L333: Introduction to Prolog

Steve Harlow
sjh1@york.ac.uk

Contents

List of Exercises

1 Database Prolog

1.1 Introduction

Programming languages can be classified in a variety of ways. One major division is between *procedural* languages and *declarative* languages. Procedural languages require the programmer to specify in some detail the steps the program must execute in order to accomplish its intended task. In declarative languages, on the other hand, the programmer provides in the program a definition of the task to be accomplished, but is not concerned with the details of how the computer will use the definition.

Most conventional programming languages (Basic, C, Fortran, Pascal etc.) are procedural. Prolog belongs to the class of declarative programming languages. It gets its name (PROgramming in LOGic) because it is modelled on first order logic and (in principle) requires the programmer to give a logical model of the task to be undertaken. This makes programming in Prolog a rather different experience to programming in a procedural language.

In introducing Prolog, I have followed Pereira and Shieber (1987) in breaking it down into three stages. The first, *Database* Prolog, is a restricted subset which allows us to exemplify some of the basic ideas. The second, Pure Prolog, is an extension of the first which maintains a purely logical semantics but introduces arithmetic, recursion and lists. The final extension, to Full Prolog, takes us into the range of extra-logical constructs, such as input and output, negation and flow of control.

1.2 Facts and rules

A Prolog program consists of a file containing a set of facts and (optionally) a set of rules.

1.2.1 Facts

Semantically the facts constitute a declaration of a true state of affairs. As far as Prolog is concerned, any fact in its database is treated as true.

If a file containing the fact

```
male(phil).
```

is consulted, the goal

```
?-male(phil)
```

elicits the Prolog response

```
yes
```

This is Prolog reporting that the expression evaluates as true with respect to its database.

With respect to the same database, the goals

```
?-female(phil).

?-male(chas).
```

will produce the responses:

```
?-female(phil).

no

?-male(chas).

no
```

That is, with respect to this database, these facts are not known to be true.

A database consisting only of facts is not very interesting. The action picks up when *rules* are added.

1.2.2 Rules

Logic contains rules of inference. A very basic one is the one known as *Modus Ponens.* This takes the following form

$p \supset q$	from the formula ‘p implies q’
p	if proposition ‘p’ is true
$\therefore q$	conclude that ‘q’ is true

For example:

If John is drunk, then John is happy
John is drunk
∴ John is happy

Prolog includes conditional formulae (like $p \supset q$, but with some syntactic reorganisation), and an algorithm for computing inferences using Modus Ponens, but before we look at how rules are represented in Prolog, we need to look in more detail at the relationship between standard logic and Prolog.

1.2.3 Horn Clause Logic

The kind of logic used by Prolog is a subset of First Order Logic called *quantifier-free Horn Clause Logic.* As the name suggests, there are no existential or universal quantifiers. Instead, any variable is implicitly assumed to be universally quantified. The statement

```
male(X).
```

is therefore to be taken as $\forall x. male(x)$, or 'Everything is male'.

Note that any expression which begins with a capital letter or with the underscore character is a *variable* in Prolog.

The implication sign $\rightarrow$ or $\supset$ is represented in Prolog as `:-`, and the order of antecedent and consequent is reversed, so $p \supset q$ in Prolog will become `q :- p`.

Horn Clause logic is characterised by the fact that it allows only one term[1] in the consequent of a conditional (i.e. before the `:-`). What follows the conditional must be a term (possibly complex and consisting itself of subterms including conjunctions and disjunctions).

Conjunctions, i.e. expressions which take the logical form `p` $\wedge$ `q` are realised in Prolog as `p , q`, with a comma replacing the standard conjunction sign. So, a term `p , q` is true iff both `p` and `q` are true.

Disjunctions, i.e. expressions which take the logical form `p` $\vee$ `q` are realised in Prolog as `p ; q`, with a semicolon replacing the standard disjunction sign. So, a term `p ; q` is true if `p` is true or `q` is true (or both are).

The formula $\forall x.\forall y. female(x) \wedge parent(x, y) \supset mother(x, y)$ ('If x is female and y's parent, then x is y's mother'.) translates into Prolog as

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

The part of the rule preceding the implication sign (i.e. `mother(X, Y)`) is termed the **head** of the rule and the part to the right of the implication sign is termed the **body** of the rule.

[1]A term is

1. a constant
2. a variable
3. a structure, consisting of a functor (which is a constant) followed by a sequence of terms enclosed in brackets and separated from one another by commas.

David Warren, writing in *The Art of Prolog* (xi) writes,

> The main idea was that deduction could be viewed as a form of computation, and that a declarative statement of the form:
>
> $Q \wedge R \wedge S \supset P$
>
> could also be interpreted procedurally, as:
>
> "To solve P, solve Q and R and S."

This is what Prolog does.

Given a rule such as the following:

```
mother(M, C):-
 female(M),
 parent(M, C).
```

it attempts to prove a goal such as:

```
?-mother(liz, chas).
```

by first proving `female(liz)` and then (if this is successful) by proving `parent(liz, chas)`. If this process fails at any point, Prolog will report its failure with `no`.

Note that the deduction algorithm proceeds in a left-to-right, depth-first order.

A set of facts and rules constitutes a (logic) program.

1.3 Goals and queries

A query is a means of extracting information from a logic program and consists of attempting to prove that the query is a logical consequence of the program. We will say more about how Prolog does this in section 1.4. When we pose a query to Prolog, we are setting up a goal for Prolog to try to satisfy. If Prolog is able to find facts and rules that allow it to conclude that the goal is true, we say that the goal is 'satisfied', or 'succeeds'; if a goal cannot be satisfied, we say it 'fails'.

There is a major difference between the interpretations assigned to variables in facts and queries. In facts, variables are taken as being universally quantified; in queries, they are treated as being existentially quantified.

As a fact in file, `human(X)` is equivalent to $\forall x.human(x)$ – ‘everything is human’.

As a query,

```
? human(X).
```

corresponds to evaluating $\exists x.human(x)$ – ‘is there at least one thing that is human?’.

1.4 Unification

The deduction process involves matching terms in the goal against terms in head of the rule, and terms in the body of the rule against other terms, either in the head of other rules or against facts. The matching process used is termed *unification*.

Briefly, it works as follows:

- to unify, two terms must be of the same *arity* - that is, they must take the same number of arguments.
- if this condition is satisfied, then unification reduces ultimately to a matching of the basic terms in a compound expression. Basic terms unify as follows:
    1. identical constants unify (e.g. `chas` and `chas`).
    2. a constant and a variable unify (e.g. `chas` and `X`) - and as a side-effect, the variable becomes instantiated to the value of the constant. (So from the point of unification on, the variable ceases to be a variable.)
    3. two variables unify. (`X` and `Y` will become the same variable).

