School of Science and Engineering

CSC 4301

Intro. to Artificial Intelligence

Summer 2022

Project #2 Report

Mahmoud Erradi

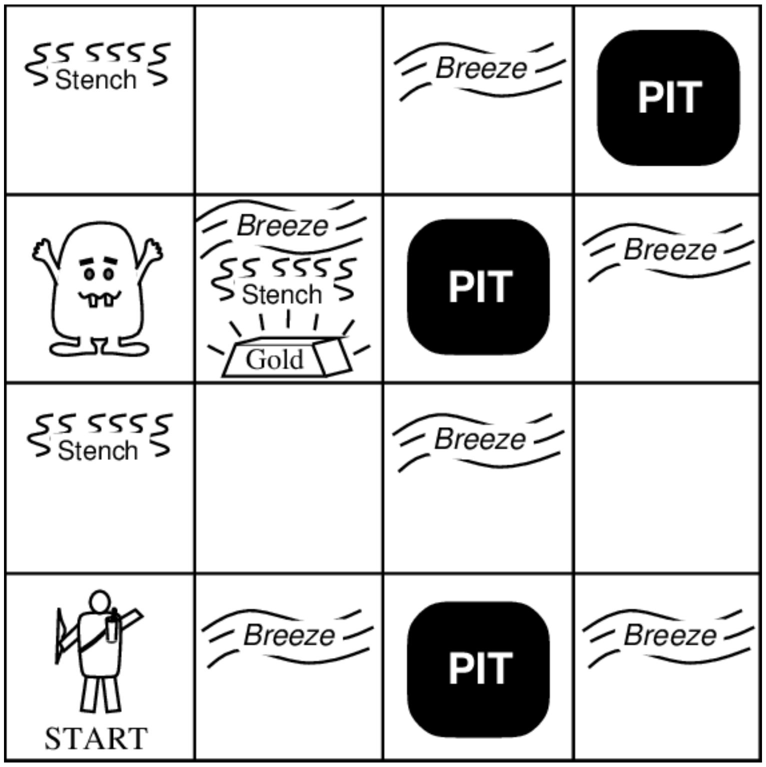
Teammate: Imane Ettabaa

Supervised by:

Dr. Tajjeeddine RACHIDI

Introduction:

Wumpus World is a game wherein a hunter explores a dark cave in search of a bounty of gold. The cave is represented by a 4x4 grid totaling 16 rooms. The hunter, normally, has no knowledge of the rooms beyond his own. The cave is mired with dangers. A few rooms contain bottomless pits in which the hunter may fall into. One room in the cave is home the Wumpus, another danger to the hunter's life. A few rooms have indicators that alert the hunter to a nearby threat. If the hunter feels a breeze, it means that a bottomless pit may be in an adjacent room. If the hunter smells a stench, it means that the Wumpus may be in an adjacent room. If the hunter detects glitter, it means that there is gold in the current gold and may be collected by the hunter. To achieve his mission, the hunter must infer danger by gathering knowledge about the cave.

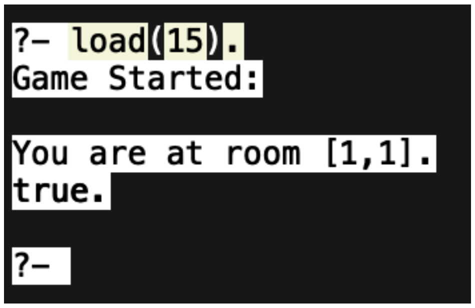


A Typical Wumpus World.

Our project aims, not to reprogram the entirety of the Wumpus World game, but to bundle a few of the essential features into a program coded using the Prolog language. We will see those feature in this report.

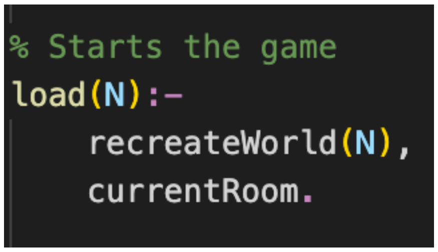
Let's start the game.

