Supplement 5

Jocelyn Paine

June 4, 1996

1 Introduction

This supplement gives you some practice with comparisons. It then moves on to a more advanced section (which you may want to skip if you start having difficulties) which talks rather more about how Prolog is executed and how it differs from logic.

2 The caves world

The first set of exercises continues with the world of caves from Supplement 2. Suppose that, as in many computer Adventure games, the world is populated by vicious animals, all inimical to human life. Each animal has a worth:

```
is_worth(dragon,500).       is_worth(gryphon,200).
is_worth(lion,50).          is_worth(vole,0.00172).
```

There are weapons, each with its own destructive power:

```
has_power( scud_missile, 10000 ).   has_power( supergun, 5000 ).
has_power( gun, 250 ).              has_power( grenade, 600 ).
has_power( water_pistol, 0.003 ).
```

We say, in a way typical of such games, that a weapon W can kill an animal A if W has more power than A is worth. Express this as a predicate `can_kill(W,A)` which is true if weapon W can kill animal A.

Assume also that the player can pick up and use weapons. Let the fact

```
holds( fred, gun )
```

mean that Fred is holding a gun: in general, let `holds(P,W)` be true if player P is holding weapon W. Can you now define a predicate `scorns(P,A)` which is true if player P holds a strong enough weapon to kill animal A?

These are all applications of comparisons and "and".

3 Sisters

In Lesson 2, I asked you to write a fact defining sisterhood. Can you repair that fact by using `\=`, so that it prevents someone being their own sister?

4 Traveller

4.1 Improving the coal trader

At the end of Supplement 4, I asked you to write a version of `act` that would drive a coal trader, but not to worry about fuel. Can you add a clause that will buy fuel if he has fewer than 10 units remaining?

You will find one possible answer in COLLIER2. The comparison operators allow you to program a test on the amount of fuel, but not to calculate the amount to buy as a function of other quantities. So thus far, you will have to write the amount in as a constant number. The next lesson deals with arithmetic, and shows you how to do such calculations.

4.2 More departures from logic

At the end of the skiing example in Supplement 1, I talked about the importance of writing mutually exclusive rules and showed two that aren't:

```
has_fitness( Skier, poor ) :-
    max_pressups( Skier, P ),
    P < 15.

has_fitness( Skier, good ) :-
    max_pressups( Skier, P ),
    P >= 10.
```

The problem with non mutually-exclusive rules is that they are logically inconsistent, since they state that two inconsistent things are both true. Logically, if the skier above could do a maximum of 13 pressups, he would have to be of both poor and good fitness!

Similarly, in COLLIER,

```
act( collier, buy, coal, 20 ) :-
    at_seller( collier, coal, Price ),
    carries( collier, coal, 0 ).

act( collier, sell, coal, 20 ) :-
    at_buyer( collier, coal, Price ),
    carries( collier, coal, 1000 ).
```

```
act( collier, move, M, dummy ) :-
    at( collier, N ),
    clockwise( N, M ).
```

the third rule, taken logically, states that "For all N, if the collier is at N, his act must be a move to the next clockwise square M". This is inconsistent with the preceding rules which say that sometimes the act must be to buy or sell.

So why does this work? The reason is that, as mentioned at the end of Lesson 5, Prolog works through the clauses from the top down, and chooses the first one that succeeds. It may later *backtrack* and look for another solution, as it did with `loves(X,Y)` in

```
loves(X,Y), Y \= mary.
```

If this were to happen in either of the knowledge bases above, it would indeed produce two mutually inconsistent solutions: a fitness that's both good and poor, or an act that's both a move and a buy (or sell).

This doesn't matter in Traveller, because I have written the program surrounding `act` so that it swallows the first solution `act` yields up, and does not backtrack and ask for any others. Hence you can get away with mutually inconsistent rules, as long as you remember that they are tested from the top down, the first solution coming from the first clause to succeed. I have written all the examples used with the Tutor so that this is true.

However, you should be aware that this is a departure from logic, and that we're using Prolog's execution order to resolve inconsistencies. This will become obvious if you load COLLIER, move the third clause to the top, and then call `run`.

There are various ways around the problem of inconsistencies. The purest approach is always to write conditions in full. Were we to do so with COLLIER, the third rule would become

```
act( collier, move, M, dummy ) :-
    not((
            at_seller( collier, coal, Price ),
            carries( collier, coal, 0 )
    )),
    not((
            at_buyer( collier, coal, Price ),
            carries( collier, coal, 1000 )
    )),
    at( collier, N ),
    clockwise( N, M ).
```

Here I have used `not` (mentioned briefly in the animals classifier of Supplement 4) to negate the first two conditions. This approach is logically good because the third rule now carries with it all the information it needs, and no longer has to rely on its position relative to the others. It makes the program easier to read, because a reader can understand the third rule in isolation, without having to examine the preceding ones.

Unfortunately, writing all conditions in full can result in very long-winded programs, and in inefficiencies. This is because Prolog has to test some conditions more than once. An alternative approach is to use a special symbol called *cut* and written as an exclamation mark:

```
act( collier, buy, coal, 20 ) :-
    at_seller( collier, coal, Price ),
    carries( collier, coal, 0 ),
    !.

act( collier, sell, coal, 20 ) :-
    at_buyer( collier, coal, Price ),
    carries( collier, coal, 1000 ),
    !.

act( collier, move, M, dummy ) :-
    at( collier, N ),
    clockwise( N, M ).
```

Textbooks often make a great fuss about the cut, trying to give the impression that it is rather more complex than, say, quantum mechanics, or the justification for monetarism (the first is false; the second may well be true). In this context though, there's no mystery. The cut tells Prolog "when you have got here, do not consider any alternative solutions". So the first cut is saying, in effect, "if buy was one solution, sell and move are completely ruled out".

You can test cut's effect by setting up `collier`'s load and position (using `at` and `carries`) so that he is at a retailer with a load to sell. Then ask the question

```
act( collier, Act, Arg1, Arg2 ).
```

and reply `y` every time Prolog asks whether you want another solution. Without the cuts, you will get two solutions, a `sell` and a `move`. With them, only one.

This is one of cut's two uses; in the other, it's inserted between goals to stop the earlier one backtracking. I shall say no more about cut until the final lesson. In the meantime, I may write rules that look inconsistent, but I will rely on any predicates that call them never to ask for more than one solution. You may do the same ... but you should know when you are doing it.

When I return to the cut, I shall present it as a kind of internal "if", which can be used in writing predicates that perform one action or another depending on the result of a condition. This will be a fiction, but a benign one: as long as you follow the programming style I recommend, no problems will arise.