Assume the following Prolog program, with four facts and one rule.

```
male(phil).

female(liz).
```

```
parent(phil, chas).
parent(liz, chas).

mother(M,C):-
 female(M),
 parent(M,C).
```

The goal:

```
?-mother(liz,chas).
```

is evaluated like this:

```
?- mother(liz,chas).
 1 1 Call: mother(liz,chas) ?

 2 2 Call: female(liz) ?
 2 2 Exit: female(liz) ?

 3 2 Call: parent(liz,chas) ?
 3 2 Exit: parent(liz,chas) ?
 1 1 Exit: mother(liz,chas) ?

yes
```

In this case the terms in the consequent of the rule matched immediately against facts in the database. Let's add the following fact and rules to those given above

```
male(harry).

parent(chas,harry).

grandmother(GM, C):-
 mother(GM, P),
 parent(P, C).
```

The goal `grandmother(liz,harry)` evaluates as follows:

```
?- grandmother(liz,harry).
 1 1 Call: grandmother(liz,harry) ?

 2 2 Call: mother(liz,_732) ?

 3 3 Call: female(liz) ?
 3 3 Exit: female(liz) ?

 4 3 Call: parent(liz,_732) ?
 4 3 Exit: parent(liz,chas) ?
 2 2 Exit: mother(liz,chas) ?

 5 2 Call: parent(chas,harry) ?
 5 2 Exit: parent(chas,harry) ?
 1 1 Exit: grandmother(liz,harry) ?

yes
```

As before, Prolog first tries to prove the leftmost term in the antecedent, but this requires that it search further, since `mother(liz,_732)` is not a fact. Only when it has successfully concluded this subproof (via `female/1` and `parent/2`) does it continue to attempt the second clause in the definition of `grandmother/2`.[2]

Note that this version of Prolog (like most), renames the variables you supply it with. In this case `C` has become _732. This is to avoid accidental clashes of variable-names. Note also that when `GM` unifies with the constant `liz`, Prolog only uses the constant in its subsequent search.

1.5 Exercises

Exercise 1 *Extend the program by adding rules for the following family relationships (add more people if necessary, so that you can check your results):*

[2]Note the standard conventional way of citing a Prolog predicate in the form `predicate_name/arity`.

`brother(X, Y)`	*where X is Y's brother*
`sister(X, Y)`	*where X is Y's sister*
`son(X, Y)`	*where X is Y's son*
`daughter(X, Y)`	*where X is Y's daughter*
`married(X, Y)`	*where X is married to Y*
`ancestor(X, Y)`	*where X is Y's ancestor*

What problems do you encounter? What is the nature of the problems? What solutions (if any) can you suggest. (Reading the literature on Prolog will help.)

2 Pure Prolog

2.1 Recursion and lists

Recursion is an extremely powerful tool and one which is widely used in Prolog programming.[3]

Although recursion can be used over many different data structures, one of the most frequently encountered in NLP environments is the list. So it is to this that we turn our attention first.

2.1.1 Lists

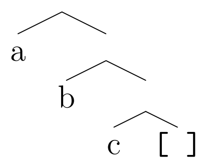
Lists in Prolog are themselves terms, and consist of a sequence of terms separated from one-another by commas and enclosed at each end by matching square brackets:

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

Note that, as you might expect, `[ ]` represents the empty list.

Internally to Prolog, lists are represented as right-branching trees. So the structure of `[a, b, c]` would be

[3]For those of you who know about these things: Prolog does not contain 'while' loops, so recursion replaces loops.



The first element of a list is called the **head** of the list, and the remainder is called the **tail** of the list. Note that (perhaps counterintuitively) the last element of every non-empty list is the empty list; although the normal Prolog notation suppresses this fact. So `a` is the head of all the above lists; `[ ]` is the tail of the first example, `[b]` is the tail of the second example, and `[b, c]` is the tail of the third example.

Because of the importance of the distinction between the head and tail of a list, Prolog provides a convenient notation that can be used to match against the head and tail of any list.

```
[X | Y]
```

Where `X` will match against the head, and `Y` against the tail. (Of course, any variable names may be used, not just `X` and `Y`.)

To illustrate these concepts, we will also introduce an operator which can be used to evaluate unification (discussed in section 1.4) directly. This operator is the equals sign `=/2`. The expression `A = B` is true in Prolog, if `A` is a term, and `B` is term, and `A` and `B` unify. The following can be typed in and evaluated:
Two constants:

```
?- a = a.

yes

?- a = b.

no
```

A constant and a variable:

```
?- a = X.

X = a ?

yes
```

Two variables:

```
?- X = Y.

Y = X ?

yes
```

Using this operator, we can illustrate how variables unify with lists:
Lists and variables are both terms:

```
?- [ ] = X.

X = [ ] ?
```

Single-element lists unify (because they have the same 'arity'):

```
?- [a] = [X].

X = a ?

yes
```

The following fails, because the two lists have different numbers of elements:

```
?- [a, b] = [X].

no
```

The following unify because both lists are of length 2:

```
?- [a, b] = [X, Y].

X = a,

Y = b ?

yes
```

Note carefully the difference between the above and the following using `Head|Tail` notation.

`X` matches the first element of the list `[a]`, which is the constant `a`, and `Y` matches the tail, which is the empty list:

```
?- [a] = [X | Y].

X = a,

Y = [ ] ?

yes
```

Here again, `X` matches with `a`, and `Y` matches with the tail of the list `[a, b]`, which is itself the list `[b]`:

```
?- [a, b] = [X | Y].

X = a,

Y = [b] ?

yes
```

Finally, a three element list:

```
?- [a, b, c] = [X | Y].

X = a,

Y = [b, c] ?

yes
```

A couple of trivial examples of predicates using lists are the following:

2.1.2 `first/2`

A predicate, which when supplied with a list a first argument, returns the first element of the list as its second argument:[4]

```
first([First|Rest], First).
```

[4]The `first/2` predicate could just as well have been defined as follows, with explicit calls to the unification operator `=/2`:

```
first(List, First):-
 List = [Head|Tail],
 First = Head.
```

2.1.3 `second/2`

One which returns the second element of a list:

```
second([First, Second|Rest], Second).
```

2.1.4 `tail/2`

One that returns the tail of a list:

```
tail([First|Tail], Tail).
```

2.1.5 Remarks on Variables

Naming variables The only syntactic requirement that Prolog places on variables is that they start either with an upper case letter, or with an underscore. This means that you could restrict yourself to writing `X`, `Y` or `Z` when you want a variable. For example, the program `first/2` in footnote 4 might look like this:

```
first(X, Y):-
 X = [Z|W],
 Y = Z.
```

This will do exactly the same job as the original definition, but it is not a good idea - as programs become more complex, they become very difficult to understand if you use such simple names for variables. Even this simple example illustrates how opaque such definitions can be come.

