Supplement 4

Jocelyn Paine

June 4, 1996

1 Introduction

Because the best way to learn a language is to immerse yourself in it, this supplement is larger than the others, and shows Prolog at work in a number of different situations. The final section introduces a trading game, in which the aim is to program an automatic trader to buy and sell so as to make the greatest possible profit. This is open-ended; the game itself can be extended in many ways, while building truly adaptable traders involves a number of principles from A.I.

As you work through this, please bear in mind the logical meaning of each Prolog clause you encounter. If you have any difficulty understanding it, try applying the translation rules given in Lesson 4.

2 Classifying animals (PP)

This section introduces a simple animal classifier. Please start by loading the `animals` knowledge base. Make a printed copy of it if you want, too. You can look at it by doing `showlib animals.pl` from Prolog, and print it by `ctcprint` from VED.

The classifier is similar to the ski advisor of Supplement 1, but unlike the latter, it contains some *unary* (one-argument) predicates, such as `gives_milk` and `flies`. There are none with more than two arguments, but you will see some of these later, in the trading game.

A complete list of predicates follows. After each, I have stated what I intend it to mean. For example, from the description of `spots` below, you can see that the fact

```
spots( pip, black ).
```

means Pip has black spots.

- `chews_cud(A)`: A is an animal which chews its cud.
- `colour(A,C)`: A is an animal whose colour is C (`black`, `black_and_white`, or `tawny`).
- `eats(A,F)`: A is an animal that eats food type F (`grass` or `meat`).

- `eye_direction(A,D)`: A is an animal whose eyes point in direction D (`forward` or `sideways`).
- `flies(A)`: A is an animal that can fly.
- `gives_milk(A)`: A is an animal that gives milk.
- `has(A,P)`: Animal A has body part P (`claws`, `feathers`, `hair`, `hoofs`, or `pointed_teeth`).
- `is_a(C1,C2)`: Class C1 of animals is included in class C2. The names of classes (including the animals themselves) are `albatross`, `animal`, `bird`, `carnivore`, `cheetah`, `giraffe`, `mammal`, `ostrich`, `penguin`, `tiger`, `ungulate` and `zebra`.
- `lays_eggs(A)`: Animal A lays eggs.
- `long_legs(A)`: Animal A has long legs.
- `long_neck(A)`: Animal A has a long neck.
- `spots(A,C)`: Animal A has spots of colour C.
- `stripes(A,C)`: Animal A has stripes of colour C.
- `swims(A)`: Animal A can swim.

Notice that I have taken care to indicate what type of thing each argument refers to, and what the names used for individuals of each type are. For instance I have listed the body parts used in the second argument of `has`. If you write programs for yourself, it is important to make sure that you know what each predicate does, how many arguments it has, what can go in or come out, and which order the arguments are in. It is all too easy to think that you should write `gives(A,milk)` when elsewhere you have `gives_milk(A)`; that you should write `colour(A,dark)` when elsewhere you have `color(A,black)`; that you should write `lays(eggs,A)` when elsewhere you have `lays(A,eggs)`. When writing down such information, you need not be as formal as I have, but you should note it down somehow. If you have programmed before, you will know that most computer languages let you put such notes as comments in your program.

Care is especially necessary because Prolog does not detect spelling errors. Usually they just make it say "No". For instance, suppose you have the fact

```
colour( pip, black ).
```

and you ask

```
colour( Who, dark ).
```

Prolog treats `dark` as completely different from `black`, and will say `no`, when what you want is probably `Yes - Who=pip`. The same would happen if you were to ask

```
colour( dark, Who ).
black_colour( Who ).
```

2.1 Running the program

Another distinction I made was between "observable" and "non-observable" data. Given an expert system for diagnosing blood infections, it would be reasonable for it to ask for the patient's age and sex, and for the location of major wounds. It would be less reasonable for it to ask for the genus and species of the organisms present in the blood: this can not be observed directly, and is what the system is supposed to be inferring. In the animal classifier, I intended the observable data to be everything about an animal except its classification, which `is_a` is supposed to find for you. So to find out what an animal is, first give it an arbitrary name, e.g. `pip`. Then add facts describing what you can observe about it:

```
gives_milk(pip).
```

and so on. Then ask

```
is_a( pip, X ).
```

Try this by inputting three facts to tell Prolog that Pip is (a) feathered; (b) black-and-white; (c) able to swim. Ask what kind of animal he is. Then try the same for a few other animals. This resembles what you did with the skiing advisor.