The picture seen above represents but one of the many starting configurations of the game. Indeed, we can evaluate the performance of our agent using the starting configurations found here: https://github.com/alexroque91/wumpus-world-prolog/blob/master/worldBuilder.pl



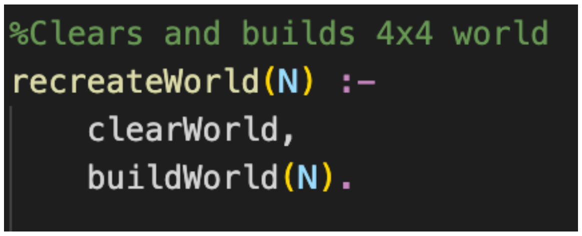
```
?- load(15).
Game Started:

You are at room [1,1].
true.

?-
```

In our program, the load/1 predicate can take a value from 1 to 2000, corresponding to the possible starting configurations. When successful, it returns true, as well as a "Game Started" message and the current room occupied by the hunter.



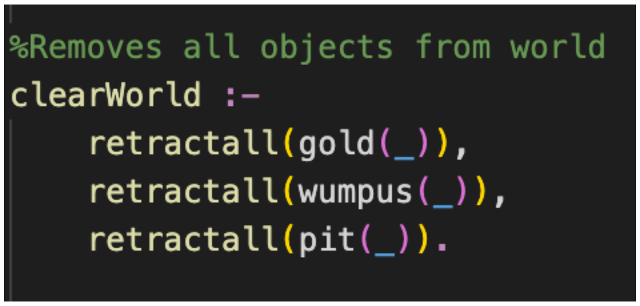
```
% Starts the game
load(N):-
    recreateWorld(N),
    currentRoom.
```

The load calls upon the recreateWorld/1 predicate,



```
%Clears and builds 4x4 world
recreateWorld(N) :-
    clearWorld,
    buildWorld(N).
```

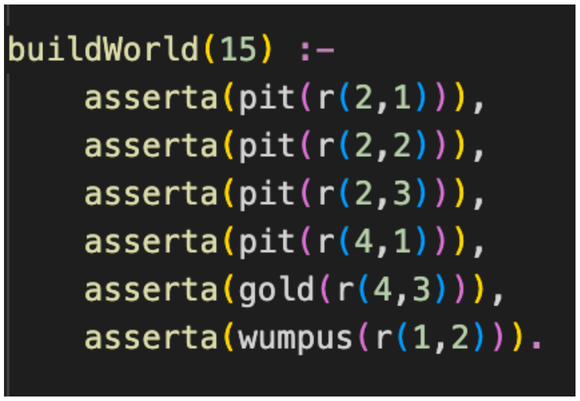
which call upon clearWorld/0



```
%Removes all objects from world
clearWorld :-
    retractall(gold(_)),
    retractall(wumpus(_)),
    retractall(pit(_)).
```

To remove all objects from the current starting configuration. As such, load can be used many times throughout the program to change the configuration.

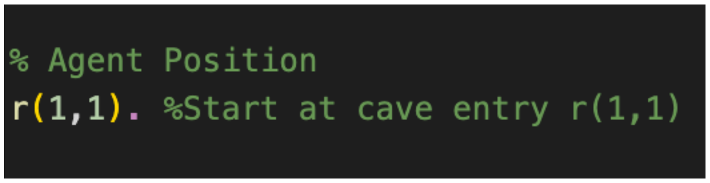
buildWorld/1 takes the value given to load/1 and uses it to create a unique starting configuration:



```
buildWorld(15) :-
    asserta(pit(r(2,1))),
    asserta(pit(r(2,2))),
    asserta(pit(r(2,3))),
    asserta(pit(r(4,1))),
    asserta(gold(r(4,3))),
    asserta(wumpus(r(1,2))).
```