By giving your variables mnemonic names, you help to document the function of your programs and to explain to a reader (including yourself) what information the variables are intended to convey.

Use mnemonic variables — make your programs self-documenting.

The version in the text has moved these calls from the body of the definition into the head of the definition (as a result of which the body is empty). This technique is called 'unfolding' and is a commonly used technique in Prolog programming.

The anonymous variable If you have tried putting any of the above examples into a file which you then consult into Prolog, you will have discovered that it complains. For example, loading the rule for `second/2` produces the following:

```
    Warning: (<filename>:1):

        Singleton variables: [First,Rest]
```

This is because each of the variables `First` and `Rest` occurs only once in the definition. Because the most common use of variables in Prolog is to convey information from one part of a program to another, the occurrence of any singleton variable triggers a warning, to alert the programmer to the possibility of an error. What to do?

1. Ignore the warning. It doesn't affect the running of the program in any way; but there may be a genuine mistake hidden in the list of singleton variables which becomes hard to spot

2. Use the underscore symbol instead of your singleton variables. This would change the definition of `second/2` to this

   ```
   second([_, Second|_], Second).
   ```

   Although we have two occurrences of the underscore in this definition, Prolog treats them as unrelated (and identical in interpretation to the original definition), as 'anonymous'.

2.1.6 Exercises:

Exercise 2 *Rewrite the definitions of* `second/2` *and* `tail/2` *above to include explicit calls to the unification operator* `=/2`*.*

Exercise 3 *Define a predicate* `fifth/2` *that, given a list as first argument, returns the fifth element of the list as its second argument. E.g.*

```
?- fifth([1,2,3,4,5,6,7], X).

X = 5

yes
```

Exercise 4 *Recall that every list (the empty list) has both a head and a tail. Use this fact, and the* ***head|tail*** *notation to define a predicate* ***is_list/1*** *that returns true if its argument is a list (including the empty list) and false otherwise.*

Exercise 5 *Define a predicate* ***cons/3****, which takes a list as its second argument, anything as its first argument and returns as its third argument a new list in with the first argument as head and the second argument as tail.*

2.1.7 Recursion

The `ancestor/2` problem of exercise 1 discussed above provides a classical example of recursion. Such recursive rules always have two components,

1. the **base** of the recursion

   ```
   ancestor(P, C) :-
       parent(P, C).
   ```

   and

2. the **recursion** itself

   ```
   ancestor(A, D):-
       parent(A, C),
       ancestor(C, D).
   ```

The base of the recursion is a non-recursive clause, which, in a computational context, tells the computation when to stop. Here the minimal definition of ancestor is a parent. The recursion crucially involves the predicate calling itself.

2.1.8 `member/2`

A classical example of recursion in list-processing is identifying whether some item is an element of a list. This predicate is commonly called `member/2`. It is true if the item is on the list, and false otherwise. The base is that situation where we can be sure that we have found the item in question on the list. The limiting case here would be if the item sought is the first one on the list. We can generalise from this to the following clause:

```
 member(Item, List):-
   List = [Item|Rest].
```

This is OK if the item sought is indeed the first item, but what if it isn't? In that case, we need to ignore the first item and proceed to check the remainder (i.e. the tail) of the list (note the use of the anonymous variable for the item we aren't interested in):

```
member(Item, [_|Tail]):-
   member(Item, Tail).
```

If the item sought is on the list, this procedure will find it, and Prolog will respond with 'yes'; if not, it will fail, and Prolog will respond with 'no'.[5] Note that, as usual with many Prolog predicates, member/2 can be used to generate solutions; if the second argument is instantiated to a list and the first is a variable, the predicate can to used to select items off the list:

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

X = a ? ;

X = b ? ;

X = c ? ;

no
```

2.1.9 append/3

Another classical example of recursion is the predicate append/3. This predicate's three arguments are all lists. append(X, Y, Z) is true if the Z is a list that is the result of sticking the list Y onto the end of the list X. For example, append([a,b], [c,d], [a,b,c,d]) is true.

The definition of append/3 is similar to that of member/2 in that both involve working down a list item by item. In the case of append/3, however,

[5]This definition of member/2 can be simplified slightly by 'unfolding' the call to =/2 and placing it in the head of the rule, to give:

```
 member(Item, [Item|List]).
  member(Item, [_|Tail]):-
  member(Item, Tail).
```

no test is being carried out, and eventually the end of one of the lists will be reached, leaving the empty list. This gives the base: what is the result of appending an empty list to a list `L`? Clearly just `L`. In Prolog:

```
append([ ], List, List).
```

What if the first list isn't empty? We need to take its head, save it, and stick it onto the front of the result. The result itself comes about as the consequence of carrying out the same procedures on the tail of the list. This is the recursive clause:

```
append([Head|Tail], List, [Head|Result]):-
     append(Tail, List, Result).
```

`append/3` can also be used to generate solutions. Here are some examples:

1. last argument a variable

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

   X = [a,b,c,d] ?

   yes
   ```

2. first argument a variable

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

   X = [a,b] ?

   yes
   ```

3. second argument a variable

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

   X = [c,d] ?

   yes
   ```

4. first two arguments variables (and multiple solutions requested)

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

X = [ ],

Y = [a,b,c,d] ? ;

X = [a],

Y = [b,c,d] ? ;

X = [a,b],

Y = [c,d] ? ;

X = [a,b,c],

Y = [d] ? ;

X = [a,b,c,d],

Y = [ ] ? ;
```

2.1.10 Exercises

Hint: in all of the following, the base involves the case where the first list argument is the empty list.

Exercise 6 *Define a predicate `delete/3` whose arguments are 1) a term, 2) a list and 3) another list consisting of the second argument minus the first occurrence of the term, if it occurs. For example, `delete(b, [a,b,c,a,b], X)` should give `X = [a,c,a,b]`. Note also that `delete/3` should not fail; if the item to be deleted is not on the list, the original list should be returned as the value of the third argument. E.g. `delete(a, [b,c,d], X)` should give `X = [b,c,d]`*

Exercise 7 *Define a predicate `delete_all/3`, like `delete/3` except that the third argument is minus all occurrences of the first argument. E.g.*

`delete_all(b, [a,b,c,a,b], X)` should give `X = [a,c,a]`. `delete_all/3` should behave like `delete/3` if its first argument is not on the list.

Exercise 8 *Define a predicate `reverse/2` whose arguments are both lists, the second being the mirror image of the first. E.g. `reverse([a,b,c], X)` should give `X=[c,b,a]`. Hint: you will find it useful to use `append/3` in the recursive clause of `reverse/2`.*

Exercise 9 *Write a recursive definition that will translate a string of English words into their French (or German or Swedish or . . . counterparts). You will need a set of 2-place facts giving the English and French counterparts, and a two-place recursive rule that works its way down a list, looking up the word-translations item by item, and putting the resulting translation into the second argument. The predicate should produce the following kind of result:*

```
?- translate(['John', is, an, idiot], French).