2.2 Names of things

In Lesson 4, I said that atoms can be used to name individuals, to name classes, and to name properties. Make an alphabetical list of the atoms in ANIMALS (including the name `pip`), and say how each is used.

2.3 Adding new classification rules

Now have a go at extending the classifier. Start by translating the following sentences into Prolog. If you need to talk about concepts, such as flying ability, that the example already uses, then use the same names. Otherwise, make up sensible names of your own, and ensure they're consistent throughout. Note: when translating some sentences, you may require more than one rule.

To illustrate something important about logic, I have chosen two sentences that, strictly speaking, are of no help as they stand. Which are they, and why?

- A horned ungulate is an antelope.
- Bats are the only flying mammal.
- Molluscs have shells.
- Something with gills, fins, or scales is a fish.
- Rodents have two large front teeth and side-pointing eyes.

Notice incidentally the different ways English has of expressing how the properties of an object imply its classification.

2.4 Notes

You will see that two of the original rules contain the test

```
not( flies(A) )
```

The `not` predicate is unique amongst those you have met so far because it takes a *goal* as argument. Suppose you have the fact

```
flies( pip ).
```

and you ask the question

```
not( flies(pip) ).
```

You will get the answer `no`. If you then delete `flies(pip)` and ask the same question, you'll get the answer `yes`. So `not` inverts the truth of its argument.

In some circumstances, `not` can be tricky to use. You can get an example of this by asking

```
not( flies(P) ).
```

and observing that it does not give you the names of all the non-flying animals. More will be said about this in Lesson 6; in brief, `not` does funny things if its argument contains variables which don't have values when `not` is called. This doesn't happen in the classifier, because I've written it so that all such variables will get values before `not` is called.

Finally, please note again that Prolog is *not* an expert system. It lacks features that an expert system should have, such as an easy means of entering observable data, and an explanation in English of its conclusions. However, for the purpose of this course, that does not matter, and you can use Prolog to simulate the kind of reasoning a real expert system would do.

3 Prolog versus English (PP)

The knowledge in an expert system can be expressed in Prolog's style of logic. But what about other kinds of knowledge? For example, one on-going area of research is into natural-language understanding: there are many programs that can read text or listen to speech, and extract some kind of meaning from them. A program called Frump was written at Yale by Roger Schank to make summaries of news stories. It was directly connected to the teletype output from news agencies such as Reuters and scanned the text coming along the wire for key words and phrases, compared them with what it knew from previous stories, and output a summary of anything new to a printer. Such programs need some way to represent the meaning of their input, and some form of logic is often first choice. I hope this exercise convinces you that this is feasible.

In fact, almost all schemes used for representing knowledge in A.I., with the possible exception of neural nets and genetic algorithms, can be translated into the standard kind of formal logic (known as *first-order predicate calculus*). And although Prolog is more restricted in where it allows quantifiers than formal logic, it is still possible, by following certain rules, to translate any statement of formal logic into Prolog.

If you don't like logic, you might now be saying "if I want to study A.I., I should try to find a less logical language than Prolog". That will not help for long. Languages such as Pascal, Lisp, and Basic do at first sight seem to require less proficiency in logic. This is because they are lower level than Prolog, and can't accept logical statements directly. When you're trying to represent knowledge in these languages, you first have to decide on its logical meaning, and *then* translate into a notation that's lower level and even less clear. Prolog at least avoids some of this work.

You will also find logic used as a universal language in the A.I. literature. Many papers use it to specify what complex programs do, or to capture patterns of human cognition. And an increasing number of non-Prolog A.I. languages are based on logic. That's certainly true of the languages in which most expert systems are programmed, including the eminently commercial ones used for share dealing by big City companies. So a knowledge of logic may be useful for your financial health ...

3.1 Proverbs and other sentences

Please start by translating the following sentences into colloquial English. In this and future exercises, be careful with the order of arguments in predicates. I have adopted the convention that, where a predicate has two arguments: `p(X,Y)`, then it would make sense if written in English `X p Y`.

