by finding a clause whose head matches the goal (possibly binding variables), and then carrying out the body, from left to right (taking account of any variable bindings so far). Each goal in turn is treated in this way. When a goal fails (either because there are no matching clauses left, or the built-in predicate it relies upon returns failure), the system re-tries the preceding goal to see if that can be achieved in another way (i.e. by finding a different clause). The overall result returned by a predicate is usually conveyed by the binding of a value to one of its originally unbound variables.

12 Unification and Equality

(See Clocksin & Mellish 2.4)

The process we have so far referred to as "matching" is more precisely known as *unification*, and we say that two Prolog items *unify* if they can be matched. The unification process is central to Prolog (it is the way in which variables acquire bindings) and we will encounter more complex examples of its workings as we go on.

We mentioned earlier that there is an equal sign, "=" in Prolog. However, this symbol really means "unify with" rather than "is equal to". This means it can act as a simple comparison:

|?- b = b. yes |?- a = b. no |?- 5 = 5. yes |? - 5 = 7. no These work because, for simple However, when variables are in though it is simply carrying out

These work because, for simple constants like a, b, 5,7, "can unify" and "are equal" give the same answer. However, when variables are involved, the "=" sign can seem to be acting as variable-assignment, even though it is simply carrying out unification:

```
|?- Variable1 = b.
Variable1 = b?
yes
|?- a = Variable2.
Variable2 = a?
yes
|?- Variable = b, a = Variable.
no
|?- 5 = X.
X = 5?
```

The failure in the third example here is because the first unification succeeds, binding Variable to b, and then the second unification fails because Variable cannot then unify with (and become bound to) a.

When we come to consider more complex structures, we will see that unification is defined roughly thus:

Two simple items unify if either they are identical, or at least one of them is an unbound variable, or they are variables both bound to items which unify. Two compound structures unify if they are the same kind of structure, contain the same number of components, and each of the corresponding components unify.

This definition is a simple example of *recursion*: a concept (unify) is defined for some simple "base" case, and then defined for non-simple examples in a way which decomposes the problem into simpler parts. We shall see more of recursion later on.

13 Arithmetic

(See Clocksin & Mellish 2.5, 2.3, 6.11)

Prolog is highly unusual amongst programming languages in that arithmetical expressions are *not* evaluated automatically when used. Suppose we type in:

| ?- 3 + 4.

You might expect the system to reply with 7 (followed by yes). However, what we get is:

```
{EXISTENCE ERROR: 3+4: procedure user:(+)/2 does not exist}
```

Prolog does not normally treat + as a symbol meaning addition of numbers. An expression like 3 + 4 is seen as a *structure*, in which the two components (3,4) are simply further data items. As a structure, it can be manipulated using unification:

| ?- X = 3+4. X = 3+4 ? | ? - V + W = 3 + 4. V = 3, W = 4 ?

yes

In that last example, we see an illustration of unification acting on a compound (non-simple) structure. The two structures are of the same type (involving the + operator), have the same number of components (2), and those components individually unify (∇ with 3, W with 4). More complicated examples allow for structures within structures:

| ?- (3 + 4) * (7 + 9) = (A + 4) * M.A = 3, M = 7+9 ? yes | ?- (3 + 4) * (7 + 9) = (C + D) * (E + F).C = 3, D = 4, E = 7, F = 9 ?

However, Prolog does have some arithmetical ability built into it. If you want an arithmetic expression to be evaluated, the is operator will achieve this (giving an effect similar to an assignment statement in a conventional language):

```
?- N is 3 +1.
N = 4 ?
yes
| ?- Total is (3 + 4) * (7 + 9).
Total = 112 ?
```

yes

This does *not* work the other way round – if you put the unbound variable on the right of the operator, it's an error:

| ?- 6 + 8 is M. {INSTANTIATION ERROR: _33 is _34 - arg 2}

The is operator is not entirely like a conventional assignment, as it binds the variable using unification. This means that if the variable is already bound, but the value unifies, that is acceptable:

?- M is 6, M is 3 * 2.