French = [Jean,est,un,imbecile] }

yes
```

2.2 Operators

Before looking at other examples of recursion, it will be helpful to look at **operators**, because these are used in Prolog arithmetic. The basic arithmetic operations of addition, subtraction, multiplication etc., could be treated as predicates within the kind of Prolog style you have encountered so far. Addition, for example, might look like this:

```
+(X, Y, Z)
```

where `X` and `Y` are the arguments to be added, and `Z` is the result of the addition. Indeed, some Prologs allow you to do addition like this. Many more Prologs, however, define arithmetical operations in a way which is much more like the standard conventions, and write the above equation as

```
Z is X + Y
```

This is a novel format, because it writes the predicate symbols `is` and `+` between their arguments, without brackets. This is because these two symbols have been defined within Prolog as infix operators.

In working with operators, we need to pay attention to two factors: **precedence** and **associativity**.

2.2.1 Precedence

In an expression with more than one operator, we need to know which operator to evaluate first. For example, the arithmetical expression $1 + 2 * 3$ is potentially ambiguous. It could be $(1 + 2) * 3$ (i.e. 9), or $1 + (2 * 3)$ (i.e. 7). In fact, the convention in mathematics is to the latter: multiplication has precedence over addition. The `*` operator in Prolog is defined so that it too has precedence over the `+` operator.

2.2.2 Associativity

Precedence involves the case where there are different operators in a single expression. Associativity involves multiple occurrences of the same operator. For example, $3 - 2 - 1$ gives different results depending on whether it is treated as $(3 - 2) - 1$ (i.e. 0), or $3 - (2 - 1)$ (i.e. 2). In arithmetic it is the first order which is conventional: the left-most pair of numbers is evaluated first, then the next left-most, and so on. The subtraction operator is said to be *left-associative*.

Precedence and associativity are defined in Prolog with the built-in predicate `op/3`. The first argument determines precedence, the second associativity and the third defines the operator symbol. i.e.

```
op(Precedence,Associativity,Operator)
```

The associativity argument is also used to declare whether the operator is an infix operator (like the arithmetical ones), a prefix operator or a postfix operator.

The precedence argument is an integer. The principle is: the lower the number, the higher the precedence. In SWI-Prolog, the arithmetical operators have the following precedence values:

Operator	Precedence Value
+	500
-	500
*	400
/	400

The conventions for defining the order (prefix, infix and postfix) and associativity (left, right, none) of operators are as follows, where `f` stands for the position of the operator and `x` and `y` for its arguments:

Prefix	Postfix	Infix	Associativity
fx	xf	xfx	none
	yf	yfx	left
fy		xfy	right

The occurrence of the argument symbol `y` indicates that an expression may contain multiple occurrences of the operator in question, and its position (left or right of `f`) indicates whether it is left or right associative.

The set of built-in operators and their properties can be explored by use of the built-in predicate `current_op/3`:

```
?- current_op(Precedence,Associativity,Operator).

Operator = |,

Precedence = 1100,

Associativity = fxy ?

yes
```

Forcing Prolog to backtrack, producing further responses, by using semicolon `<return>` will produce a complete listing of all the operators know to the system at the time the goal is posed.

2.3 Arithmetic in Prolog

Prolog is not the programming language of choice for carrying out heavy-duty mathematics. It does, however, provide arithmetical capabilities. The pattern for evaluating arithmetic expressions is (where `Expression` is some arithmetical expression)

```
X is Expression
```
[6]

The variable `X` will be instantiated to the value of `Expression`. For example,

```
?- X is 10 + 5.

X = 15 ?

yes

?- X is 10 - 5.

X = 5

yes

?- X is 10 * 5.

X = 50

yes

?- X is 10 / 5.

X = 2

yes

?- X is 10 + 5 * 6 / 3.

X = 20

yes
```

[6]Remember that the equals sign is used in Prolog for unification, not arithmetic.

It is important to note that `Expression` is not evaluated by itself. You have to supply a variable (followed by the infix operator `is/2`) to collect a result.

Other pre-defined Prolog arithmetic infix operators are

`>`	greater than
`<`	less than
`>=`	greater than or equal to
`=<`	less than or equal to

Later in the course we will use `op/3` to define operators of our own, to make programs easier to read. Here we will return to the topic of defining recursive rules, with the addition of arithmetic.

The built-in predicate `display/1` can be used to reveal the 'real' constitution of expressions containing operators. For example,

```
?-  display(X is 10 + 5 * 6 / 3).
is(_45,+(10,/(*(5,6),3)))
true ?
```

shows that `is/2` is the principal functor, then `+`, then `/` and so on. This is very handy if you are unsure about the effects of your operator declarations.

2.3.1 `length/2`

It is often useful to be able to calculate (or check) the length of a list. With arithmetic operators this is something that we can now do. The base is the empty list, which is obviously of length 0. This give us the Prolog clause:

```
length([ ], 0).
```

The recursion simply requires that we add 1 for each item on the list:[7]

```
length([H|T], N):-
   length(T, N1),
   N is N1 + 1.
```

[7]`length/2` comes predefined in many Prologs. If you put this definition in a file and consult it, you will receive an error message. The solution is to rename the predicate in your definition, e.g. to `mylength/2`.

2.3.2 Exercises

Exercise 10 *Define predicate `square/2` that takes a number as its first argument and returns the square of that number as its second argument. (No recursion in this one.)*

Exercise 11 *Define a predicate `power/3` that takes numbers as its first two arguments `P` and `N` and returns as the value of its third argument a number which is N to the power of `P`. E.g.*

```
?- power(3,2,P).