If you look at just the names, you may think you know what the translation is going to be: several of the facts clearly resemble familiar proverbs. However, I have deliberately mangled some of the Prolog, so if you just rely on surface features, you may be mislead. Also, some are not legal Prolog. Which?

- ```
  admires(P,P) :- is_a(P,narcissist).
  ```
- ```
  hesitates(X) :- lost(X).
  ```
- ```
  colour( X, purple ) :- is_a( X, mushroom ), is_poisonous( X ).
  is_poisonous( X ) :- is_a( X, mushroom ), colour( X, purple ).
  colour( X, purple ), is_a( X, mushroom ).
  colour( X, purple ), is_a( X, mushroom ).
  ```
- ```
  is_good(X,architect) :- has_built(X,Y), stays_up(Y).
  built_well(X,Y) :- has_built(X,Y), stays_up(Y).
  ```
- ```
  is_good(X) :-
      is_a(X,pilot),
      has_takeoffs_so_far(X,N),
      has_landings_so_far(X,N).
  ```
  ```

- ```
  suitable_for( X, Y ) :-
      is_a( Y, goose ), is_a( Z, goose ),
      is_male( Y ), is_female( Z ),
      suitable_for( X, Z ).
  ```

- ```
  better_than( X, Y ) :-
      is_act_of( X, giving ), is_act_of( Y, receiving ).
  ```

- ```
  is_silly_fish(F) :-
      is_a(B,bait),
      was_caught_at_time_with(F,T1,B),
      was_caught_at_time_with(F,T2,B),
      T1 \= T2.
  ```

  The predicate `was_caught_at_time_with(F,T,B)` means that "F was caught at time T using bait B". The operator `\=` means not equal, and is here being used to compare times.

Finally: is `loves(mary,everyone)` a good translation of "Mary loves everyone"? Is `not_lonely(X) :- knows(X,someone)` a good translation of "X is not lonely if X knows somebody"? Why or why not?

3.2 English to Prolog

Along the same lines, have a go at translating the following statements into Prolog. This is harder; you will need some ingenuity in thinking up suitable predicates.

- Everything can be expressed in predicate calculus, even irony.
- Using the predicates `unlike(Thing1,Thing2)` and `is_a_business(Thing)`, translate "There's no business like show-business".
- There is no proverb which is not true.
- The Dutch proverb "Geen koren zonder kaf" translates as "No corn without chaff". Assuming this means "No piece of corn exists without a piece of chaff in the same place", translate into Prolog, using predicates `is_in(Thing,Place)` and `is_a(Thing,ThingsType)`.

3.3 Abstract individuals

You are now used to treating everyday objects like animals and people as individuals. A.I. researchers have found, when writing natural language processing programs, that they often need to do the same with more abstract things, such as acts, periods of time, and speeds. It's interesting to see how you can do this in Prolog.

For example, in Lesson 3, I justified Prolog by saying that it makes it easy to express sentences like "Mary gave Fred a calculator". You could write this as
  ```

```
gives( mary, fred, calculator ).
```

But suppose we wanted to express the following:

> On the day before Christmas, Mary eagerly gave a diamond-studded Faberge calculator to Fred for his Christmas present as they languidly downed foaming pints of nut-brown ale, basking in the luxuriantly sumptuous surroundings of Oxford's Welsh Pony hotel.

This sentence relates, via the act of giving, at least five pieces of data: a time, a giver, the given, the recipient, and the place. In addition, there are some other bits of information about the character of the act (eager), about what else was going on at the same time, and so on. In principle, you could write an enormous predicate with 10 or 20 arguments, allocating argument number 5 to be the place, 12 to be the character of the act, and so on. But this would get terribly unwieldy.

As another example, consider expressing "Paul gave Elspeth a book" and "Robb gave Carole a book". This is no problem:

```
gave( paul, elspeth, book ).
gave( robb, carole, book ).
```

But suppose we also want to say that Paul's gift to Elspeth was given before Robb's gift to Carole, and we also want to indicate that the two books were copies of the same work (but not the same individual). It is not possible to add such information to the above facts easily, and most A.I. programmers would not try. Instead, they would use a style of representation in which the acts of giving and the books are themselves named. Here is one way to do so.

```
act( giving, g1 ).                act( giving, g2 ).
actor( g1, paul ).                actor( g2, robb ).
object( g1, b1 ).                 object( g2, b2 ).
recipient( g1, elspeth ).         recipient( g2, carole ).
is_a( b1, book ).                 is_a( b2, book ).
title(b1,'Beer Atlas of Britain').
                                  title(b2,'Beer Atlas of Britain').
printed( b1, 1991 ).              printed( b2, 1989 ).
edition( b1, 9 ).                 edition( b2, 5 ).
before( g2, g1 ).
```

This means "`g1` is an act of giving; its actor is Paul; its object is `b1` (a book); `b1` is printed in 1991" and so on.