М = б

yes

That is, the is operator can be used to test if an existing value is equal to a calculated one.

If a variable has been bound to an expression, then is can be used to evaluate that expression:

| ?- Calculation = (2 * 3) + (5 * 4), Value is Calculation.

Value = 26, Calculation = 2*3+5*4 ?

yes

Also, a variable bound to an expression, as in the above example, can then appear in another expression, and is will evaluate the whole thing:

?- Calculation = (2 * 3) + (5 * 4), Value is Calculation + 1.

Value = 27, Calculation = 2*3+5*4 ?

yes

Because is uses unification, it will not work to write

?- N is 24, N is N + 1.

expecting N to become 25 (as would happen with assignment in a traditional language). The N is N + 1 expression fails, as the two sides of the is do not unify -N cannot simultaneously be 24 and 25.

14 Terminology and notation

(See Clocksin & Mellish 2.1)

The various kinds of expressions that we have encountered so far can be classified as shown in Figure 1.

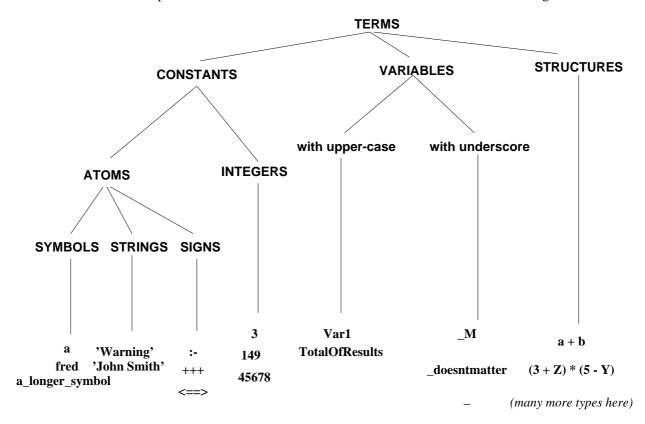


Figure 1: Types of item in Prolog

Constant atom symbols are generally used to represent specific objects, e.g. peter, grouchomarx, the_house_on_the_hill (underscores can be placed within a symbol to make it more readable). Constant atom strings are also used for this purpose, when we wish to include letters or punctuation which would not be allowed in a simple symbol, and are also used for general textual messages. Signs are usually used where we wish to introduce our own notation, with the sign naming some "operator" (see later). Integers represent numbers.

Variable names must start with either an upper-case letter or an underscore.

If the variable is simply the underscore, then it can match anything and never takes on a binding.

Structures are an important part of Prolog. Although the only examples so far are arithmetic expressions, there are others, both built-in and user-defined; see later.

For a predicate, the number of arguments that it has is known as its *arity*, and a predicate with N arguments is known as an "N-ary" predicate.

It is normal practice to refer to predicates not simply by their names (e.g. larger, parentof) but by their names paired with their arities: larger/3, parentof/2, etc. One reason for this is that a program may contain different predicates with the same name but different arities – larger/2 and larger/3 for example. Although it is not, in general, a good programming practice to have lots of similar-looking predicates – it can lead to confusion – this freedom can be handy where there is a predicate which should do much the same computation whatever number of arguments it gets. In particular, one of the arguments might be optional.

15 Lists

A widely-used type of data structure in Prolog is the *list*. A list is a collection, in order, of zero or more data items. The way of writing out a list in full is with the data items separated by commas and with square brackets enclosed them all; here is a list of four symbols:

[a, b, c, d]

A list can be any length, and can contain any kind of data item, even a mixture of types of data item; here is a list of 5 assorted items:

[a, 'A piece of text', 475, (a + 3) * (d - 7), _Var2]

Lists can contain be bound to variables, and can contain variables, resulting in lists being unifiable with other lists:

```
?- [a, b, c, d] = [Var1, b, Var2, d].
Var1 = a,
Var2 = c ?
yes
?- [1, What, 2, X, 3] = [A, b, C, d, E].
A = 1,
C = 2,
E = 3,
X = d,
What = b ?
yes
| ?- [(a + X), (Y + b)] = [(W + c), (d + b)].
W = a,
X = C
Y = d?
yes
| ?- [[X, a]] = [X,a].
no
| ?- [[X, a]] = [[b, Y]].
X = b,
Y = a ?
yes
```

This is an illustration of what we mentioned earlier about the recursive nature of unification – two structures (such as lists) unify if they are the same type of structure, and have components which unify – and so when there are structures within structures the unification process continues down inside the inner structures. These inner structures can also be lists, as in the last two examples above.