P = 8

yes
```

Note that this requires a recursive rule and the use of arithmetic.

3 Full Prolog

Earlier, I partitioned Prolog into 3 subsets:

1. **Database Prolog**
2. **Pure Prolog**
3. **Full Prolog**

Database Prolog was what we started with; Pure Prolog introduced lists, maths and operators

Full Prolog goes beyond purely logical constructs and introduces various non-logical predicates for handling such operations as input and output, and controlling the execution of programs.

3.1 Input and output

The most useful input output predicates are `write/1`, `read/1` and `nl/0`.

`write(term)` is true if `term` is a Prolog term. As a side-effect, it causes `term` to appear on the current output device (e.g. your screen in the Winterm window).

`nl` is always true, and as a side-effect, sends a newline instruction to the current output device. So the conjunction of a `write/1` instruction and a `nl/0` instruction will result in a term being displayed on the screen, followed by a carriage return.

`read(X)` is true if the user types a term followed by a full-stop. `X` becomes instantiated to the term. E.g.

```
?- read(X).
|: this.

X = this ?

yes
?-
```

`read/1` can be used to input a list of words (because a list is a term), but the list has to be typed complete with brackets and commas:

```
?-read(X).

[this,is,a,list].

X = [this,is,a,list]

yes
```

Getting Prolog to accept a string of atoms and convert them into a list is quite heavy-duty Prolog programming and involves the use of `get/1` and `get/0`, which accept single characters (in ASCII numerical format); the building of a list of such numbers; their conversion from a list of numbers into a Prolog atom, using the predicate `name/2` and, finally the construction of a list of such atoms.

`display/1` was mentioned on page 23. This is useful for showing the meaning of operator declarations.

`listing/0` and `listing/1` are predicates which display the contents of the Prolog database on the screen.

`listing/0` will write all the predicates known to Prolog (the ones defined in the files you have consulted and also a number of built-in ones).

`listing/1` takes the name of a predicate as argument and writes its definition to the screen.

These are useful for checking that Prolog has actually ingested the information you have supplied it with. Sometimes, if there is a bug in your program, it may not load completely, or it may load in a form you don't expect. `listing/n` will enable you to check your anticipated version against what Prolog believes. (In the event of a disagreement, Prolog is right!)

Here is an example.

```
Welcome to SWI-Prolog (Version 4.0.9)
Copyright (c) 1990-2000 University of Amsterdam.
Copy policy: GPL-2 (see www.gnu.org)

For help, use ?- help(Topic). or ?- apropos(Word).

?- listing.

pcehome_(A) :-
        pcehomestore_(A), !.

pcehome_(A) :-
        (   getenv('XPCEHOME', B)
        ;   current_prolog_flag(home, C),
            (   current_prolog_flag(xpce_version, D),
                atom_concat('/xpce-', D, E)
            ;   E='/xpce'
            ),
            atom_concat(C, E, B)
        ),
        exists_directory(B), !,
        absolute_file_name(B, A),
        asserta(pcehomestore_(A)).
etc.
```

No files have been consulted, so Prolog only knows about various built-in system predicates. So now consult a file called `append.pl` and try again.

```
?- [append].
%append compiled, 17 msec 848 bytes

yes
```

```
?- listing.

myappend([], A, A).
myappend([A|B], C, [A|C]) :-
        myappend(B, C, C).


%   Foreign: window_title/2

mymember(A, [A|B]).
mymember(A, [B|C]) :-
        mymember(A, C).

yes
```

Now Prolog knows about `member/2` and `append/3`. Here is an example of the use of `listing/1` which here just shows the definition of `member/2`.

```
?- listing(mymember).

mymember(A, [A|B]).
mymember(A, [B|C]) :-
        mymember(A, C).

yes
```

Finally, and rather strangely, `listing/1` always succeeds, even if its argument is an unknown predicate – it just doesn't display anything!

```
?- listing(foo).

yes
```

3.2 Metalogical predicates

Other predicates in full Prolog allow you to test for the type of an expression:
`var(X)` is true if `X` is a variable when `var/1` is called.
`atom(X)` is true if `X` is an atom
etc. Any Prolog textbook will have a section discussing such predicates.

3.3 Cut

One very important, and very tricky, predicate is the 'cut', written as the exclamation mark `!`. This is quite dangerous in inexperienced hands (such as yours!). It is an instruction to Prolog not to back-track looking for other solutions. Here is are some examples:

3.3.1 Examples

Negation as failure In dealing with data-base Prolog, we discovered that the correct definition of `brother/2` and `sister/2` require a test for inequality:

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

We know how to check that two terms are the same (by unification, using `=/2`), but how do we check that they are different? In principle by checking that they do NOT unify. In fact, Prolog has a built-in predicate `\+/1` which allows this to be done:[8]

```
?- \+ a=b.
yes
```

So `different/2` could be programmed as:

```
different(X, Y):-
    \+ X=Y.
```

The interesting thing from the present perspective is how `\+/1` itself is programmed. The standard definition is the following two clauses:

```
\+ X :- X, !, fail.

\+ X.
```

[8]`\+/1` is defined as a prefix operator, so the normal brackets are not needed (unless what you are negating is a compound expression containing conjunctions or disjunctions.

These clauses MUST occur in this order, if the definition is to work. When `\+ term` is called, `X` is instantiated to `term`, and the first call in the body of the first clause of `\+/1` tests to see if `term` is true. If it is known to the Prolog data-base, this call will succeed and control will pass to `!/0` ('cut'). Cut always succeeds, so control passes to the built-in predicate `fail/0`. `fail/0` always fails. (Surprise, surprise.)

Normally, when a goal fails, Prolog backtracks to see if there are any alternative solutions. Here, to see if `term` has any more solutions. It will then repeat the procedure until all the possibilities have been exhausted, at which point control would pass to the second clause of `\+/1`, which (since it has no conditions) would succeed.

However, the presence of the cut in the first clause prevents this behaviour. Instead, when the call to cut has succeeded, Prolog 'cuts away' all the information about alternative choices before the cut. Therefore, when `fail/0` is encountered, there is nowhere else to go and the call \+ term fails straightaway.

The only situation in which the second clause will be reached is if `term` itself fails. This will be the case if `term` can't be found in the Prolog database, or proved indirectly.

So, `\+ a=b` will fail if `a=b`; and will succeed if `a=b` fails.

Read carefully a Prolog textbook before using the cut yourself. (Using `\+/1` is OK, provided you realise that when it fails its 'meaning' is 'Prolog was unable to find a solution'. I.e. 'don't know', rather than false. This definition of negation is called 'negation by failure'.

delete_all/3 You may have noticed that the predicate `delete_all/3` (which you defined in response to an exercise) does not work as anticipated if you force Prolog to give you more than one solution. Here is what happens:

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

X = [b] ? ;

X = [b,a] ? ;

X = [a,b] ? ;

X = [a,b,a] ? ;
```

```
no
```