Of course, Prolog doesn't care which style you use. There are a number of differing schemes, each with its pro- and opponents, but most resemble in some way the one above. Such schemes usually break down complex ideas into a large number of elementary predicates, in the way I broke down "Paul gave the book to Elspeth" into the predicates `act`, `actor`, `object` and `recipient`.

You will not be expected to use such schemes in the rest of this course, although if you want to do a project that involves generating them from English, we have some bits of program that can help. But it's interesting to see how they work out, and it will give you practice in reading Prolog programs. Below I have expressed a well-known limerick in this way. Can you work out what it means?

```
is_a(l1,lady).
has_age(l1,young).
started_during(t,d1).
slower( c, s ).
follows(d1,d2).
is_a( s, speed ).
follows( n, d2 ).
took_trip(l1,t).
velocity_of( light, c ).
is_a(d2,day).
is_a( c, speed ).
velocity_of( l1, s ).
is_a(d1,day).
is_a(n,night).
ended_during(t,d2).
is_a( t, trip ).
```

4 Navigating through data structures (PE,AI)

Generally, Prolog programs often fall into two parts. One is fixed data — networks, family trees, and so on — described by structural predicates like `meets` and `is_a`; the other is some component which wanders around it, changing facts, generating output, reading input, and so on. You have already had some practice at expressing the fixed data, and in Lessons 7 and 8, you will get an idea of how to do the navigation. To prepare you for this, some examples of what you can do by navigating are shown in the rest of this supplement. While reading them, you may like to bear in mind the following quote:

> A German with starving wife and family sought assistance from the Winter Aid Fund. Entering an imposing building, he found himself facing two doors, one marked "Party Members", the other "Non-Party Members". Going through the second door he was faced with two more, one marked "Subscribers to the Fund", and the other "Non-Subscribers to the Fund". Going through the latter, he was once again faced with two more doors, and so on. After an hour or two, having come to two doors, one marked "Single Men", the other marked "Family Men", he chose the latter and found himself out in the street where he had started. A friend asked him whether he had got any help. "No" he said, "but heavens, what organisation!".

4.1 Science fiction stories

Start by loading knowledge base `spin` with the command `library(spin).` It defines a predicate with no arguments at all, also called `spin`. Since all the predicates you have seen so far have had arguments, you may find it hard to see how argument-less predicates could be any use. After all, the truth value of a predicate depends on its arguments; surely, if it has none, then it must be always true or always false? However, as we shall see in Lesson 7, such predicates can be useful. There is more to Prolog than logic.

Start by calling `spin` a few times, by asking the question

```
spin.
```

Now, have a look at the knowledge base. It contains a number of unconditional facts whose predicate is `element`, with three arguments. How do they describe the stories `spin` generates? Sketch the process by which you think `spin` operates.

Note incidentally that the arguments to `element` are almost all quoted atoms. We need to quote them because they contain spaces, capitals and other funny characters that would not make sense in unquoted atoms.

4.2 Musical rhythms

Now load `notes`. Call `spin` again. Now it generates rhythms to be used as a basis for fugues. In the output, `c` means crotchet; `m` minim, and `q` quaver. This is taken from a previous practical project, namely a fugue-generating program written by Lucy Mackintosh, which first generated a basic two-bar rhythm structure, then superimposed a melody, and finally constructed other harmonically-related melodies for the other voices. The melody generator is rather too complicated for this stage of the course, so what I've shown you is just the rhythm part.

4.3 Prolog and grammars

.

By altering the clauses for `element`, you can describe a variety of other structures, for example menus, conversations, or stages in someone's lifetime: school leads to PPE or psychology or philosophy; PPE leads to merchant banking; psychology leads to a D.Phil and thence to academic fame; philosophy leads to destitution .... You might try this. When you have done so, you can call `spin` again.

