Supplement 8

Jocelyn Paine

June 4, 1996

1 Introduction

These exercises give some more practice in using Prolog as a control language.

2 The caves world

Remember that this is described by knowledge base DD, which contains three kinds of fact:

```
is_a( a, cave )
meets( b, a ).
is_at( dragon, b ).
```

Supplement 2 described what these mean.

Assume that the knowledge base also contains a clause for each player giving their score: `score(fred,500)`. In Supplement 7, we saw how this might be updated as a player kills more of his enemies.

Write a predicate which has no arguments called `check`. When `check` is called, it examines the score: if it is greater than 2000, it writes a message saying "You have won", otherwise it does nothing.

When you have done that, enter some `is_worth` facts defining the values of the enemies, as suggested in Supplement 5: `is_worth(dragon,500)` and so on.

Then write a predicate called `kill(P,A)`. Here P is a player (e.g. `fred`) and A is an animal (e.g. `dragon`). What `kill` should do is to assume that P has just killed A, and to remove the old `score` fact, replacing it by a new one in which the worth of A has been added to P's score. You can do this with `gain`, as described in Supplement 7. Because A is now dead, `kill` should also replace the old `worth` fact for A by one giving it a worth of 0.

You can build on this and the other cave world exercises to write a complete Adventure game. Its players could be human, or they could be automatic like the traders in Traveller. In the former case, the program would ask what action the player wanted to take. This could involve some natural language analysis: it would be nice if a player could type `Throw firebomb at Wumpus`, `Eat packed lunch` or `Run South-West to` `the blasted tree`. In the latter case, the behaviour of the players would be dictated by predicates, defined perhaps like `will_be` in Supplement 2, but more complicated. The file DDMOVE, which you can `showlib`, is just a start in this direction.

3 Links

Please load knowledge base LINKS2. This is like LINKS, but instead of describing just one straight chain, it describes a tree-like network. As before, you have clauses for `goes_to(X,Y)` if X leads to Y, but now X may have more than one successor.

You also have clauses for `weight(X,W)`, where X is a square and W is a numerical weight associated with it. The idea is that squares with the lowest weights define a preferred route. Your job is to write a predicate like `look`, but it has to take the lowest-weight successor at each square.

This is not as hard as it sounds. First, assume you have a predicate `best(X,Y)` which, given square X, returns the lowest-weight successor in Y. If you have this, you can easily write `look`, just by copying the code in the lesson.

So how do you write `best`? Look back at the end of Supplement 6. This should give you enough information.

3.1 Search

The previous section involves a kind of search known as *hill-climbing*, where if you have several choices at any point, you calculate a numerical measure (a *weight*) for each, pick the best, and explore that, proceeding in the same way for all subsequent choices. In LINKS2, the weights were pre-calculated by me, and they have no obvious relevance to anything else. However, you can apply the same method to problems where the weights are not known in advance.

Consider the problem of synthesising some new wonder-compound, say interferon. There is usually more than one reaction that will produce a given compound. The chemical literature is full of lists of reactions, the circumstances under which they can be applied, and the chemical needs and costs of each. So — we start at interferon. We make a list of all reactions that can plausibly be used. Think of each as a path leading off from interferon. From general principles, we can then assign a weight to each. Some, such as the one where the reactants are so unstable that their molecules need to be isolated inside individual micro black holes, and the one that requires you to boil the reactants in a 50:50 hydrofluoric acid/liquid radium mixture inside a 10 cubic meter diamond reaction vessel, will be rather costly — i.e. high weight. Others, like the one where you distil the reactants from a kilo of fermented garden slugs, will be less so. Having assigned the weights, pick the lowest. That gives you the cheapest reaction. You then proceed in the same way with its starting materials, and so on. A similar procedure is in fact used in commercial programs for synthesis planning.

My first attempt at the `route` predicate of Supplement 5 worked in this way. At each square, it looked at all those leading from it. It then worked out the distance of each to the destination, picked the nearest, and then moved on to it as the next square. Because of bends in the streets, this did not always give the shortest path, but it was a good first approximation.

4 Lists again

For our final example, I will look at lists again, and show how they can be combined with the other things you have learnt. During the 17th Century, "canting academies", or books of criminal slang were popular. One such, available in the Bodleian Library, and written by Oxford author Richard Head — best known for his debased and much-banned autobiographical *The English Rogue* — is entitled just that: *The Canting Academy*. It contains a number of entries which will tell you for example that the slang term for a sheep was "bleating cheat", and for prison, "queer ken".

I have copied part of *The Canting Academy* into the file HEAD, which defines the predicate `dictionary(Normal,Slang)`. Now the obvious way to write such a predicate is to use atoms: for example, if we were converting between English and French, and ignoring niceties like gender:

```
dictionary( chat, cat ).
dictionary( chien, dog ).
```

However, because many of Head's entries consist of more than one word, it is better to use lists, which are the ideal way to store variable-length sequences.

To see how these are used, have a look at, and experiment with, the files HEAD and READER. These contain the dictionary, together with a simple program that reads normal words or phrases from the terminal and looks up the slang equivalent. These combine all the features of Prolog you have seen so far, including lists, command predicates, logic predicates, input, and output. Unlike in Lesson 7, I have not used `read` and `write` for input and output: as I said then, these are not very convenient. Instead, I have written my own input-output predicates. By looking at the program, and the predicate names, see whether you can work out which these are, and how they are used.