Recall the definition of `delete_all/3`:

```
1. delete_all(_, [ ], [ ]).
2. delete_all(Item, [Item|Tail] , Result):-
       delete_all(Item, Tail, Result).

3. delete_all(Item, [Head|Tail] , [Head|Result]):-
      delete_all(Item, Tail, Result).
```

What happens is that, when the user requests a second solution, Prolog first of all goes back to the last procedure call made in providing the first solution (this was the second line of clause 2 of the definition, which succeeded by using clause 1 of the definition), and tries to see if there is an alternative way of satisfying the goal

```
delete_all(a,[ ],_595)
```

There isn't, so Prolog *backtracks*; having tried clause 2 unsuccessfully, it now tries clause 3. Clause 3 succeeds with `Item = a` and `Head = a`, as can be seen in the trace:

```
?- delete_all(a, [a,b,a], X).
 1 1 Call: delete_all(a,[a,b,a],_93) ?

 2 2 Call: delete_all(a,[b,a],_93) ?

 3 3 Call: delete_all(a,[a],_595) ?

 4 4 Call: delete_all(a,[],_595) ?
 4 4 Exit: delete_all(a,[],[]) ?
 3 3 Exit: delete_all(a,[a],[]) ?
 2 2 Exit: delete_all(a,[b,a],[b]) ?
 1 1 Exit: delete_all(a,[a,b,a],[b]) ?

X = [b] ? ;

 1 1 Redo: delete_all(a,[a,b,a],[b]) ?
 2 2 Redo: delete_all(a,[b,a],[b]) ?

 3 3 Redo: delete_all(a,[a],[]) ?

 4 4 Redo: delete_all(a,[],[]) ?
 4 4 Fail: delete_all(a,[],_595) ?  (in clause 2)

 4 4 Call: delete_all(a,[],_813) ?  (in clause 3)
 4 4 Exit: delete_all(a,[],[]) ?
 3 3 Exit: delete_all(a,[a],[a]) ?
 2 2 Exit: delete_all(a,[b,a],[b,a]) ?
 1 1 Exit: delete_all(a,[a,b,a],[b,a]) ?
```

The problem arises because we have assumed that the second clause excludes the third - but it doesn't. Now that we have access to the cut, we can impose this assumption on the program.

There are (at least) two ways to do this:

1. Indirectly by including an inequality test in clause 3:

```
3. delete_all(Item, [HeadTail] , [Head|Result]):-
       \+ Item = Head,
       delete_all(Item, Tail, Result).

?- delete_all(a, [a,b,a], X).
 1 1 Call: delete_all(a,[a,b,a],_93) ?

 2 2 Call: delete_all(a,[b,a],_93) ?

 3 3 Call: a=b ?
 3 3 Fail: a=b ?

 3 3 Call: delete_all(a,[a],_603) ?

 4 4 Call: delete_all(a,[],_603) ?
 4 4 Exit: delete_all(a,[],[]) ?
 3 3 Exit: delete_all(a,[a],[]) ?
 2 2 Exit: delete_all(a,[b,a],[b]) ?
 1 1 Exit: delete_all(a,[a,b,a],[b]) ?

X = [b] ? ;

 1 1 Redo: delete_all(a,[a,b,a],[b]) ?

 2 2 Redo: delete_all(a,[b,a],[b]) ?

 3 3 Redo: delete_all(a,[a],[]) ?

 4 4 Redo: delete_all(a,[],[]) ?

 4 4 Fail: delete_all(a,[],_603) ?

 4 4 Call: a=a ?
 4 4 Exit: a=a ?

 3 3 Fail: delete_all(a,[a],_603) ?

 2 2 Fail: delete_all(a,[b,a],_93) ?
 2 2 Call: a=a ?

 2 2 Exit: a=a ?

 1 1 Fail: delete_all(a,[a,b,a],_93) ?

no
```

2. Use the cut to prevent backtracking. We insert a cut after the head of clause 2

```
delete_all(_, [], []).

delete_all(Item, [Item|Tail] , Result):-
     !,
     delete_all(Item, Tail, Result).

delete_all(Item, [Head|Tail] , [Head|Result]):-
```

```
 delete_all(Item, Tail, Result).
```

Once execution of the program has passed the point at which the cut occurs, it cannot subsequently backtrack beyond this point. This means that once the head of clause 2 has unified with a goal, there is no longer any possibility of resorting to clause 3, as the following trace shows.

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

 1 1 Call: delete_all(a,[a,b,a],_93) ?

 2 2 Call: delete_all(a,[b,a],_93) ?

 3 3 Call: delete_all(a,[a],_598) ?

 4 4 Call: delete_all(a,[],_598) ?
 4 4 Exit: delete_all(a,[],[]) ?
 3 3 Exit: delete_all(a,[a],[]) ?
 2 2 Exit: delete_all(a,[b,a],[b]) ?
 1 1 Exit: delete_all(a,[a,b,a],[b]) ?

X = [b] ? ;

 1 1 Redo: delete_all(a,[a,b,a],[b]) ?
 2 2 Redo: delete_all(a,[b,a],[b]) ?
 3 3 Redo: delete_all(a,[a],[]) ?
 4 4 Redo: delete_all(a,[],[]) ?
 4 4 Fail: delete_all(a,[],_598) ?
 3 3 Fail: delete_all(a,[a],_598) ?
 2 2 Fail: delete_all(a,[b,a],_93) ?
 1 1 Fail: delete_all(a,[a,b,a],_93) ?

no
```

3.3.2 Exercises: memberchk/2

Exercise 12 *Another example. Recall the definition of* `member/2`*:*

```
member(Item, [Item|_]).
member(Item, [_|Tail]):-
   member(Item, Tail).
```

We previously used `member/2` to discover whether the first argument was an element of the second argument, but, if the first argument is a variable, it can also be used to generate a set of answers:

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

X = a ? ;

X = b ? ;

X = c ? ;

no
```

As the following trace shows, each time a goal succeeds, it is by choosing the first clause of `member/2`. Whenever an additional solution is requested, Prolog backtracks from the first clause of the definition and tries the second clause instead, until all possibilities have been exhausted.

