Supplement 2

Jocelyn Paine

June 4, 1996

1 Introduction

These exercises start off with a mini-world of connected caves inhabited by various creatures, rather like a computer Adventure game.

If an exercise asks you to define and test a predicate, please save the definition in a file for me to look at. Do ask for help if you have trouble. Don't be afraid to ask; programming is an execllent way of demonstrating that you have to make mistakes in order to learn.

2 An Adventure world (PP,PE)

The file `dd.pl` contains the following facts, which define the world of this game. These describe 11 caves called a to k, some of which are connected. I write `b meets a` to mean that there is a passage directly connecting b and a; so you can get from b to a, and also from a to b.

Please draw a map of all the caves and their connections. You can view the file by doing `showlib dd.pl` from Prolog.

2.1 The Wumpus

It is possible to write an entire Adventure game in Prolog. An important part of such a game is the map, which we now know how to write. In a real game, we would also need to describe the caves or rooms in more detail, saying for instance that room a was "a long low room half-a-meter long full of mud" and that room b contained "the magic infra-red radio-therapeutic healing portal". Prolog makes it beautifully easy to do this, and you might consider it as your project: building a complete game from the skeletal version presented here.

To provide that essential element of fear, most Adventure games contain, as well as the player, a number of adversaries simulated by the game program. A game called Wumpus, once popular at Cambridge, involved a world of caves and pits inhabited by you (the player) and a Wumpus, who would eat you if you got into the same cave. As well as Wumpuses, we can borrow from Tolkien (why not? everyone else does) and populate our world with dragons, orcs, narcs and other menaces.

One thing we must do is to keep track of everyone's location. This is simple; in DD, you will find the facts

```
is_at( dragon, b ).
is_at( fred, a ).
is_at( wumpus, c ).
```

One difference between these facts and the ones making up the map is that in a real game, the map stays the same, but the contents of the room can change. In a few turns' time, the dragon, player and Wumpus might have moved:

```
is_at( dragon, f ).
is_at( fred, k ).
is_at( wumpus, a ).
```

It is perfectly possible to write programs that change facts in this way. You don't yet know how: so far, the only way a fact can be changed is if you replace or delete it. But such changes can be made under program control, and you will eventually find out how.

In the game of Wumpus, the Wumpus was simulated by the computer, which therefore needed some representation of how it would behave towards the other inhabitants. In Prolog, this is easy to state using rules. Let's assume that the player, who we have called `fred`, will be eaten if he's in the same cave as the Wumpus. Can you translate that statement into Prolog? That is, can you concoct a fact that begins something like

```
will_be_eaten( fred ) :-
```

and continues with a translation of "fred is in the same cave as the Wumpus"?

This on its own might not seem very exciting. However, combining it with techniques from later lessons, we could write a Wumpus-simulator that would repeatedly move the Wumpus from one cave to another, calling `fred will_be eaten` after each move. If this call ever succeeded, the simulator could output a suitable message, display a few flowers and a tombstone, and then quit the game.

In outline, such a program would be something like this:

```
To run the game:
    Do one game step.
    Run the game.

To do one game step:
    A is the player's next action.
    Perform A, and update the world accordingly.

To perform A, and update the world accordingly:
    If A is 'move to X'
    Then
```

```
    Remove the old fact about the player's location.
    Replace it by one saying he's at X.
    Check the dangers.

To check the dangers:
    If the player will be eaten
    Then
    Say so, and quit.

To check the dangers:
    If the player will be burnt alive (see next section)
    Then
    Say so, and quit.
```

In this outline sketch, the predicates concerned with writing out messages, removing and replacing facts, and quitting, are called "command predicates". They don't make inferences, but actually change things. The other predicates — getting the player's next action, and checking whether he'll be eaten or burnt alive — do make inferences, and you can write them with what you now know.

Most Prolog programs follow this kind of pattern. On the outside sit anumber of "command predicates" which perform various actions. Which action they select is decided by the result of inference-making predicates, such as `will_be_eaten`.

