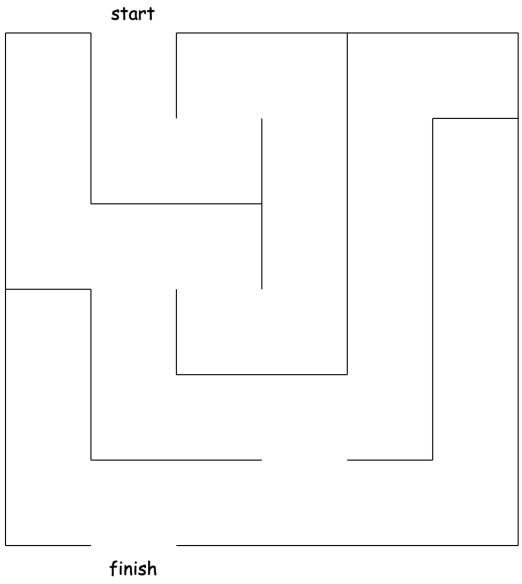
Lesson 6: Example Program - Maze

What's It All About?

1. The problem of finding a path through a maze is considered.
2. A program to more or less randomly search for a solution is written, subject only to the constraint that the maze runner cannot visit a location in the maze more than one time. The program shows that the Prolog inferencing engine can be used to perform search.
3. This lesson serves as a segue to state space problem solving.
4. A word on utility programs/libraries is also provided.

The Maze Running Problem

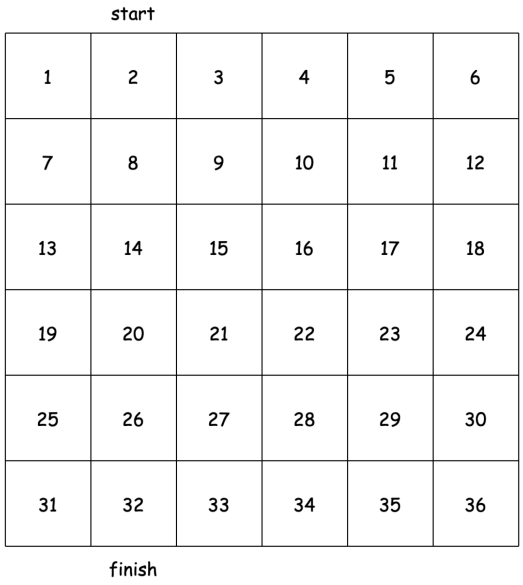
The maze is quite simple, and is pictured below.



The object is quite simply to find a path through the maze from the start to the finish.

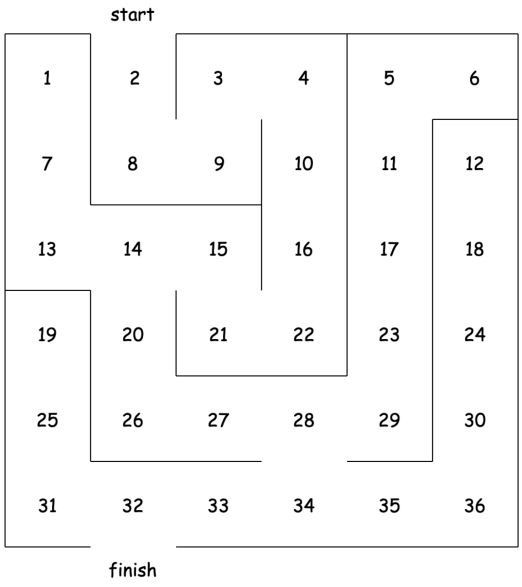
Representing the Maze in Prolog

How might we represent the maze in Prolog? A simple approach would be to imagine the maze is placed on top of a 6x6 grid of cells. Then number the cells in "row major" order with the numbers from 1 to 36, as shown in the following image:



The "start" and "finish" labels can be viewed as special cells, just outside the maze, for entry and exit.

Now, simply imagine laying the maze atop the infrastructure of the grid, and you should see something like the following picture in your mind:



That seems like a nice mental representation, but what we need is a Prolog representation.

For each of the 38 positions, we can move from the position to one or more other positions.

If we can move from one position to another position, we will say that the two positions are **connected to** one another.

We might establish a `connected-to` relation for two locations in a two step process. First, we will enter a single fact for each pair of connected locations:

```
connect(start,2).
connect(1,7).
connect(2,8).
connect(3,4).
connect(3,9).
connect(4,10).
...
connect(31,32).
connect(32,33).
connect(32,finish).
connect(33,34).
connect(34,35).
connect(35,36).
```

Then we will enter a rule to represent the fact that two cells are connected to each other:

```
connected_to(A,B) :- connect(A,B).
connected_to(A,B) :- connect(B,A).
```

At this point, you might want to play around with connectivity of the maze in Prolog for a bit. Perhaps you want to test your code. Perhaps you just want to enjoy a bit of interaction with a program that you wrote.

```
bash-3.2$ swipl
<< redacted bit >>

?- consult('maze.pro').
% maze.pro compiled 0.00 sec, 47 clauses
true.

?- connected_to(start,2).
true

?- connected_to(2,start).
true.

?- connected_to(start,finish).
false.

?- connected_to(16,Other).
Other = 22 ;
Other = 10.

?- connected_to(26,Other).
Other = 27 ;
Other = 20.
```

```
?- connected_to(Other,finish).
Other = 32 ;
false.

?- connected_to(32,Other).
Other = 33 ;
Other = finish ;
Other = 31.

?- halt.
bash-3.2$
```

Finding a Path throught the Maze

We would like to find a path through the maze. Better, expressed, we would like to be able to ask Prolog to find a path through the maze.