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

 1 | 1 call member(_1630,[a,b,c])

 1 | 1 exit member(a,[a,b,c])

X = a ;

 1 | 1 redo member(a,[a,b,c])

 2 | 2 call member(_1630,[b,c])

 2 | 2 exit member(b,[b,c])
 1 | 1 exit member(b,[a,b,c])

X = b ;
```

```
 1 | 1 redo member(b,[a,b,c])

 2 | 2 redo member(b,[b,c])
 3 | 3 call member(_1630,[c])

 3 | 3 exit member(c,[c])

 2 | 2 exit member(c,[b,c])
 1 | 1 exit member(c,[a,b,c])

X = c ;

 1 | 1 redo member(c,[a,b,c])

 2 | 2 redo member(c,[b,c])
 3 | 3 redo member(c,[c])

 4 | 4 call member(_1630,[])

 4 | 4 fail member(_1630,[])
 3 | 3 fail member(_1630,[c])

 2 | 2 fail member(_1630,[b,c])

 1 | 1 fail member(_1630,[a,b,c])
```

Here is a modified version of the standard definition of `member/2`*; one with a cut in the first clause. How does its behaviour differ from the standard* `member/2`*, and why?*

```
memberchk(Item, [Item|_]):- !.

memberchk(Item, [_|Tail]):-
   memberchk(Item, Tail).
```

Exercise 13 *Do this exercise with pencil and paper; not on a computer.*
What is the function of the following program?

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

Provide a trace of the program executing the following call and state what the value of `X` *will be when the call terminates.*

```
?- pred(X, [the,talk,of,the,town]).
```

What would be the result of forcing the program to backtrack?

3.4 Final Programming Exercise

Exercise 14 *Write a Prolog program in which you type in an English sentence and Prolog replies with another sentence that is an altered version of the one you have typed in. For example, the program might produce an interaction like the following:*

```
You: you are a computer

Computer: i am not a computer

You: do you speak french

Computer: no, i speak german
```

It is easy to write a program to do this by simply following these steps:

1. *accept a sentence that is typed in by the user*
2. *change each 'you' in the sentence to 'i'*
3. *likewise, change any 'are' to 'am not'*
4. *change 'french' to 'german'*
5. *change 'do' to 'no'*
6. *add a comma and a space after 'no',*

You should take the input to be a list; so that the interaction goes:

```
?- reply.
|:[do, you, know,french].

no, i know german

yes
```

This program is just a variant of the one we constructed in class to translate English into French, so you can model this program on that one.

A difference between this program and the French translator, however, is that here ***'reply'*** *is a zero-place predicate; to accept input from the user, it must contain a call to* ***read/1*** *as part of its definition. Similarly, Prolog's answer has to be produced using the* ***write/1*** *predicate. Note that the Prolog response does not contain any brackets or commas; you therefore will need to write a (recursive) predicate which will 'write' the items on a list one by one, with a space between each one, until it reaches the end of this list and stops - and integrate it into your program.*

Write your program so it will run indefinitely, until the interaction is terminated by the user.

4 Program examples

4.1 Natural language processing

We'll conclude first with an example of rather more linguistic relevance than the preceding ones, which deploys a lot of what we have covered so far.

Most computer programs which handle natural language input contain a component known as a *parser*. This is a program which assigns a linguistic analysis to the input (a *parse*).

Parsing is a big topic, which L433 Introduction to Computational Linguistics covers in some depth, but we will present an example here.

4.1.1 Grammar and lexicon

We need to supply the program with information about the grammar and lexicon of the language it will be dealing with.

So, first we need to decide on a representation for these. We will keep things very simple and try to stay close to what elementary linguistics texts provide.

Linguists are in the habit of writing rules such as

$S \Longrightarrow NP\ VP$

$V \Longrightarrow likes$

We can keep quite close to this kind of respresentation in Prolog by defining $\Longrightarrow$ as an infix operator:

```
 :- op(1050, xfx, '==>').
```

We can now specify a grammar as follows, keeping to the Prolog conventions of using lower case letters for constants, commas to conjoin items and full-stop to terminate a statement.

```
s ==> np, vp.
np ==> det, n.
vp ==> v, np.

det ==> the.
n ==> dog.
n ==> cat.
v ==> chased.
```

4.1.2 The parser

The parser will be defined as a predicate `parse/3`, taking three arguments:

1. its first argument will be a syntactic category
2. the second will be the string of words to be parsed (represented as a Prolog list)
3. the third will be the string of words left over when the parse is complete

`parse/3` will return 'yes' if the string can be analysed as an instance of the syntactic category. So we will get interactions like the following:

```
?- parse(s, [the,dog,chased,the,cat], []).

yes
?- parse(np, [the,dog,chased,the,cat], []).

no
?- parse(vp, [chased,the,cat], []).

yes
?- parse(np, [the,cat], []).

yes
```

The parser that we will describe here operates (like Prolog) in a left-to-right, top-down, depth-first fashion.

We start by seeing how the syntactic category we are seeking (`Cat`) is defined by the grammar. There are two possibilities

1. the category introduces a lexical item
2. it introduces one or more syntactic categories

In the first case, we compare that first item in the string of words with the right hand side of a rule which has `Cat` as its left-hand side. If they match, we have parsed the first word, with the rest of the string left over (i.e. the third argument):

```
parse(Cat, [Firstword|Restwords], Restwords):-
    Cat ==> Firstword.
```

In the second case, we need to look up a rule which has `Cat` as its left-hand side, find out what its daughters are and parse them. Because `parse/3` is expecting only a single category as its first argument, and there may be more than one daughter in a rule, we need to define a new predicate to handle the parsing of the daughters of a rule: `parsedaughters/3`:[9]

```
parse(Cat, String, Rest ):-
   (Cat ==> Daughters),
   parsedaughters(Daughters, String, Rest).
```

That is `parse/3` dealt with. We now have to define `parsedaughters/3`. Again, there a two cases to consider:

1. there is more than one daughter
2. there is exactly one daughter

If there is more than one, we pass the first and the string to be parsed to `parse/3` to deal with, and then pass the remaining daughters and what is left of the string to `parsedaughters/3` to work on further:[10]

```
parsedaughters((Cat, Cats), String, Rest):-
    parse(Cat, String, Next), parsedaughters(Cats, Next, Rest).
```

If there is exactly one, we just pass it and the string to be parsed to `parse/3`:

```
parsedaughters(Cat, String, Rest):-
   parse(Cat, String, Rest).
```

[9]We need to put the rule `(Cat ==> Daughters)` in brackets to get the precedences right. With `==>` given precedence 1050 and the comma given precedence 1000, an expression such as `s ==> np, vp` will be treated by Prolog as `==>(s, (np, vp))`, with the comma binding more tightly than `==>` (which is what we want). However, in this rule, we are also using the comma in its basic Prolog function, as the 'and' operator, and we want the body of rule to be analysed as `,(==>(Cat,Daughters),parsedaughters(Daughters,String,Rest))`, with the comma binding *less* tightly than `==>`. The only way we can get *both* results is to explicitly bracket the sub-expression which we want the 'and' comma to treat as a single unit.

[10]One subtlety here is that the comma is defined in Prolog as a right-associative infix operator. That means that a sequence of items separated by commas, such as `a, b, c, d` is to be analysed as if it were written `(a, (b, (c, d)))`.

That's it.

The complete code is:

```
parse(Cat, [Firstword|Restwords], Restwords):-
    (Cat ==> Firstword).

parse(Cat, String, Rest ):-
   (Cat ==> Daughters),
   parsedaughters(Daughters, String, Rest).

parsedaughters((Cat, Cats), String, Rest):-
   parse(Cat, String, Next),
   parsedaughters(Cats, Next, Rest).

parsedaughters(Cat, String, Rest):-
   parse(Cat, String, Rest).
```

Finally, here is a variant, `parse/4`, whose extra (second) argument builds a tree of the parse - represented like a bracketed string.

