Supplement 7

Jocelyn Paine

June 4, 1996

1 Introduction

These exercises will give you some practice in using Prolog as a control language. They also talk about the idea that information can be represented in more than one way, and that how we represent it affects the efficiency with which our programs can process it. This is an important idea, and one central to computing (and indeed to A.I.). We don't have time to cover it in detail during the course, but I would like you to see at least a small application of the idea, one which can be appreciated without needing extensive knowledge of Prolog.

2 Reading and writing

First, please write a command predicate called `tax` that takes five arguments. The first argument is an initial; the second is a title (i.e. `'Mr'`); the third is a surname; the fourth is an address. All these are atoms. The fifth is a number, representing an amount of money in pounds. An example of its use: the question

```
tax( 'D', 'Mr', 'Jones', 'Honeysuckle Cottage, Woodtown', 2500 ).
```

should display the message

```
Attn: Mr. D. Jones, Honeysuckle Cottage, Woodtown.

Dear Mr. Jones.
    You owe the sum of 2500 pounds in taxes for the year 1988-89.
Unless you pay this within a period of 7 days,  your possessions,
including any children you may have  accrued, will be  liable  to
distraint in order to pay the above-mentioned sum of 2500 pounds.
```

Secondly, the knowledge base MOW contains a predicate of one argument, which must be a number. Load this by typing `library(mow)`, and see what it does with questions like

```
mow(0).
mow(4).
```

Write a predicate which types a sensible message asking for a number, then reads a number, and calls `mow` with that number as argument.

3 Changing the database

3.1 The caves world

In the caves world (file DD), assume that each player has his or her own score:

```
score( fred, 23 ).
score( tommy, 1000500 ).
```

The idea is that this would initially be zero for each player, and would increase with kills, as players wipe out their enemies (how they might do this was described in Supplement 5).

Write and test a predicate called `gain(P,N)` where P is the name of a player and N is a number. `gain` must add N to P's score. Thus, `gain(fred,20).` would up Fred's score from 23 to 43, deleting the old clause for `score` and adding a new one. We will use this in the next supplement which considers how you might write a complete Adventure game.

3.2 Traveller

Now to Traveller. As you know, there each trader T has a clause for `at(T,S)` recording that he is at square S. Write a predicate `moveto(T,Y)` which deletes the fact that T is at his old location (call it X), adds the fact that he is at Y, and writes a message saying "T has moved from X to Y". .

3.3 Reading in programs

Finally, try combining reading and database-changes. Can you write a predicate that reads two names (atoms) from the terminal, and adds a fact saying that one loves the other? So if the names are `romeo` and `juliet`, the predicate must add the fact `loves(romeo,juliet)`. There are not many languages in which you can read in new bits of program at run time!

4 Information representation

This is the section on representation and how it affects efficiency which I promised earlier. Please consider the following map of the High Street:

```
Carfax -> Turl -> All Souls -> Univ -> Schools
```

There are several ways in which this could be described by Prolog. I have listed four below:

A

```
point(carfax).
point(brasenose).
point(all_souls).
point(univ).
point(schools).
```

B

```
point( carfax, brasenose ).
point( brasenose, all_souls ).
point( all_souls, univ ).
point( univ, schools ).
```

C

```
point( carfax, 1 ).
point( brasenose, 2 ).
point( all_souls, 3 ).
point( univ, 4 ).
point( schools, 5 ).
```

D

```
point( carfax, 0 ).
point( brasenose, 34 ).
point( all_souls, 64 ).
point( univ, 96 ).
point( schools, 108 ).
```

In D, the numbers increase, but not in any predictable fashion. They might be, say, the distance of each point from Carfax.

For each of these, try to write a predicate called `next_place(Here,Next)`, which, given the input argument Here, returns in Next the following place in the sequence, i.e. if you ask `next_place(brasenose,Next).` then Next should become `all_souls`. If you can't see a way fairly quickly, then leave that part: there may not be one.

Now, consider the four representations again. For each, write out a new set of clauses describing the street without Brasenose (not a bad idea anyway). That is, remove whichever clause(s) define(s) it, such that the remaining clauses still represent a connected map, and so that `next_place` still works.

Next write out a new set of clauses for each so that they describe the street with a new point added, say The Body Shop. Again, `next_place` must still work on these new clauses.

Finally, for each of the four representations, try writing the predicate `nth_point(N,P)`, which is given a number N as input, and which sets P to the Nth point in the sequence. Thus, `nth_point(2,P).` should set P to `brasenose`. Again, if you can't see fairly quickly how to do it, leave that part.

4.1 Comments

The point of these exercises is to show that the same information can be represented in different ways, and that the representation affects the ease of doing things with it.

In (A), the places are ordered only by their position in the knowledge base. It is actually possible to write `next_place` and `nth_point`, but only by an advanced subterfuge which involves building a list of all the points, in the same way that the route-finder does, and then searching along it.

In order to insert a fact about the Body Shop, and preserve order, you have to insert it at the correct position in the knowledge base, leaving all the other clauses in the same relative position. The same goes for deleting a point. This is like a library where the books are catalogued only by their position on the shelf, and not by catalogue numbers written on the spine. When you have to insert a new book, it must go in the correct position, otherwise it will not be properly catalogued. And if there is not already a gap, this means that all the books on one side may have to shoved up to make room. The same is true here. If you want to add a point at run-time using `adda` or `add`, you have problems, because they can only add at the start or end of the database. It turns out in standard Prolog that the only way to add a new clause anywhere else is to delete all the clauses above where you want to add, insert the new one, and then replace all the deleted ones.