A list can be empty:

| ?- List = []. List = [] ? yes | ?- [a, b] = [].

no

Lists may seem, so far, to be flat structures, in which the first element in the sequence is as accessible or as prominent as the last, or one in the middle. This is not the case. In a list, the first (front) element is the most accessible, and there is then a division between that element and the rest of the list.

This can be thought of as having the following definition of a list:

The empty list, [], is a list. Anything added to the front of an existing list makes a list.

Or alternatively:

The empty list, [], is a list. A structure of the form [X, ...] is a list if X is any item and [...] is a list, possibly empty.

That is, a list is a recursively defined structure.

There is a special notation in Prolog to allow unification of lists in this "front-and-then-the-rest" manner. A vertical bar "|" is used to separate the parts of a list (informally, you can think of the bar as meaning "joined on to the front of"). For example:

```
| ?- [a, b, c, d] = [X|Y].
X = a,
Y = [b,c,d] ?
yes
| ?- [a, b, c, d] = [X|[Z|W]].
W = [c,d],
X = a,
Z = b ?
yes
| ?- [a|[b|[c|[ ]]]] = Wholelist.
Wholelist = [a,b,c] ?
no
```

16 General structures

(See Clocksin & Mellish 2.1.3, 3.1)

16.1 Clustering items

Many programming languages have structures for clustering related pieces of information into handy chunks. In many languages, these are called *records*. The nearest equivalent in Prolog is the type of term known as a *compound* or *complex* structure. This consists of a *functor* (in the form of a constant) and one or more arguments, separated by commas and surrounded by round brackets. For example:

```
person('Dave the Dude', '58 Broadway', 36)
```

Here we have a functor *person* and three arguments.

Structures can be the value of variables, can have variables as their arguments (but *not* as their functors), and can unify with other similar structures:

```
| ?- Theman = person('Dave the Dude', '58 Broadway', 36).
Theman = person('Dave the Dude', '58 Broadway', 36) ?
yes
| ?- person(Name, Address, Age) = person('Dave the Dude', '58 Broadway', 36).
Age = 36,
Name = 'Dave the Dude',
Address = '58 Broadway' ?
yes
| ?- person(Someone, _ , 45) = person('Harry the Horse', '42nd and Vine', 45).
Someone = 'Harry the Horse' ?
```

yes

(The plain underscore does not receive any bindings.)

A structure may have any item as an argument, including another structure. This can be handy as it allows the use of a functor to make it clear what sort of thing the argument is supposed to be, and to ensure that it does not accidentally unify with some different type of value. Compare the clarity of these:

```
address(23, 140, 2122)
```

address(flat(23), streetnumber(140), postcode(2122))

Unification then works in the obvious way:

```
?- address(flat(23), streetnumber(140), postcode(2122)) =
    address(flat(F), _Street, Code).
F = 23,
Code = postcode(2122) ?
yes
?- address(flat(23), streetnumber(140), postcode(2122)) =
    address(postcode(P), _Street, flat(F)).
```

no

(Any variable beginning with underscore is not reported by the interface).

Structures, in contrast to records in some languages, do not have to be declared in any way. If a functorargument expression appears in a Prolog program in a suitable position (i.e. as the argument to something else), then it will be treated as a structure, even if the functor has not been defined anywhere as being intended as the label for a structure.

16.2 Arithmetic expressions as structures

The arithmetic expressions discussed earlier were actually structures, although their functors were *infix* functors: they were positioned *between* their two arguments rather than in front of them. The Prolog system has built-in definitions for the arithmetic symbols indicating that they are to be handled as infix operators. However, they can also be written as conventional structures:

| ? - + (2, 3) = 2 + 3.

yes

| ?- +(2, *(5,6)) = +(X, *(5, Y)).