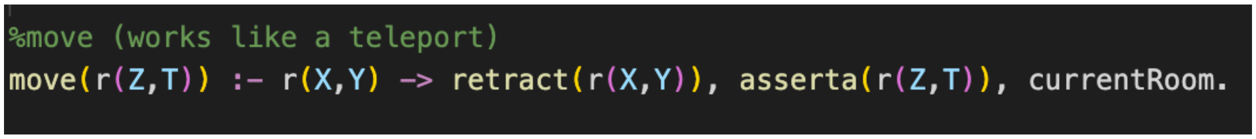
buildWorld for starting configuration 15

Now, whenever a world is created, the user is informed of the current room the hunter is occupying. In-code, the hunter always starts at room (1,1):



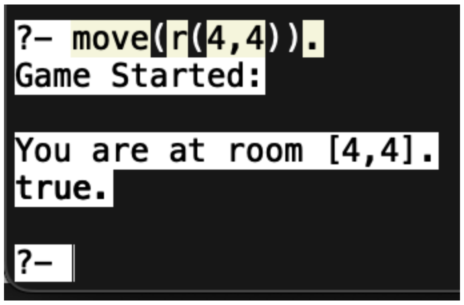
```
% Agent Position
r(1,1). %Start at cave entry r(1,1)
```

However, this can be changed using another predicate: move/1



```
%move (works like a teleport)
move(r(Z,T)) :- r(X,Y) -> retract(r(X,Y)), asserta(r(Z,T)), currentRoom.
```

The move/1 predicate was added later in order to give some playability to the “game.” It allows for the testing of the features implemented without having to change the current configuration or the code itself. It come quite in handy.

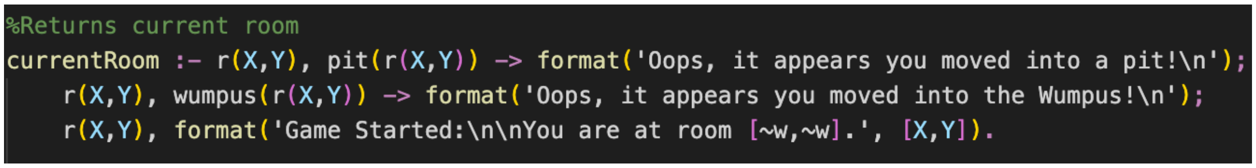


```
?- move(r(4,4)).
Game Started:

You are at room [4,4].
true.

?-
```

move/1 in action

Note: As stated in the comment above, move works more like a teleport, as it was implemented only to test other features of the program. As we saw in the screenshot. The hunter “moved” from room (1,1) to room (4,4), which is normally impossible according to the rules of the game.

currentRoom/0 serves another purpose: Let’s look at the code:



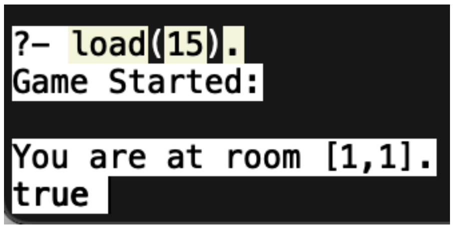
```
%Returns current room
currentRoom :- r(X,Y), pit(r(X,Y)) -> format('Oops, it appears you moved into a pit!\n');
    r(X,Y), wumpus(r(X,Y)) -> format('Oops, it appears you moved into the Wumpus!\n');
    r(X,Y), format('Game Started:\n\nYou are at room [~w,~w].', [X,Y]).
```

The currentRoom/0 predicate checks first the position of the hunter and compares it with the position of the pits and Wumpus of the current world. While implementing the feature to load multiple starting configurations of the world, it was possible to have the hunter fall directly into a bottomless pit or into the Wumpus room at the beginning of the game! For this reason, the hunter's position was fixed at room (1,1) and the move predicate was implemented. Until then, however, currentRoom/0 would inform the user about the hunter's predicament whenever a new world was loaded.

