Logic Programming using Prolog
an Introductory Exercise

Brandon Bennett

The purpose of this exercise is for you to see some examples of the Prolog logic programming language in action. Prolog is very different from procedural languages such as C, Java or Python, and is generally regarded as being quite difficult to learn (though very powerful and flexible once you have mastered it). Nevertheless, it will be very good for you general appreciation of programming techniques to get some idea of how it works. This exercise is intended to be illustrative. It should give you a flavour of the language, although it won't really teach you enough to program on your own.

To find out more about Prolog you can refer to one of the following web pages:

- The *Learn Prolog Now* tutorials at `http://www.learnprolognow.org`
- The SICStus Prolog home page at `http://www.sics.se/sicstus`

Preparation

Before starting the exercise you should do the following:

- Make a directory to store your Prolog code for this exercise (e.g. `~/Prolog/Intro-ex`).[1]
- Copy the files `family.pl` and `bb_planner.pl` (which should be on the VLE in the same folder as this document) to your own `Intro-ex` directory.

1. A Family Relationship Database

We shall start by extending the family relationships program that was introduced in the lecture on Logic Programming. Work through the following steps:

- First, open up the file `family.pl` in an editor and start up SICStus Prolog in the same directory as where this file is loaded.

  To start Prolog follow the instructions on the VLE item from where you downloaded this file. (If you have any problems starting Prolog, ask one of the helpers in the lab.)

- Now load the `family.pl` file. At the Prolog prompt type:

```
[family].
```

We shall begin by using Prolog directly at the command prompt. We shall enter *queries*, and Prolog will output matches for the *variables* occurring in these queries. It will determine these matches by searching through the *facts* and *rules* in our Prolog program file (`family.pl`). Recall that, a variable is any word starting with a capital letter. The output values will often be followed by a '`?`' symbol. This means that Prolog thinks there may be

[1]Recall that `~` at the start of a Unix or Linux directory path refers to the current user's home directory (and `~bilbo` refers to the home directory of the user called '`bilbo`)'.

other values for the variable(s) that also match the query. To see further values you should type in a semi-colon (‘`;`’).

a) Experiment with entering the following queries:[2]

   i) `?- parent_of(mary, X).`
   ii) `?- parent_of(X, Y).`
   iii) `?- grandfather_of(X, sue).`
   iv) `?- grandfather_of(X, Y).`

b) Try entering some more facts into the `family.pl` file. Add some more relationships involving additional people. As well as using the relation types already defined, add some `married` relations, and any other relations you fancy. After adding these facts, load your file again and try redoing some of the queries from the previous step, to see what you get.
(Tip: For convenience, I have added a predicate ‘`lf`’, in the `family.pl` file. This is defined so that if you enter ‘`lf.`’ as a query it will reload the file.)

c) Now you are ready to define some more *rules* for deriving further relationships from the information. Define rules defining each of the following. (Be sure to add these one at a time; and check they work correctly by reloading your program and entering suitable queries to test them.):

   i) `mother_of`
   ii) `brother_of`
   iii) `uncle_of`
   iv) `cousin_of`

2. Matching and Manipulating Lists

One of Prolog's fundamental data structures is the *list*. A list is a series of data entities, which is represented by a comma-separated sequence enclosed in square brackets. The elements of a list may themselves be complex data structures, but in this exercise we shall confine our attention to cases where the elements are what in Prolog terminology are called *atoms*. These are simply lower case words.

a) Two of the most common predicates that are used to manipulate lists are `member/2` and `append/3`). (Note that, in referring to a Prolog predicate, it is usual to use the form `predname/`*n*, where *n* is the number of argument places of the predicate — i.e. the number of data entities it relates.)

   Experiment by entering the following queries and noting what output you get. As before, you should use the ‘`;`’ to check for multiple possible answers.

   i) `?- member( b, [a, b, c, d] ).`
   ii) `?- member( X, [a, b, c, d] ).`
   iii) `?- member( X, [a, b, c, d] ), member(X, [c, d, e, f]).`

[2]Unfortunately, when using Prolog in a shell window, you don't get very good facilities for correcting typing mistakes. Depending on the shell you are using, backspace may not work properly. If not you may be able to use *ctrl*-`h` instead.

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

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

b) To do the following sections, you will need to load a library, which defined additional list manipulation predicates. To do this enter:

```
use_module(library(lists)).
```

You can now try the following predicates. Again, remember to use the ‘;’ to see further possible answers.

i) `?- nth1( 2, [one, two, three, four], N ).`

ii) `?- last( [one, two, three, four], X ).`

iii) `?- length( [x, y, z], Len ).`

iv) `?- select( X, [a,b,c,d,e], Remainder ).`

v) `?- permutation( [this, and, that], P ).`

c) Now let us look at how these list manipulation predicates can be combined to define more complex relationships involving lists. Open a new file called `listops.pl` and use this to try the following. (As usual you should load and test your file after each step.)

i) As a first example, try the following definition:

```
mixup( L1, L2, Mix ) :- append( L1, L2, Join),
                        permutation( Join, Mix ).
```

Test this with the following query:
`?- mixup( [a,b,c], [x,y], M).` (How many solutions is that?!)

ii) Now, by making use of the `select/3` relation, define a predicate
`get_two/3`,
such that `get_two(List, X, Y)` will be true when `X` and `Y` are any two members of `List`. This should be defined so that `X` and `Y` can only take the same value if `List` contains two or more occurrences of the same atom.

Test this predicate by trying a few queries using `get_two/3`. What will happen if `List` only contains one atom?

d) Often we want to split a list up so that we separate its first element (its *head*) from all the remaining elements (its *tail*). This is done using the special matching construct `[H | T]`, where the variable `H` will match the head of a list and `T` will match its tail.