X = 2, Y = 6 ?

yes

```
| ?- Result is +(2, *(5,6)).
Result = 32 ?
```

yes

Although it is possible to *create* structures with a functor which is an arithmetic operator and with several arguments, they do not make sense to is as evaluatable expressions:

| ? - + (2, 3, 4, 5) = X.

X = +(2, 3, 4, 5)?

yes

| ?- M is +(3,4,5,6).
{DOMAIN ERROR: _38 is+(3,4,5,6) - arg 2: expected expression, found +(3,4,5,6)}

16.3 Lists as structures

Lists, as described earlier, are in fact a kind of functor-argument structure, for which Prolog systems have special notations (as shown earlier) for reading and writing purposes. Lists are held together by a functor which prints as a simple dot or full-stop. This functor connects the first item in a list to the rest of the list.

?- .(a, []) = [a].

yes

| ?- List = .(a, []) . List = [a] ? yes
| ?- .(a, .(b, .(c, []))) = Alist.
Alist = [a,b,c] ?
yes
| ?- .(a, .(b, .(c, []))) = [First, b, Last].
Last = c,
First = a ?
yes
| ?- .(a, .(b, .(c, []))) = [a| .(b, [c])].

yes

In the last of these examples, we have, on the right of the unification sign, mixed the three notations for lists: vertical bar, dot-functor, and square brackets. The vertical bar separates a from some further list; that list is formed by the dot-functor linking its two arguments, b and the list [c].

As long as we use them consistently, it will work, but mixing notation like this is not recommended.

17 Circularity and loops

In any programming language, it is possible to create programs which repeat indefinitely – endless loops. Usually, this is an unwanted effect. If you suspect that your Prolog program is doing this, hit CONTROL-C to interrupt the program. You will get the prompt:

Prolog interruption (h for help)?

Typing h (followed by RETURN) will show what options are available, typing a (followed by RETURN) will abort the run of the program and take you back to the normal command-line user interface for Prolog.

It is possible in Prolog to create data structures which are themselves circular. These have to be handled with care, and are *not* recommended at this stage, but may occasionally be appropriate for certain tasks. One problem with these doubly-linked items is that printing them causes an endless loop. Since the Prolog user interface prints out the results of your top-level command, if that command instantiates a variable to a circular (looping) structure, the interface will try to print it, cycling endlessly round the loop.

For example, the following goals (starting with uninstantiated variables) create circular structures:

? - X = X + 1.

? - Y = 1 + Y.

?-B = g(B).

Unification in Prolog is based on a well-defined mathematical operation from formal logic, but in logic such circular structures are not permitted: proper logical unification has a check (the *occurs* check) to see that a variable is never unified with anything which contains it. Prolog, in the interests of efficiency and being a general programming language, has omitted this check, and so permits these structures.

These few notes on circular structures have been included as a warning in case you accidentally create some – they are not recommended for everyday use.

References

[Bratko 90]	I. Bratko. <i>Prolog Programming for Artificial Intelligence (2nd edition)</i> . Addison Wesley, 1990. ISBN 0-201-41606-9.
[Clocksin & Mellish 94]	W.F. Clocksin and C.S. Mellish. <i>Programming in Prolog (4th edition)</i> . Springer-Verlag, 1994. ISBN 0-387-58350-5.
[O'Keefe 90]	R. O'Keefe. The Craft of Prolog. MIT Press, 1990. ISBN 0-262-15039-5.
[Ross 89]	P. Ross. <i>Advanced Prolog, Techniques and Examples</i> . Addison Wesley, 1989. ISBN 0-201-17527-4.
[Shoham 94]	Y. Shoham. Artificial Intelligence Techniques in Prolog. Morgan Kaufmann, San Francisco, 1994. ISBN 1-55860-319-0.
[Sterling & Shapiro 94]	L. Sterling and E. Shapiro. <i>The Art of Prolog (Second edition)</i> . MIT Press, 1994. ISBN 0-262-19338-8.