Note that now, with the move predicate, it is possible to move into a bottomless pit or into the Wumpus. currentRoom informs the user of that.

Playing the game, really!

Now, that we know how to load a new world. Let's load a starting configuration and start a new game.



```
?- load(15).
Game Started:

You are at room [1,1].
true
```

What can we do?

At this point, the hunter has a few options:

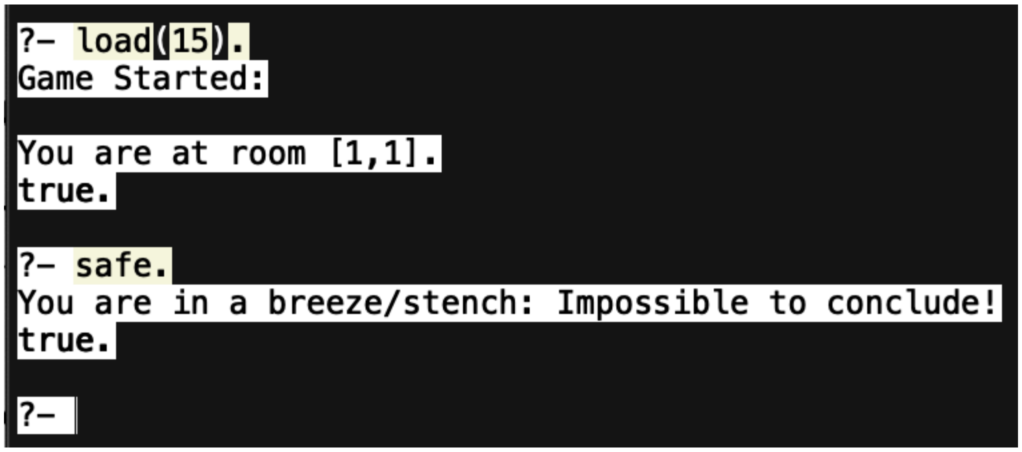
1. Checking the safety of the adjacent rooms.
2. Checking the room for any gold.
3. Shooting in the adjacent rooms in an attempt to hit the Wumpus.
4. Moving.

We will experiment with the first three using different configurations:

safe/0:

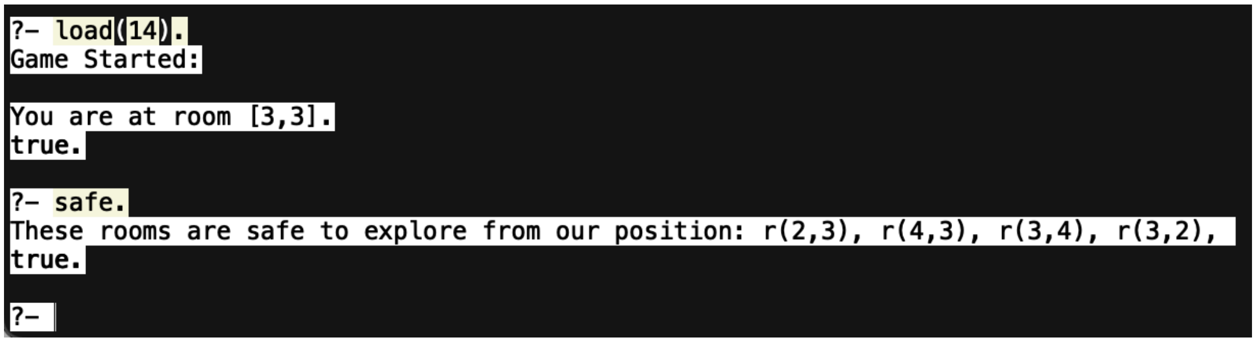
We can check the safety of adjacent rooms using the safe/0 predicate.