This is a very simple demonstration of Prolog in grammar-writing: we are all familiar with grammars for describing sentences, but they can also describe other structures, such as stories or music. In general, you can use a grammar in one of two ways. The first is for analysing structures into their parts. The second, which we do here, is generating structures from those parts.

These kinds of grammar, incidentally, have been made out of some very simple Prolog components, and would not really be suitable for describing the structure of sentences. (Why?) However, Prolog also contains a special feature called DCG rules that allow you to write grammars for natural languages very easily. If you want to try your hand at this, ask later on in the course.

Note that the grammar in SPIN lacks coherence between its elements, as was probably obvious from some of the stories. In the same way, a grammar for English which stated that

```
A sentence is a noun phrase followed by a verb phrase
```

would not as it stands prohibit singular nouns from being followed by plural verbs. Nor would it prohibit sentences like "Melancholy Justice flirted purposefully with 127 purple elephants". To take account of such things, we need some way to specify *agreement*: to say that the form of one element much depend on that of another. You can do this easily with DCG rules.

Incidentally, SPIN is similar to a *transition matrix* which is another method used to describe behaviour and changes of state in a system. In a transition matrix, each element of the grammar is linked to each element which can follow it by a number: the number specifies the probability that one will succeed the other. So instead of, as in `spin`, choosing any branch with equal probability, we can state that some branches are high-probability and others are very low. Here's one way we might encode a transition matrix for behaviour at CTC:

```
state( 1, do_next_exercise ).
state( 2, go_and_drink_coffee ).
state( 3, leave_in_disgust ).
state( 4, leave_happy ).
state( 5, ask_for_help ).
state( 6, apply_help_given ).

prob( 1, 1, 0.2 ).  prob( 1, 2, 0.2 ).  prob( 1, 3, 0.5 ).
prob( 1, 4, 0.05 ). prob( 1, 5, 0.05 ). prob( 1, 6, 0 ).
```

Here, `prob(X,Y,P)` means "there is a probability of P that state Y will follow state X". There is no chance of state 6 following state 1; it can only follow state 5 (you can't apply help you don't have). We could write a modified `spin` that took these probabilities into account, and you might find the idea useful if you want to have a go at story generation or something similar.

.

4.4 Diagnosis

Now load DEPRESSION. This knowledge base defines a no-argument predicate called `run`, for diagnosing the causes of depression in terms of provoking agents and vulnerability factors. Like the fugue-generator, it was written as a previous practical project, this time by Alison Fry.

To use the program, type `run.`. It will pose you a number of questions about your current and past circumstances: to answer them, type `y` or `n` (or, if requested, a number), followed by a RETURN. You do not need a full stop after the reply. When answering, the worse you make things seem, the more factors that will appear in the diagnosis.

Look at the knowledge base again. What structure do the facts define, and how do you think they're used?

4.5 Note

You have now encountered two kinds of diagnostic system. The first, the animal classifier, was written by converting each rule directly into a Prolog clause, then relying on Prolog's question-answering ability to get results.

The second, the depression system, worked at a different level. Instead of translating each rule directly to Prolog, I followed the Winter Aid Fund quote, and built a kind of network. Think in terms of rooms, where each room contains a question. You take a different exit for each possible answer to this question, thus coming to a new room in which there sits either another question or a diagnosis. You may find this way of doing things useful for your project: many students do.

5 The game of Traveller (PP)

5.1 The board

In the previous section, you saw that Prolog can be used to navigate around structures defined by sets of facts. The following program navigates in a more literal sense. Please start by looking at the file entititled BOARD which comes with this supplement. It defines the basic predicates `square`, `building`, `joins`, `in`, `loop`, `sells_fuel`, `buys` and `sells`, plus three ones derived from them: `adjacent`, `clockwise` and `distance`.

Together, these describe a simple street layout, with squares containing shops and fuel stations. Unlike Monopoly, the squares are not confined to the outside of the board, but can form streets, some looped back on themselves, wandering over its interior.