2.2 The dragon

Now, I said nothing about the dragon in the last exercise. Let's suppose that it's more deadly than the wumpus, because it can shoot flame from one cave to another — but it can't shoot further than that. Can you translate into Prolog, "the player (`fred`) will be burnt alive if he is in a cave that is next to the one the dragon is in". Start with

```
will_be_burnt_alive( fred ) :-
```

Test your answer by moving the player and dragon to various caves, and asking

```
will_be_burnt_alive( fred ).
```

You may have a problem to do with unidirectionality (I won't say more than that because it may let the game out of the bag). When you think you have tested your new fact adequately, and you are either satisfied, or you have got something wrong and can't see how to continue, please go to the next section. But try this one first.

2.3 Prolog is not logic

The problem you may have met is that you had written a fact such as this:

```
will_be_burnt_alive( fred ) :-
    is_at( fred, P ),
    is_at( dragon, D ),
    meets( P, D ).
```

Now, suppose you also had

```
is_at( fred, b ).
is_at( dragon, a ).
```

then the question

```
will_be_burnt_alive( fred ).
```

would give the answer "Yes".

But if you had had

```
is_at( fred, a ).
is_at( dragon, b ).
```

then the same question would have been answered "No", despite the fact that the two caves are connected. This is because Prolog doesn't know that `meets` is two-way: as explained at the end of Lesson 1, it doesn't know that, if A meets B, then B necessarily meets A.

To get round this, you might think of adding the fact

```
meets( A, B ) :- meets( B, A ).
```

This is certainly correct, logically speaking. Unfortunately, Prolog is not a complete implementation of logic, and you can't expect that, just because a program makes logical sense, Prolog will necessarily be able to run it. In fact, as some of you have already found, adding a fact like this is likely to get Prolog into an infinite loop. If you wonder why this should be, the stuff on execution order in a later lesson will give you a clue. Should you now be worrying that you now can't rely on Prolog at all, relax! Most of the exercises I set will run quite happily, and when I think they might not, I'll give you ample warning.

The standard way round the difficulty above is to define a new predicate called `joins`, or some other suitable name:

```
joins( Cave1, Cave2 ) :- meets( Cave1, Cave2 ).
joins( Cave1, Cave2 ) :- meets( Cave2, Cave1 ).
```

Then, whenever you want to talk about one cave joining another, use `joins` and not `meets`:

```
will_be_burnt_alive( fred ) :-
    is_at( fred, P ),
    is_at( dragon, D ),
    joins( P, D ).
```

Always using `joins` in your other definitions will get rid of the problem, and provide you with a safely symmetrical version of `meets`. If you hadn't done something like that already, try it now.

Incidentally, don't be thrown by my use of multi-letter variables `Cave1` and `Cave2` in the two facts for `joins`. Variables can have any number of letters, as long as they start with an upper-case one. I just wanted something more meaningful than `X` and `Y`.

2.4 More connection predicates

Now define a predicate that begins

```
is_two_from( Cave1, Cave2 ) :-
```

and which is true if there is one cave between Cave1 and Cave2, thus:

```
-------     ------     -------
!Cave1!-----!    !-----!Cave2!
-------     ------     -------
```

Does your definition allow a cave to be two from itself? Why? What needs to be done to stop it happening?

Now, do the same for

```
is_three_from( Cave1, Cave2 ) :-
```

There are two ways of doing this. One uses `is_two_from` after the if; one does not. Try them both.

Finally, define a predicate

```
is_near( Cave1, Cave2 ) :-
```

which is true if the caves are adjacent, *or* two apart, *or* three apart. You will certainly need more than one clause for this.

2.5 Recursion

This exercise is more difficult, but worth trying. Can you write a predicate

```
on_route_to( Cave1, Cave2 ) :-
```

which is true if Cave1 is connected to Cave2 by a path of any length. (If you look at your map, you will see that not all the caves are connected: i and j are joined to each other, but to no other caves. k is joined to no others at all.)

When trying this, some people say "there are only eleven caves, so I only need a finite number of facts: I'll just enumerate all possible connections". That is not what I want. I want a general rule that will work no matter what the map.

This is harder, and I don't expect everyone will manage it; but if you were able to do the `is_ancestor_of` example in Lesson 2, you should be able to cope with this, and then you'll be well on the way to understanding recursion. There was another example of recursion in the `flows_to` predicate of Supplement 1. In the Traveller trading game of Supplement 4, you will see a use for such a predicate.