Let's try it!


```
?- load(15).
Game Started:

You are at room [1,1].
true.

?- safe.
You are in a breeze/stench: Impossible to conclude!
true.

?- 
```

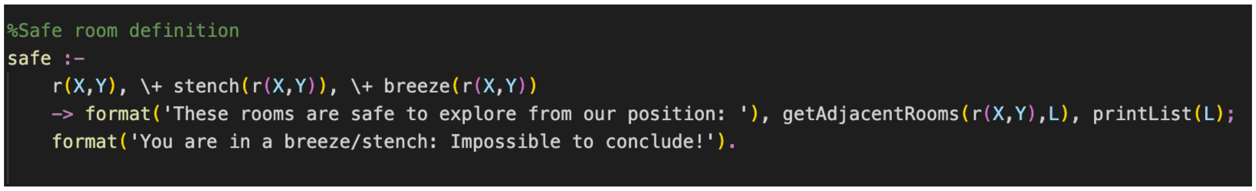
It appears our attempt failed. Indeed, if the hunter is in a room with stench or breeze then it is impossible for him to definitively conclude the safety of the nearby rooms. This is a situation when safe fails completely.

To showcase when safe works, we will have to load a new starting configuration, or to move to another room.


```
?- load(14).
Game Started:

You are at room [3,3].
true.

?- safe.
These rooms are safe to explore from our position: r(2,3), r(4,3), r(3,4), r(3,2), 
true.

?- 
```

With the absence of breeze/stench, the hunter should be able to definitively conclude the safety of the adjacent rooms.

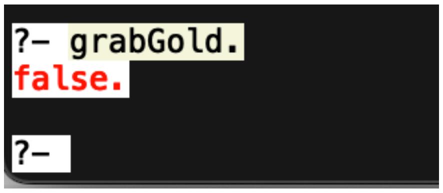
safe/0:


```
%Safe room definition
safe :-
    r(X,Y), \+ stench(r(X,Y)), \+ breeze(r(X,Y))
    -> format('These rooms are safe to explore from our position: '), getAdjacentRooms(r(X,Y),L), printList(L);
    format('You are in a breeze/stench: Impossible to conclude!').
```

It uses a predicate for getting the adjacent rooms in a list, then another for printing the list.

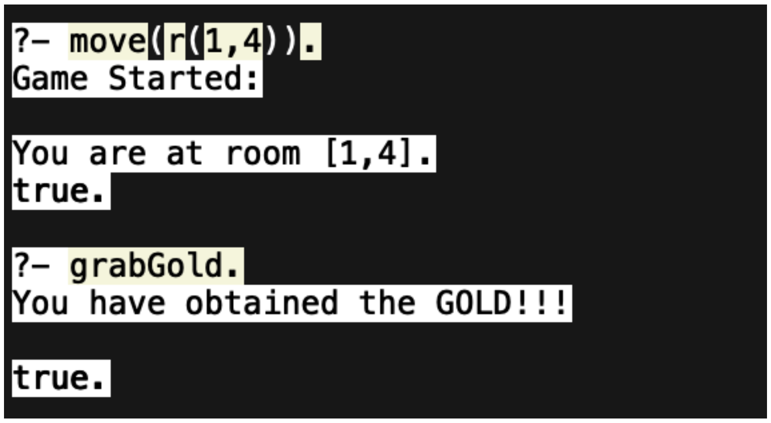
grabGold/0:

Now let’s try to grab some gold! Attempting to use the grabGold predicate yields:



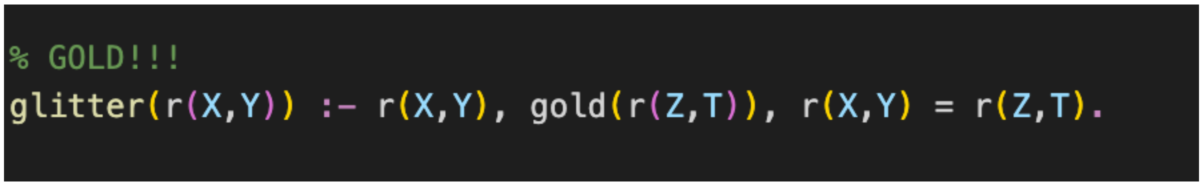
```
?- grabGold.
false.

?-
```

False, because there is no gold in our current position, so let's try to move to a room with gold in it. We are currently in starting configuration 14, which has gold in position (1,4), so let's move there.