How might we represent a path? As a list!

Demo - with some intermediate output

```
bash-3.2$ swipl
<<redacted>>

?- consult('maze.pro').
% maze.pro compiled 0.00 sec, 48 clauses
true.

?- solve.
>>>:  PathSoFar = [start]
|: g.
>>>:  PathSoFar = [2,start]
|: g.
>>>:  PathSoFar = [8,2,start]
|: g.
>>>:  PathSoFar = [9,8,2,start]
|: g.
>>>:  PathSoFar = [3,9,8,2,start]
|: g.
>>>:  PathSoFar = [4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [22,16,10,4,3,9,8,2,start]
|: g.
```

```
>>>:  PathSoFar = [21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [29,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [23,29,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [17,23,29,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [11,17,23,29,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [5,11,17,23,29,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
a|: g.
>>>:  PathSoFar = [6,5,11,17,23,29,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [35,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [36,35,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [30,36,35,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [24,30,36,35,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [18,24,30,36,35,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [12,18,24,30,36,35,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [33,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [32,33,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
>>>:  PathSoFar = [finish,32,33,34,28,27,26,20,14,15,21,22,16,10,4,3,9,8,2,start]
|: g.
start
2
8
9
3
4
10
16
```

```
22
21
15
14
20
26
27
28
34
33
32
finish
true

?-
```

Demo

```
bash-3.2$ swipl
<<redacted>>

?- consult('maze.pro').
% maze.pro compiled 0.00 sec, 1 clauses
true.

?- solve.
start
2
8
9
3
4
10
16
22
21
15
14
20
26
27
28
34
33
32
finish
true

?- halt.
bash-3.2$
```

Code

```
% ---------------------------------------------------------------------
% solve(Start,Solution) :: succeeds if a Solution represents a path from
% the Start=start to finish.

solve :-
  extend_path([start],Solution),
  write_solution(Solution).

extend_path(PathSoFar,Solution) :-
  PathSoFar = [finish|_],
% check('>>>','PathSoFar',PathSoFar),
  Solution = PathSoFar.
extend_path(PathSoFar,Solution) :-
% check('>>>','PathSoFar',PathSoFar),
  PathSoFar = [CurrentSpot|RestOfPath],
  connected_to(CurrentSpot,NewSpot),
  not(member(NewSpot,RestOfPath)),
  extend_path([NewSpot|PathSoFar],Solution).

write_solution([]).
write_solution([H|T]) :-
  write_solution(T),
  write(H),nl.

check(Label,Name,Value) :-
  write(Label),write(':  '),
  write(Name),write(' = '),
  write(Value),nl,
  read(_).
```

On Packaging Utilities

Regardless of the language that you are using, you will find that the packaging of utilities, relatively small but variously useful bits of code, can greatly assist in your work. In Prolog, packaging code for reuse, whether small bits or large domain specific libraries of the stuff, is accomplished quite simply by placing the code in files and then loading the relevant files at appropriate times.

By way of example, consider the following small collection of utilities, placed in a file called `inspectors.pro`, that can be valuable in tracing the execution of a program, and consequently, for purpoeses of debugging.

```
% ---------------------------------------------------------------------
% ---------------------------------------------------------------------
% --- File: inspectors.pro
% --- Line: Utilities for inspecting memory during program execution
% ---------------------------------------------------------------------

% ---------------------------------------------------------------------
% --- These two can be used to print the value of a variable, labelled
```

```
% --- in two ways, and pause for the programmer to check out the
% --- situation. The firstone is generally useful. The second one is
% --- applicable only whenthe value of the veriable is a list, and it
% --- will print the valuein reverse order which is sometimes just
% --- what is desired. The first label generally pertains to a location
% --- in the program. The second label is just the name of the variable
% --- to which the value is bound.

check(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  write(Value),nl,
  read(_).

checkr(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  reverse(Value,RValue),
  write(RValue),nl,
  read(_).

% ------------------------------------------------------------------------
% --- These two are like the previously described checking predicates,
% --- except that they do not do the pause.

trace(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  write(Value),nl.

tracer(Label,Name,Value) :-
  write(Label),
  write(Name),write(' = '),
  reverse(Value,RValue),
  write(RValue),nl.

% ------------------------------------------------------------------------
% --- Like trace, but without the extra labelling functionality.

show(Name,Value) :-
  write(Name),write(' = '),
  write(Value),nl.

showr(Name,Value) :-
write(Name),write(' = '),
  reverse(Value,RValue),
  write(RValue),nl.
```

Note that the `check` predicate was coded within the maze finding problem. After finding it useful in the situation at hand, I decided to create the file of inspectors with future programming in mind. This is typicaly how utility files come to be. Programs in the next lesson will illstrate the use of these utilities.