More specifically, the predicates have the meanings given below:

- `square(S,X,Y)`: S is the number of a square, lying at co-ordinate (X, Y). Each square has a unique number; these are integers (whole numbers), and work upwards from 1. Adjacent squares do not necessarily have adjacent numbers.
- `building(S,B)`: B is the name of the building in square number S.
- `joins(S1,S2)`: whereas `square` defines where squares lie and what they contain, `joins` defines how they are connected. There is one clause, `joins(S1,S2)` for every pair of connected squares. To make things simple, squares can only be connected by vertical, horizontal, or 45-degree diagonal lines.

- `in(S,L)`: The board contains several streets that curve back on themselves, forming closed loops. Each loop has a name (you will see why later). There is one clause, `in(S,L)` for every square S that lies in a loop called L. You can think of the loops as being distinct regions of a city, such as the City in London or the central part of Oxford, except that to keep things simple, they do not contain any roads apart from the loop itself.

- `loop(L)`: This defines the names of the loops. There is one clause for each loop.

- `adjacent(S1,S2)`: This is true if S1 joins S2. Note that it is defined in terms of `joins`: S1 is next to S2 if S1 joins S2 or S2 joins S1. The reason for this is something we alluded to in the cave world example of Supplement 2. If Prolog has a fact such as

  ```
  joins(66,67).
  ```

  it does not automatically assume that the converse — `joins(67,66)` — is true. So we have to define a new predicate which holds no matter which way the squares are said to be joined. We could instead have had two sets of facts — both `joins(66,67)` and `joins(67,66)` — but this would waste space.

  The import of this is that if you want to find out about connected squares, you should use `adjacent`, not `joins`.

- `clockwise(S1,S2)`: This is true if square S1 and S2 are both in the same loop, and S2 is immediately next to, and clockwise of, S1. Its purpose is to give you a simple way of finding your way round loops. It is defined in terms of `joins`, and only works because I took care, when generating the board definition, to make sure that the `joins` facts always had their arguments in an order which would make it work.

- `distance(S1,S2,D)`: This sets D to the distance between S1 and S2. It uses `is`, an operator for doing arithmetic that you'll meet in Lesson 6. The formula is the standard one for working out distance in terms of X and Y co-ordinates: `X^2` means X squared, and `sqrt(...)` means the square root of.

If you care to compare the BOARD file with the diagram of the board, it should become clear what these facts say.

5.2 The buildings

In this era of recession — I wrote that nine months ago, and it is still true! — it is important that we all be good consumers and keep the economy moving. The macro-economic knowledge base of Supplement 2 showed what happens when we fail. So, bearing this in mind, the buildings all concern buying and selling, as you may have guessed from their names.

There are three kinds of building: wholesalers, retailers, and fuel-stations. Each retailer sells one (and one only) of five types of good to the public: coal, diamonds, glasses (tableware, not eyewear), peaches, and televisions. The retailers buy their stocks from traders, who in turn buy from wholesalers. By buying cheap at a wholesaler and selling dear to a retailer, a smart trader can make a hefty profit. Wholesalers, retailers, and fuel-stations are described by the facts below:

- `sells(B,Good,Price)`: B names a wholesaler, and will be one of the names given as the first argument of `building`. Good is the type of product sold. Like retailers, each wholesaler deals in only one kind of product. Price is the price at which the wholesaler sells one unit of the product to a trader (i.e. the price at which the trader *buys* it).

- `buys(B,Good,Price)`: B names a retailer. As before, Good is the product type. This time, Price is the price at which the retailer buys from the trader, i.e. the price at which the trader *sells* it.

- `sells_fuel(B,Price)`: This tells you which buildings are fuel stations. Price is the price per unit of fuel, and B is the name of the building.

5.3 Trading

To play Traveller, you have to move around the board, making as much profit as possible. It takes some skill to keep going: for example, you must not allow yourself to run out of fuel. But there is a twist. You yourself are not playing Traveller — or not directly. It is your task to program an automatic trader in Prolog, and to give it rules that determine when and where to buy, sell, or move. Once you start your trader off, it must run without any help from you. The only thing you can do is to watch the output: sympathise if it runs out of fuel or money; applaud if it makes a huge profit.

Before showing you how to program a trader, I need to say some more about prices. Goods are bought and sold in "units". So a trader may tell Prolog that it wants to buy 20 units of coal, or to sell 100 units of diamonds. To make Traveller more realistic, you can think of these units as being sacks (of coal); individual boxes (for diamonds and televisions); boxes of six (glasses); boxes of twenty-four (peaches). However, it doesn't matter how many things are inside; you can only buy in whole units.

