Supplement 6

Jocelyn Paine

June 4, 1996

1 Introduction

This supplement gives some practice with arithmetic and `not`. It also talks about how the Traveller route-planner works, and introduces *lists*, which you will meet again after leaving the Tutor.

2 Traveller

2.1 Some arithmetic

In Supplement 5, I asked you to amend COLLIER so that he would buy more fuel if he was at a fuel station and had less than 10 units left. At the time, you only knew how to make him buy a fixed amount, regardless of his need.

Now, using `is`: can you amend your definition (or that in COLLIER2), so that he always buys exactly what's needed to refill his fuel tank up to its limit?

Now, consider moving from Colliery Row to The Hub. Even if you do make a coal trader who always buys exactly the right amount of fuel, he will eventually run out of money, because the net cost of fuel on each cycle is greater than the profit to be made from coal. The Hub offers richer pickings; both televisions and glasses should (barring chance events) enable a careful trader to make a steadily increasing profit. Can you program a Hub trader?

2.2 Gateways

I have written the code of a more intelligent trader. One of the predicates in it is `gateway(Region,Square)`. This is true if Square is a gateway to Region; that is, if Square is inside Region, but also connects to a square that is outside it. What is wrong with the definition below?

```
gateway( Region, Square ) :-
    in( Square, Region ),
    adjacent( Square, Other ).
```

2.3 Using `not` to find the cheapest goods

Having found and corrected the error in `gateway`, have a look at `nearest_gateway`, below. The idea is that `nearest_gateway(From,Region,Gateway)` is true if Gateway is that gateway to Region which is nearest to From. It could be used by a trader who has started off some way from a particular region, and who wants to move into that region before starting to trade.

```
nearest_gateway( From, Region, Gateway ) :-
    gateway( Region, Gateway ),
    distance( Gateway, From, D ),
    not((
            gateway( Region, Other ),
            distance( Other, From, D1 ),
            D1 < D
       )).
```

Translate the definition into English. There is a bug in it (one I committed myself when writing the game); what is it?

Now, one can write a structurally similar predicate for finding the cheapest wholesaler. Call it `cheapest_seller_in_region(Good,Region,Square)`. It is true when Square offers the lowest price of any square selling Good within Region. Define it!

You will find the answers in TRADER..

2.4 Searching for regions

Suppose we wanted to find all regions R not equal to a given region ThisR (which might be, say, `'The Hub'`) Would the question `R \= ThisR?` find them? If not, what would?

3 Route finding and lists

I talked earlier about a trader that might, using `nearest_gateway`, begin at a random point on the board and make his way to a particularly favourable region before starting to trade. In order to do this, we need some means of finding a route between two squares. In fact, it ought not to be *a* route; it ought to be the *shortest* route.

Supplement 2 asked you to write `on_route_to` for the caves world. If you were happy with recursion, you may have succeeded in producing something logically correct. However, your predicate probably had several disadvantages:

1. If the goals were in the wrong order (something technically called left-recursion), the predicate may have got into an infinite loop
2. Even if not, it may have got trapped in a loop if there were closed paths in the space being searched. This is almost certain, since I have not yet introduced the techniques necessary to prevent this.

3. Even if the predicate avoids looping, it might find a path that is not the shortest. The two problems above are specific to Prolog; they arise because it is not faithful enough to logic. This third problem is language-independent. In order to write a predicate that finds shortest paths, you need to know a lot about *search*. This is a large topic in A.I. and operational research, and there are many different methods for writing programs to find shortest or least-cost paths in various types of network.

Traveller itself defines such a predicate: `route(Square1,Square2, Route)`. It is true if Route is the shortest route between Square1 and Square2. To see how it represents routes, load TRAVELLER, and then ask the question

```
route( 66, 11, R ).
```

You may notice a slight pause before the answer.

For the route between squares 66 and 11, `route` will give the answer

```
[28, 25, 26, 35, 34, 23, 24, 17, 33, 5, 6, 7, 29, 30, 31, 32, 11]
```

Tracing the sequence out on the board should convince you that it is the shortest route. It is represented as a new type of Prolog object, one you have not met before. In A.I., we frequently need to deal with sequences: sequences of words making up a sentence, sequences of actions making up a plan, sequences of points making up a path. This need was realised in the early days of A.I. computing, round about 1958, and gave rise to programming languages, the most famous of which is Lisp, with facilities for handling *lists*. Prolog also implements lists, and this is what `route` returns.

Lists are particularly valuable because they can vary in length. When I introduced `act` in Traveller, I said that if your action was `move`, the next argument had to be `dummy`. This is because `act` has a fixed number of arguments, and so an argument that's not needed by some action still has to be there. With lists we would not have this problem: we could represent actions as

```
[ move, 18 ]
[ buy, coal, 20 ]
[ sell, peaches, 5 ]
```

where we only use as many places as we need.

Another situation in which lists would make sense is the noughts-and-crosses game of Supplement 2. In this, we had facts like

```
o_moves_to( bbbbxbbbb, obbbxbbbb ).
o_moves_to( bbbbxbbbb, bobbxbbbb ).
```

where positions on the board are represented as atoms. This is not a good way to represent them, though it's all that was available at the time. For example, if we were printing out the result of a move, we would need to produce a diagram of the board. To do this, we must, for each square on the board, be able to determine whether it contains an O or an X. Now, although we can see that the 5th letter (corresponding to the middle square, say) of `bbbbxbbbb` is `x`, there's no way to discover this from within a program. Atoms are indivisible: that's why they're called atoms. However, if we represent board positions as lists

```
[ b,b,b,b,x,b,b,b,b ]
```

then it is possible to write a recursive predicate that returns the Nth item of any list. We do in fact have a stock list-handling predicates, including a "find Nth element" one, if you need them later.

Lists are also good for representing sentences. Later on in the course, you will see how to use lists of words in building your own Eliza; to tell it for instance that the response to any list `[my,R,hates,me]` where R names a relative (e.g. sister) is `[tell,me,more,about,your,R]`, while that to `[i,like,A]` where A names an action (e.g. sailing) is `[you,say,you,like,A]`. It is possible to write general rules or templates that recognise certain kinds of list and transform them into other lists, and this gives Prolog a lot of the flexibility you need for A.I.