3 Representing information spatially (AI)

The following exercises demonstrate that different kinds of information can be represented in the same way, and that methods devised for handling one can be transferred to the others. In particular, they emphasise the way in which methods of representing "spatial" things like maps and networks can be used for more abstract kinds of data. This is worth knowing, because we are very good at visualising things laid out in space, and it will help greatly if we can learn to visualise other problems in the same way.

The caves world uses facts like

```
is_a( a, cave ).    is_a( b, cave ).
```

to identify caves, and

```
meets( a, b ).      meets( b, c ).
```

to describe their connections.

Here are three more sets of facts, each representing a different kind of knowledge. Compare them with the clauses for `is_a` and `meets`.

1

```
is_a( budget_needs_more_money    ,  event ).
is_a( exports_decrease           ,  event ).
is_a( production_decreases       ,  event ).
is_a( workers_incentives_decrease,  event ).
is_a( real_wages_decrease        ,  event ).
is_a( tax_increases              ,  event ).
is_a( exports_decrease           ,  event ).

causes( budget_needs_more_money    , tax_increases ).
causes( budget_needs_more_money    , real_wages_decrease ).
causes( tax_increases              , workers_incentives_decrease ).
causes( real_wages_decrease        , workers_incentives_decrease ).
causes( workers_incentives_decrease, production_decreases ).
causes( production_decreases       , exports_decrease ).
causes( exports_decreasecauses     , budget_needs_more_money ).
```

2

```
is_a( composite        ,  class ).
is_a( dicotelydon      ,  class ).
is_a( flowering_plant  ,  class ).
is_a( plant            ,  class ).
is_a( life             ,  class ).
is_a( animal           ,  class ).
```

```
is_a( vertebrates       ,  class ).
is_a( fish              ,  class ).
is_a( bony_fish         ,  class ).
is_a( daisy             ,  class ).
is_a( crucifer          ,  class ).
is_a( monocotelydon     ,  class ).
is_a( mammal            ,  class ).
is_a( cartilaginous_fish,  class ).
is_a( shark             ,  class ).

includes( bony_fish         ,  shark ).
includes( fish              ,  bony_fish ).
includes( fish              ,  cartilaginous_fish ).
includes( cartilaginous_fish,  herring ).
includes( vertebrates       ,  fish ).
includes( vertebrates       ,  mammal ).
includes( animal            ,  vertebrates ).
includes( living_thing      ,  animal ).
includes( living_thing      ,  plant ).
includes( plant             ,  flowering_plant ).
includes( flowering_plant   ,  dicotelydon ).
includes( flowering_plant   ,  monocotelydon ).
includes( dicotelydon       ,  composite ).
includes( dicotelydon       ,  crucifer ).
includes( composite         ,  daisy ).
includes( composite         ,  dandelion ).
includes( crucifer          ,  wallflower ).
```

3

```
is_a(  bbbbxbbbb,  position ).
is_a(  obbbxbbbb,  position ).
is_a(  bobbxbbbb,  position ).
is_a(  xobbxbbbb,  position ).
is_a(  oxbbxbbbb,  position ).
is_a(  xobbxbbbx,  position ).
is_a(  xobbxbobb,  position ).

o_moves_to( bbbbxbbbb, obbbxbbbb ).
o_moves_to( bbbbxbbbb, bobbxbbbb ).
x_moves_to( bobbxbbbb, xobbxbbbb ).
x_moves_to( obbbxbbbb, oxbbxbbbb ).
o_moves_to( xobbxbbbb, xobbxbbob ).
x_moves_to( xobbxbbob, xobbxbbbx ).
```

The first example describes a small portion of what economists call macro-economics: the relations between inflation, unemployment, etc. The relations are not at all esoteric, in fact they are taken from *There's a Hole in My Budget* by Flanders and Swann, in which there is a "dialogue between the Prime Minister and Chancellor, who wander round the room in a slow inflationary spiral". The same form could be used to describe other economies, as well as numerous non-economic phenomena (I have such a translation of *The Gasman Cometh* if anyone wants it.)

The second example classifies living things.