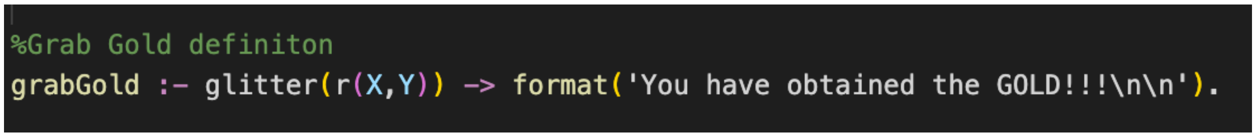
```
?- move(r(1,4)).
Game Started:

You are at room [1,4].
true.

?- grabGold.
You have obtained the GOLD!!!

true.
```

As expected, we have obtained the gold. Whenever a new world is created, a room containing gold is generated. As per the rules of the game, gold should emit glitter, which is implemented using the following predicate:



```
% GOLD!!!
glitter(r(X,Y)) :- r(X,Y), gold(r(Z,T)), r(X,Y) = r(Z,T).
```

Which puts glitter in whatever position the gold happened to spawn in. Then the grabGold predicate checks if glitter is indeed in the current room:



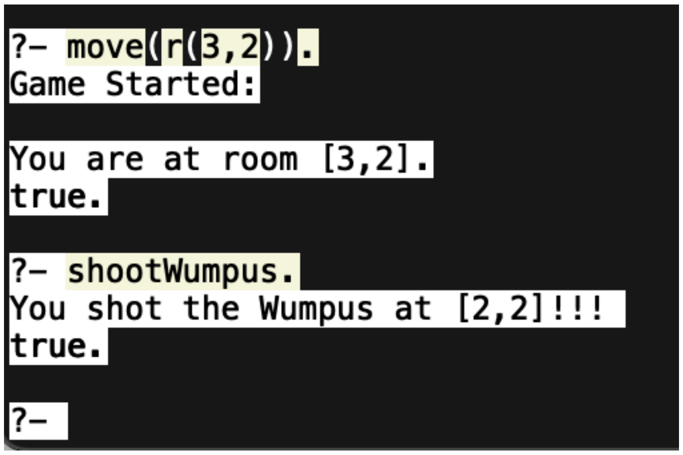
```
%Grab Gold definiton
grabGold :- glitter(r(X,Y)) -> format('You have obtained the GOLD!!!\n\n').
```

Otherwise, it returns a false.

Now let's try to shoot the Wumpus.

shootWumpus/0:

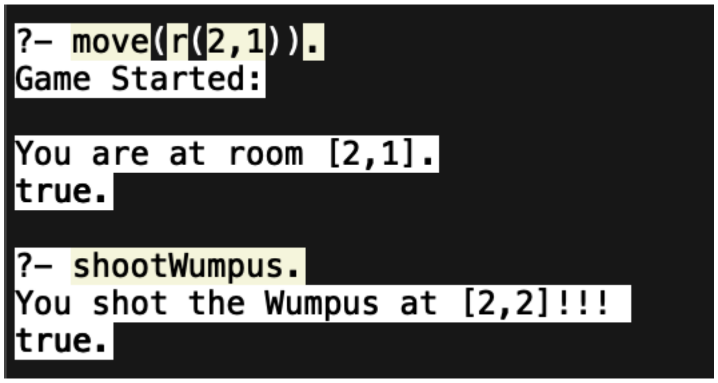
First, let's move to a room adjacent to the one where the Wumpus is. In configuration 14, the Wumpus is in room (2,2), so let's move to (3,2), which is safe and adjacent to the Wumpus room.



```
?- move(r(3,2)).
Game Started:

You are at room [3,2].
true.

?- shootWumpus.
You shot the Wumpus at [2,2]!!!
true.

?-
```