The buying and selling prices given by `buys` and `sells` are all quoted as the price per unit.

Products have volumes, measured in cubic feet. These are given by the only fact I haven't yet mentioned: `unit_volume(Good,V)`. V is the volume occupied by one unit of Good (all goods of the same type have the same unit volume — containerisation). This matters because as a trader, you drive a lorry whose size is strictly limited. It has a capacity of 1000 cubic feet. You can always fill it up to this amount, provided you have the money, but you cannot cram in any more above the limit.

When you start off, you are given £5000 in cash, to spend as you will. There is no upper limit on your cash — you can accumulate indefinitely — but you are not allowed to go below zero. There are no credit facilities in this game.

Your lorry has a fuel tank whose capacity is twenty units. When you start off, it is full. At each stage in the game, you are allowed to do one of four things:

- Buy some fuel.
- Buy some stock from a wholesaler.
- Sell some of your stock to a retailer.
- Move one unit along any road.

Thus, when you move, you can only move to an adjacent square, and then only if it is connected to your current square. Each move costs one unit of fuel, so after twenty moves you will be out of the game unless you remember to refuel.

To buy or sell, you must be on the same square as the person you're dealing with. You can only buy (fuel or tradeable goods) if you have enough money; otherwise you will be thrown out of the game for fraud. If you try to buy more fuel than you have room for in your tank, you will certainly spend the money; your tank will be filled, and any excess will just be spilt. However, if you try to buy more of a good than you have room for, the transaction will be forbidden for safety reasons, and you will be kicked out as an unsafe driver.

You can only sell a good if you have enough of it in your lorry. If you haven't, you will again be thrown out for fraud.

5.4 Running a trader

Now, please do the following. Start up Prolog, then do

```
library(traveller).
```

This will load the software which implements the game. Now do

```
library(trader).
```

This loads code which defines a particular kind of trader. Finally, type the question

```
run( trader, 67 ).
```

The game will start by giving the display below. The fuel and tank size are written out in units; each trader begins with a full tank. Initial cash is £5000, and initial loads are all zero. The total load is displayed in cubic feet, and the stock of each good is displayed both in units and in cubic feet, using `unit_volume` to provide conversion factors.

```
trader is on square 67.
  Fuel 20 in tank size 20.
  Cash 5000.
  Total load 0 cu ft in lorry size 1000.
    Stock of televisions = 0 units (0 cu ft).
    Stock of peaches = 0 units (0 cu ft).
    Stock of glasses = 0 units (0 cu ft).
    Stock of diamonds = 0 units (0.0 cu ft).
    Stock of coal = 0 units (0 cu ft).
```

As the game continues, you will see a summary of the actions the trader takes at each turn, and the result on his fuel, cash, position, and stocks. Wait until the trader makes his way to The Hub, and has gone through a few cycles of buying and selling; when you have seen enough, you can interrupt the game.

Now, how can you write a trader of your own? Well, when you call `run`, it starts by asserting some clauses which describe the trader's state. These are:

- `at(T,Square)`: T is the name of the trader, and Square is the number of the square he is on. This always starts off as the second argument to `run`.
- `carries(T,Good,Qty)`: T is the name of the trader, Good is the name of one of his goods (`coal`, `diamonds`, `glasses`, `peaches` or `televisions`) and Qty is the number of units he has.
- `cash(T,Cash)`: T is the name of the trader, and Cash is the amount of cash he has.
- `fuel(T,Qty)`: T is the name of the trader, and Qty is the number of units of fuel he has.
- `max_load(T,Vol)`: T is the name of the trader, and Vol is the maximum volume his lorry can hold. In this version of Traveller, it is 1000 cubic feet.
- `tank_size(T,Qty)`: T is the name of the trader, and Qty is the maximum number of units he can get into his tank. In this version of Traveller, it's 20.
- `total_load(T,Vol)`: T is the name of the trader, and Vol is the total load in cubic feet.