The third example shows a subset of the possible moves and board states in the game of noughts and crosses — note that if extended to be complete, it would show all possible moves, not just those in one particular game. Board states are represented as names made up of the latters `b` for an empty square, `x` for an X, and `o` for an O. This representation is not very wieldy — later, you will learn better ways to represent such things. In fact, *lists*, mentioned in Supplement 6 in connection with route-finding, would be ideal for this purpose.

3.1 Parents, siblings, and routes

For the first exercise, can you refer to the `on_route_to` fact which you were asked to write in the exercises on caves. You may have found it too difficult to write, but you can appreciate what it means. So what would its analogue mean in each of the three examples above?

Secondly, refer back to Lesson 2. In the three examples above, what are the analogues of (a) X being Y's parent; (b) X and Y being siblings; (b) X being Y's ancestor?

3.2 Representing a neural network

Now consider a neural network. You can either think about real biological neurons in the body, or about an artificial neural network. What predicates might you choose to describe such a network? Sketch part of it, and write out a few facts describing your sketch.

Here are some suggestions that might help: Each neuron has one or more inputs from other neurons and one or more outputs. The inputs may be excitatory, or inhibitory. Each neuron has a current activation level. And so on.

Incidentally, Prolog is not a good language for running connectionist networks — its arithmetic is too slow. However, it is an excellent language for describing their structure. Charles Benett, one of our past students, wrote a program called Neuroguide, which contained part of a neuroanatomy atlas, and allowed you to interrogate it and follow around the connections.

3.3 Your own project?

Think of any other subject where knowledge might be represented in the same way — perhaps one you want to work on for your project. What predicates would be useful?

3.4 Types of space

The four descriptions above are analogous to the caves one. The caves world and the neural net represent *physical* space: by analogy, the other three descriptions can be said to represent an *abstract* space, which contains *points* (the events, classes, or board states), and their *connections* (the causal links, taxonomic connection, or moves). When used in applications like game-playing, or other problem-solving, the space is often called a *state space*, because each point marks a different state of the problem.

If the caves world were part of this universe, each connection would have a length. We don't represent these lengths in our exercises, because all we are concerned with is the structure of the network. Using mathematical terminology, we can say that we are only interested in the *topological properties* — those to which distance is irrelevent — and not the *metric properties* — those concerned with distance. Like the people who make London Underground maps, we ignore distance.

In some types of space, we don't just ignore distance; it has no meaning in the first place. You can see that this is true of the classification example, for instance; it makes no sense to say that one of the `includes` is longer than another.

```
includes( dicotelydon, crucifer ).
includes( composite, daisy ).
```

So watch out when you use Prolog to represent such structures in the way that I have done. If your predicates don't say anything about distance, is it because it has no meaning, or because you have ignored it? The same can be asked of other properties, of course. If your program doesn't mention some property of a situation, is it because the property doesn't exist, or because you have ignored it?

The caves world and these four examples differ in other ways. The caves world and economic model are potentially *cyclic* or looped, because one point can be linked through a chain of paths or causes back to itself. The classification and noughts-and-crosses are *acyclic* or unlooped. We can also say that they are *tree-structured*, because there is a natural way to draw them as a tree. When writing Prolog programs, watch out for cyclic spaces: you will need to use special techniques to write route-finders and other predicates, otherwise they may get into an infinite loop.

In the economic model, classification, neural net, and noughts-and-crosses, the links are *directed*: one end is somehow different from the other. To traverse a causal link backwards is not the same as following it forwards. The caves world is not directed. If you are writing a predicate that describes a directed space, then take care that you get its arguments the right way round:

```
includes( dicotelydon, crucifer ).
```

does not mean the same as

```
includes( crucifer, dicotelydon ).
```

Finally, in the classification and noughts-and-crosses, the points themselves can be *ordered*: it is natural to say of one class of animals that it includes another; it is natural to say of one state of the noughts-and-crosses game that it is later in the game than another. There is no analogue of this with the other examples. In general, spaces which are both acyclic and directed can be ordered in this way. Intuitively, there is one overriding direction, and once you've followed it, you can't get back.

You will not be expected to remember these terms in the rest of the course. The point of this section, apart from trying to make you visualise abstract problems, and to show how they can be represented by Prolog, was to give a taste of the considerations that A.I. people face when representing knowledge. A rich vocabulary has been developed for describing information structures; the examples above are but a small part of it.