In (B), each point carries with it the address of its neighbour, and its relative position doesn't matter. Getting from one point to the next is easy, i.e. implementing `next` is easy:

```
next_place( X, Y ) :- point( X, Y ).
```

If you insert a fact about the Body Shop, or delete one about All Souls, you have to "tie up the ends" so that the sequence is not broken. If you don't do this, the new point, even though there, is inaccessible. Some people have suggested that something similar happens when we know we know something, but can't recall it: the memory is there, but we have lost our pointers to it.

Writing `nth_point` requires moving from one point to the next as many times as necessary: if you can't see how now, you should by the time you've done the LINKS exercises in the final supplement.

This very general idea, of associating with one piece of data the address of the next, is used in computing under the general name of *list processing*, and we say that one piece of data (e.g. `carfax`) *carries the address of*, or *points to* the next (e.g. `brasenose`). In the previous supplement you encountered the lists returned from `route`. These are a very specific instance of list processing. When Prolog displays them they look as though they are a single object, but the computer actually stores them in an analogous way to (B), with one element pointing to the next.

In (C), each point carries with it a numeric tag. If you add or subtract one from the tag, you get the tag for a neighbouring point. Getting straight to the Nth point is much easier than in (A), (B), or (D). This association of numbers with pieces of data, and the quick indexing it allows, is used in computing: we say that the points are held as an *array*, each *subscripted* or *indexed* by a given number. Arrays in other programming languages are stored in an analogous way.

One non-computational example of tagging is Littlewoods. At the meat and cheese counter, you used to have to queue. In 1985, "because of arguments about who was to be served next", the system changed so that you had to collect a numbered ticket from a dispenser, and then wait until the number is called. The information about order is now held in the tickets, and not in the arrangement of the queuers. This has been called the difference between *parking* and *marking* by Fairthorne in *Towards Information Retrieval*.

Another is book indexes. In principle, a book could be re-written as a giant index, which would list for every word in the book, the locations in the text at which it appears. This would be much easier to use when searching for a specified word, but not very pleasant to read! Nevertheless, exactly the same information is present in both.

(D) is a tricky one. As in (C), each place has a numeric tag, but the differences between the tags of successive points are not the same. To write `next_place(X,Y)`, you have to find the number for X, and then search for the least number bigger than it, using a method similar to that for `nearest_gateway` in Supplement 6. `nth_point` can only be implemented by repeating this process N times.

So the choice of representation is not trivial. Which representation you choose depends very much on how you will use the information.

4.2 Errors and efficiency

It is instructive to consider the effects of corrupting the database in each case. Suppose we wrote each clause on a punched card, and shuffled the cards, then this would only matter to (A).

If we were to delete a clause from each set, this would not matter much to (A), (C) or (D), but it would break (B) into two chains, whose relative position would then be unknown. This shows that the effect of errors on information depends very much on how the information is represented. This idea is often used in psychological experiments. Given some kind of error (e.g. a brain lesion), you can try to deduce from the behavioural changes it causes, how information is stored in the brain.

Now let's consider databases — not the Prolog database, but commercial conmputer databases. Let's suppose that we have a file of employees and their salaries: for example,

```
Chavasse      34000        Linton      12500
Hannington    20500        Matthews     9842
Latner        13000        Slinger     30000
```

If we want to search, as we might in a real database, for all employees with a salary of less than 12000, then as the file stands we would have to examine every entry. However, if we were to sort it in order of salary, we could stop as soon as we found one that was 12000 or over. Alternatively, we could build some kind of index giving the position of the first person earning 9000, 10000, 11000, and so on.

You can see how the time taken to find the information depends on the way in which we order and index the files. This is true not only of files, but of any data structure, and the study of such matters is a large and important part of computing.

Conversely, given some system, one can try to discover the internal representation by seeing how its response time varies with certain inputs — Supplement 1 had an example of this, under inference. As with errors, psychologists use response-time experiments to try and infer things about how our brains represent information.

4.3 A final example

The caves world offers a particularly clear example of alternative representations. It was set up using atoms such as `a` and `b` to name caves. We used the predicate `meets` to indicate that two caves were directly connected, so `meets(a,b)` meant that `a` was linked to `b`, with no other caves in between.

In an alternative representation, instead of naming caves, we can name *passages*. I have done this in the listing below.

```
is_a( p, passage ).     is_a( q, passage ).     is_a( r, passage ).
is_a( s, passage ).     is_a( t, passage ).     is_a( u, passage ).
is_a( v, passage ).     is_a( w, passage ).     is_a( x, passage ).

intersects( p, q ).     intersects( q, r ).
intersects( p, t ).     intersects( q, s ).
intersects( t, s ).     intersects( w, v ).
intersects( t, u ).
intersects( u, v ).
intersects( u, w ).
intersects( w, v ).
```

Here, `p` names a passage — a path that has a cave at each end, but not anywhere else along it. `intersects(p,q)` means that one end of passage `p` meets one end of passage `q`, and that there is a cave there. By drawing a complete map, you can satisfy yourself that these facts describe the same world as that introduced in Supplement 2.

How would you extend this style of representation to describe the location of the cave-world's inhabitants? What are its advantages or disadvantages?

4.4 Moral

There is always more than one way to represent any situation in Prolog. Each way is good for some tasks and bad for others.