Quick Guide to Prolog

Program Structure:

A Prolog program is a collection rules of the form:

<predicate name>(<list of terms>).

and rules of the form:

<predicate name>(<list of terms>) :- <predicate name> (<list of terms>),
<predicate name> (<list of terms>),
...
<predicate name> (<list of terms>).

Terms

Terms represent objects in the world. They are used as the "arguments" to predicates. They can be simple constant terms, like:

john
mike
male
penguin

Or complex terms. A complex term is built from other terms in the following way:

<term name>(<list of terms>)

some examples of complex terms:
s(0)
s(s(0))
color(sweater)

Complex terms look a lot like predicates, but they are **different**. Terms (both simple and complex) represent *objects* in the world, while predicates represent *relations* among terms. So, if it represents an object, it's a term, and if it represents a true/false statement about a group of terms, it is a predicate.

ALL PREDICATES AND TERMS NEED TO BEGIN WITH A LOWER CASE LETTER. PROLOG ASSUMES THAT ANYTHING THAT BEGINS WITH AN UPPER CASE LETTER IS A VARIABLE

Example Program:

```
parent(john, sue).
parent(john, mike)
parent(betty, sue).
parent(betty, mike).
parent(sue, linda).
parent(mark, linda).

male(john).
male(mike).
male(mark).
female(sue).
female(betty).
female(linda).
```

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

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :-  parent(X,Z),
                  ancestor(Z,Y).
```

Once a program is loaded into memory, you can query it:

```
?- parent(john, sue).
    yes

?- parent(X,sue).
    X = john     ;                    The ';' is a way to ask if there are any other possible answers
    no
?- parent(john, X).
    X = sue      ;
    X = mike     ;
    no

?- ancestor(X, linda).
    X = sue          ;
    X = mark         ;
    X = john         ;
    X = betty        ;
    no
```

Lists

Lists are a special type of complex term. Some examples are:

```
[betty, john, mark]
[sue, mike]
[john]
[]
```

There is a special form for lists which can be used in predicates :

```
[First | Rest]
```

This form will match with any non-empty list: First gets bound to the first element in the list, and rest gets bound to the rest of the list (not unlike car and cdr from lisp).

This can be more easily seen with an example. Assume there is the fact:

```
list([a,b,c,d]).
```

then the query list([First | Rest]) will give the following result:

```
?- list([First | Rest])
    First = a
    Rest = [b, c, d]
```

More Programming Examples:

One predicate that is useful is the equality predicate, true when two terms are the same. It is very easy to write:

```
equal(X,X).
```

List programming

Now we'll look at some list programming examples:

/* **first(Elem,Lst)** True if Lst is a list, and Elem is the first element of that list */

Like many predicates that deal with lists, first will use the [X | Xs] form:

```
first(Elem,Lst) :- equal(Lst, [FirstElem | Rest]),
                   equal(Elem, FirstElem).
```

Now, this is a perfectly good definition of first, but the equals are actually not necessary, and first can be written more simply as:

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

Now we'll look at a slightly more complicated predicate :

/* **member(Elem,Lst)** True if Lst is a list, and Elem is a member of Lst */

This rule will be recursive, and we need to think of what the base case and the recursive case are.

Base case : Elem is the first element of Lst
Recursive case : Elem is a member of the rest of Lst

So, looking at each of these cases:

```
member(Elem, Lst) :- first(Elem,Lst).
member(Elem, Lst) :-  equal(Lst, [First | Rest]),
                      member(Elem, Rest).
```

As before, it turns out that we don't actually **need** the equals predicate, and member can be rewritten as:

```
member(Elem, [Elem | Rest]).
member(Elem, [First | Rest]) :- member(Elem, Rest).
```

The Programming Environment

Prolog programs are a collection of facts and rules. You can write your prolog programs using any text editor that you like. Once you have written your Prolog code, you need to be able to test it, by giving it queries. Here are the steps:

- Compose your Prolog program in a text editor, save it as <filename>
- Start the Prolog interpreter, gprolog
- Load your code into the interpreter with the command ?- consult('filename').
- Query your code
- When you are done, the command ?- halt. will exit Prolog

Once you start querying your code, you will find that it probably doesn't work quite the way that you want. So, make changes to your code file, save the changes, and then reconsult the file.

Comments

Anything in your source code surrounded by /* */ is considered a comment, and is ignored by the interpreter.

Warnings & Errors

Any time a variable occurs only once in a rule, you will get a warning. So consulting the following example

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

will give the warning :

```
compiling /.automount/nexus/n/home/galles/cs345/tst.pl for byte code...
/.automount/nexus/n/home/galles/cs345/tst.pl:2 warning: singleton variables [Rest] for first/2
/.automount/nexus/n/home/galles/cs345/tst.pl compiled, 15 lines read - 1455 bytes written, 27 ms
```

This warning says that in the rule for the predicate first, which takes two terms, the variable Rest only appears once. It is unusual for a variable to appear only once in a rule – the interpreter gives you a warning because you might have made a mistake typing in the code. The code still works fine, even with the warning. You can avoid these warnings by beginning each variable that appears only once with an underscore:

```
first(Elem, [Elem | _Rest]).
```

Variables that begin with an underscore are called anonymous variables, because you don't care what their value is. Basically, you are telling the interpreter "I know that this variable only appears once, don't warn me about it"

Quitting Prolog

You can quit the prolog interpreter with the command:

```
?- halt.
```

An end of file symbol (control-d) will also quit the interpreter.