For example, try the following query:
`?- [H | T] = [head, followed, by, the, tail].`

More, generally we can use a similar construct to match several elements at the beginning of a list and also any remaining elements. For example, try:
`?- [H1, H2, H3 | T] = [one, two, three, and, all, the, rest].`

As well as being used to divide up a list, the `[H | T]` syntax can be used to add a new head to a list. To see this, enter the following query and look at the value that gets assigned to `NewList`:
`?- List = [one,two,three], NewList = [zero | List].`

Notice that in Prolog ‘=’ is a matching relation, not an assignment, so we get exactly the same effect by entering the query:
`?- [one,two,three] = List, [zero | List] = NewList.`

3. Using a Logic-Based Planning Program

Planning, is a sub-field of the *Knowledge Representation and Reasoning* research area. It is concerned with finding sequences of actions which could be employed to achieve some goal.

To illustrate this we consider an old puzzle known as the *Missionaries and Cannibals* problem. The scenario runs as follows:

> There is group of three missionaries and three cannibals, who are attempting to cross a river (under the guidance of the missionaries). They all start on one bank and have a canoe that can carry any two people across. The aim is, by a series of crossings, to get all six people to the other side. The catch is that if there are ever more cannibals than missionaries on either side of the river, the cannibals will eat the hapless missionaries.

To get a good idea of the problem and its complexity, you can play with a simulation at the following web page:

`https://www.game.st/game_276_missionaries_and_cannibals.html`

The idea of this part of the exercise is to write a program which will automatically find a solution to this problem — i.e. a sequence of states which take us from the initial state to the goal, with every transition being a legitimate boat crossing (and no uncivilised meal times occurring en route).

This is a complex problem, requiring a sophisticated solution. However, I will give you most of the code ready written, and you only have to add some details. This may still be challenging, but given the examples in the exercises above and the step by step instructions that follow, you should be able to write a working program.

Remember to load your program after each step to check for any syntax errors. Try using simple queries to test the predicates as you define them to check they work as you intend.

Here goes:

a) View the file `bb_planner.pl` (that you copied right at the beginning) and *carefully* read the explanation at the beginning of the file. (Don't worry if you don't yet understand what is meant by a *state representation* and a *transition rule*; we are just coming to that.)

b) In order for the planner to work, it must manipulate *states* representing the scenario under consideration — in this case the locations of missionaries and cannibals on the banks of the river.

   We can represent a group of objects by a list. For instance: `[boat, canni, canni, missi]` and since there are two banks, we can represent the whole situation at any given point in the scenario by two lists. Prolog provides flexible syntax for combining data entities to form more complex structures. One simple way to do this is using the plus symbol, '`+`'. Thus, I can represent a state in this scenario by a structure such as:[3]

   ```
   [boat, missi, missi, canni, canni] + [canni, missi]
   ```

   This corresponds to the sate where the boat, two missionaries and two cannibals are on the left side of the river, and one canibal and one missionary are on the right side.

   Use this notation to formulate the `initial_state` and `goal_state` declarations described in `bb_planner.pl`, and add these to your own file `cannibals.pl`.

[3]Note that in this representation '`+`' has no special meaning. It is simply used to combine two lists together to form a more complex data structure. I could have used any binary operator — such as '`-`' or '`=`'.

c) You now need to define the `equivalent_states/2` predicate, which gives the conditions under which two states are equivalent. Notice that the ordering of elements within each of the two lists in our state structure is not important. This predicate can be written fairly simply by using the `permutation/2` predicate. (Test your definition by making some sample queries using `equivalent_states/2` applied to both equivalent and non-equivalent states to see if it gives the right answer.)

d) Now comes one of the most important parts. You need to specify possible transitions from state to state. Here is an example rule for one possible type of transition:

```
transition( Left + Right, Remaining + [boat, X | Right] ) :-
                                        select(boat, Left, Others ),
                                        select(X, Others, Remaining ).
```

Study this and try to figure out what it means. Then try to write additional rules in a similar style, which specify other ways in which one can go from one state to a possible subsequent state. (You should only need a few of these. Be sure to test them with queries.)

e) One also needs to specify the `legal_state/2` predicate, which specifies the conditions under which a state is allowed — i.e. the cannibals should not be able to eat any missionaries.

This is slightly tricky but note the following: one can examine each of the sides of the river separately to determine whether it is 'snacktime'. All the possible 'snack' situations correspond to permutations of a small number of lists of `canni` and `missi` elements (which may or may not also include the `boat`).

You can define a separate `snacktime` predicate and then define a `legal_state` to be one where snacktime occurs on neither side of the river. Recall that to add to a rule the condition that a given predicate is not true, you can precede the predicate with the negation operator '`\+`'.

(Test your `legal_state/2` predicate thoroughly.)

f) And finally, you can see if your program can solve the Missionaries and Cannibals puzzle.

- Load the final version of your program. (`bb_planner.pl` should get loaded automatically, if you have followed the instructions at the top of `bb_planner.pl`).
- Enter the query: `find_solution.`

If you have coded your problem specification predicates correctly the program will chug away for a short time reporting that it is trying to find solutions of increasing length. If all is well it should eventually find a sequence of states corresponding to a solution, which it will output to the screen.

Check to see if this solution is correct.

(At this point you may find you don't get what you want. This is normal. Have a go at figuring out what might be wrong. If you can't fix it, ask for help.)

I hope you enjoyed this rather challenging exercise and that it has given you some insight into the kinds of advanced programming problems tackled by researchers.

Brandon