```
parse(Cat, [Cat,Firstword], [Firstword|Restwords], Restwords):-
    Cat ==> Firstword.

parse(Cat, [Cat|Tree], String, Rest ):-
   (Cat ==> Daughters),
   parserest(Daughters, Tree, String, Rest).

parserest((Cat, Cats),  [Tree, Resttree], String, Rest):-
   parse(Cat, Tree, String, Next),
   parserest(Cats, Resttree, Next, Rest).

parserest(Cat, Tree, String, Rest):-
   parse(Cat, Tree, String, Rest).
```

Here is an example:

```
?- parse(s, T, [the,dog,chased,the,cat], []).

T = [s,[np,[det,the],[n,dog]],[vp,[v,chased],[np,[det,the],[n,cat]]]] ?

yes
```

4.2 A propositional logic interpreter

This program to demonstrate the use of operators in Prolog.

1. Logical connectives `and`, `or`, `if` and negation are defined as operators and given the same semantics as their Prolog counterparts.

2. A small procedure `play/0` is provided to allow the answers 'true' and 'false' to be supplied and

3. there is a simple front end to present an interface for the user to interact with

The program starts automatically when the file is consulted and presents a prompt `>`. The user inputs a sentence in propositional logic and the program outputs its truth value and then re-presents the prompt. The program terminates when the user types `stop.`.

```
?- [logic].
%logic compiled

> p.
true
> q.
true
> r.
false
> p and q.
true
> p and r.
false
> p or r.
true
> r or p.
true
> p implies q.
true
> p implies r.
false
> r implies p.
```

```
true
> r implies r.
true
> p implies ~r.
true
> stop.
goodbye

yes
```

4.3 The program

4.3.1 Operator definitions – syntax

```
:- op(1100, xfy, implies).

:- op(1000, xfy, or).

:- op(900, xfy, and).

:- op(800, fy, ~).
```

`or`, `and` and `implies` are defined as infix operators.   is defined as a prefix operator. Note that these definitions are unusual in that they must be preceded by the `:-` operator. Omit this and you will get an error message to the effect that you are trying to redefine a system predicate. Defining a new operator consists of satisfying a goal which as a side-effect assigns the relevant precedence and associativity value to the operator.

4.3.2 Operator definitions – semantics

`or` is defined as Prolog disjunction:

```
or(X, Y):- X;Y.
```

`and` is defined as Prolog conjunction:

```
and(X, Y) :- X, Y.
```

is defined as Prolog negation as failure:

```
~X :- \+ X.
```

The definition of `implies` is a little more cunning. The following is a standard tautology of propositional logic:

$p \supset q \equiv \neg p \vee q$

as can be seen from the following truth table:

(p	⊃	q)	≡	((¬	p)	∨	q)
t	t	t	t	f	t	t	t
t	f	f	t	f	t	f	f
f	t	t	t	t	f	t	t
f	t	t	t	t	f	t	f

We can use this equivalence to define `implies` like this:

```
implies(X, Y):- \+ X ; Y.
```

4.3.3 The database

Declare `p` and `q` to be true.

```
p.
q.
```

We assume that everything other than `p` and `q` is false, but we need to tell Prolog that we haven't simply forgotten some of our definitions

```
:- prolog_flag(unknown, _, fail).
```

This allows Prolog to simply fail if given an unknown predicate.

4.3.4 The front end

The top-level predicate `play/0` writes a prompt and accepts input with the built-in `read/1` predicate, then evaluates the input.

```
play:-
   write('> '),
   read(X),
   evaluate(X).
```

`evaluate/1` has three cases:

1. If X has the value 'stop', then write 'goodbye' and terminate.

   ```
   evaluate(X):-
      X = stop,
      write(goodbye), nl.
   ```

2. Call X and if it succeeds, write 'true' and continue, via another call to play/0.

   ```
   evaluate(X):-
      X,
      write(true), nl,
      play.
   ```

3. Otherwise, write 'false' and continue.

   ```
   evaluate(_):-
      write(false), nl,
      play.
   ```

Finally, we insert a directive at the end of the file which Prolog will attempt to evaluate as soon as it consults it:

```
:-play.
```

4.3.5 Complete program listing

```
/* file logic.pl */

/* A simple program to demonstrate the use of operators in Prolog.
Logical connectives and, or, if and ~ (negation) are defined as
operators and given the same semantics as their Prolog counterparts.
A small procedure play/0 is provided to allow the answers 'true' and
'false' to be supplied.  To run the program, type 'play.'  and then
input a propositional calculus expression, followed by a full stop and
carriage return.  e.g.

?- play.
```

```
> |: p and q.
true
> p and r.
false
> r and r.
false
> q and r.
false
> q or r.
true
> p implies q.
true
> p implies q and r.
false
> p implies q or r.
true
> stop.
Goodbye

Type 'stop.' to finish. */

% Operator definitions

:- op(1000, xfy, or).

:- op(900, xfy, and).

:- op(800, fy, ~).

:- op(1100, xfx, implies).

% Allow Prolog to simply fail if given an unknown
% predicate

:- prolog_flag(unknown, _, fail).

% Definitions of connectives
~X :- \+ X.
```

```
and(X, Y):-  X , Y.
or(X, Y):-   X ; Y.

% This one is cunning, 'X implies Y' is defined with
% the semantics of  (~X) or Y -- which has the same
% truth conditions.

implies(X, Y):- \+ X ; Y.

% p and q are true.
p.
q.

% Top level. Write a prompt and accept input. Then
% evaluate input.
play:-
   write('> '),
   read(X),
   evaluate(X).

% If X is 'stop', then output 'goodbye' and terminate.
evaluate(X):-
   X = stop,
   write(goodbye), nl.

% If X succeeds, write 'true' and continue.
evaluate(X):-
   X,
   write(true), nl,
   play.

% Otherwise, output 'false' and continue.
evaluate(_):-
   write(false), nl,
   play.

:-play.
```