The name T of the trader is the first argument to `run`. So for `run(trader,67)`, you get clauses

```
at(trader,67).
cash(trader,5000).
fuel(trader,20).
carries(trader,coal,0).
carries(trader,diamonds,0).
```

```
carries(trader,glasses,0).
carries(trader,peaches,0).
carries(trader,televisions,0).
max_load(trader,1000).
tank_size(trader,20).
total_load(trader,0).
```

Note that, apart from `carries`, there is only one clause of each kind.

Having done this, `run` then tries to find out what the trader's first action is to be. Will he move to a new square, buy something, or sell something? It does so by calling (i.e. by asking itself the question)

```
act( T, Action, Arg1, Arg2 )
```

where T is the name of the trader. The file TRADER contained clauses *I* have written for `act`: to play Traveller, *you* must write some for your trader.

When `act` is called, it must set Action to one of `move`, `buy`, or `sell`. If Action is `move`, Arg1 must become the next square to move to, and Arg2 must become the atom `dummy`. If Action is `buy` or `sell`, Arg1 must become the name of a good, and Arg2 must become the quantity in units that is to be bought or sold. Finally, if Action is `buy`, Arg1 can also be `fuel`. In this case, the trader is buying fuel, not a good for resale.

5.5 What facts does `act` need?

You can demonstrate this in the following way. Do

```
library(circler).
```

This loads a simple version of `act`, driving a trader named `circler`. To look at its definition, you can do `showlib circler.pl`. There is one clause, which makes the circler always move clockwise, never buying nor selling. Try

```
run( circler, 18 ).
```

and see what happens.

Now, how can you try `act` for yourself? You need several things. Firstly, the board-description predicates `square`, `building`, `in`, `loop`, `adjacent`, `clockwise`, `distance`, `sells`, `buys` and `sells_fuel` must be in place. Well, this would have happened when you loaded TRAVELLER.

Secondly, you need to fake up some facts which represent a `circler` at some point in its travel. These are the ones defined by `at`, `carries`, `cash`, `fuel`, `max_load`, `tank_size` and `total_load`. These are treated specially. As `run` proceeds, it updates the knowledge base with these predicates. You can add your "faked" ones via VED.

5.6 Testing act

You are now ready to try out `act`. Look at the definition of `act` from CIRCLER. The only trader-state predicate it calls is `at`. This means that, so long as you have a clause for `at`, you can test this version of `act`. So input one — `at(circler,18)`, perhaps — and then type

```
act( circler, Action, Arg1, Arg2 ).
```

You should find that Action becomes `move`, Arg1 is whichever square is clockwise of the current one, and Arg2 is `dummy`.

All this may have seemed rather long-winded, but the gist of it can be stated quickly. To run a trader, you call `run`, passing it the trader's name and initial square. To find out what the trader will do next, `run` calls `act`, passing the trader's name in the first argument, and getting back the name of an action, plus some other information in the other arguments. `act` has two sources of information. The first is the board predicates. The second is the predicates that describe a trader's current state. All these take the trader's name as their first argument. As `run` proceeds, it updates these. If you want to continue a run from where it left off, perhaps after changing some of the state predicates, you can use `go_on`. This takes one argument, the trader's name. Finally, you can of course ask questions that involve `act` directly, if you want to test how it responds to various situations.

5.7 A bit of practice

Can you now define the following predicates:

- `square_sells( Square, Good, Price )`: This holds if Square sells Good at Price.
- `square_buys( Square, Good, Price )`: This holds if Square buys Good at price.
- `at_seller( T, Good, Price )`: This is true if trader T is at a square which sells Good at Price.
- `at_buyer( T, Good, Price )`: This is true if trader T is at a square which buys Good at Price.

Answers will be found in the file COLLIER, but don't look until you have tried the next section.

5.8 Coal trading

With these predicates, can you define a version of `act` that drives a coal-trader round the Colliery Row circle? It will be like `circler`, except that if he stops at a coal merchant, he should buy or sell as appropriate. It is possible to make very complicated rules when trying to squeeze out the maximum possible profit. Don't bother with that yet; you probably don't know enough Prolog anyway. Just have him buy a fixed amount at each wholesaler, if his lorry is empty; and sell a fixed amount to each retailer, if his lorry is not empty. You can ignore the need to buy fuel for the moment.

One possible solution is in the file COLLIER. I have pre-calculated the amount of coal to buy on each cycle. Later, you can try to build a more adaptable trader.