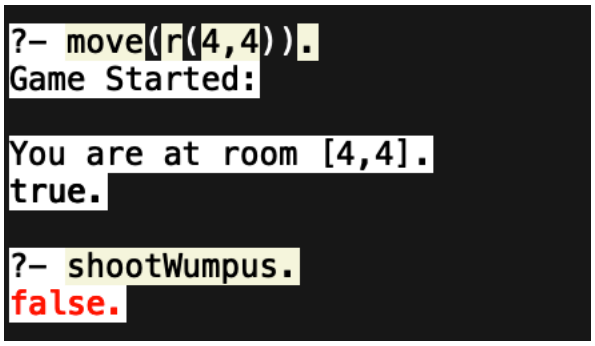
We successfully shot the Wumpus, which was in room (2,2)! As per the requirements of the project, shootWumpus checks every adjacent room for the Wumpus and returns a true if it was found.

So, the Wumpus can be shot from every adjacent room like room (2,1):



```
?- move(r(2,1)).
Game Started:

You are at room [2,1].
true.

?- shootWumpus.
You shot the Wumpus at [2,2]!!!
true.
```

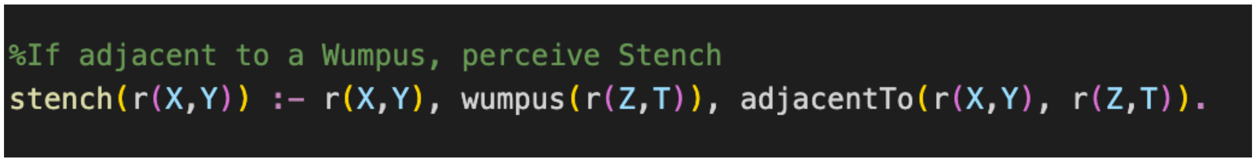
If successful, shootWumpus returns the position of the Wumpus. In this case room (2,2).



```
?- move(r(4,4)).
Game Started:

You are at room [4,4].
true.

?- shootWumpus.
false.
```

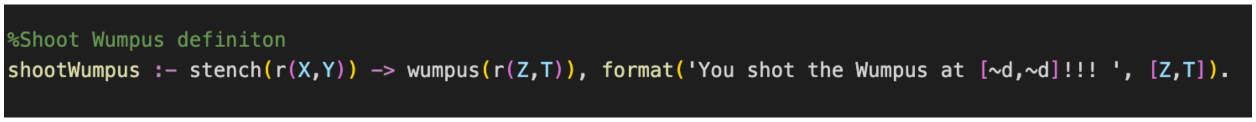
Of course, shootWumpus returns false if the Wumpus is not in one of the adjacent rooms.

This predicate allows stench to come out of the rooms adjacent to whichever one the Wumpus happened to spawn in.



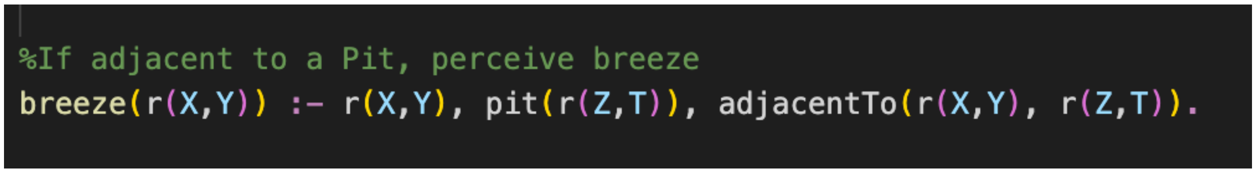
```
%If adjacent to a Wumpus, perceive Stench
stench(r(X,Y)) :- r(X,Y), wumpus(r(Z,T)), adjacentTo(r(X,Y), r(Z,T)).
```

And shootWumpus checks for that stench:



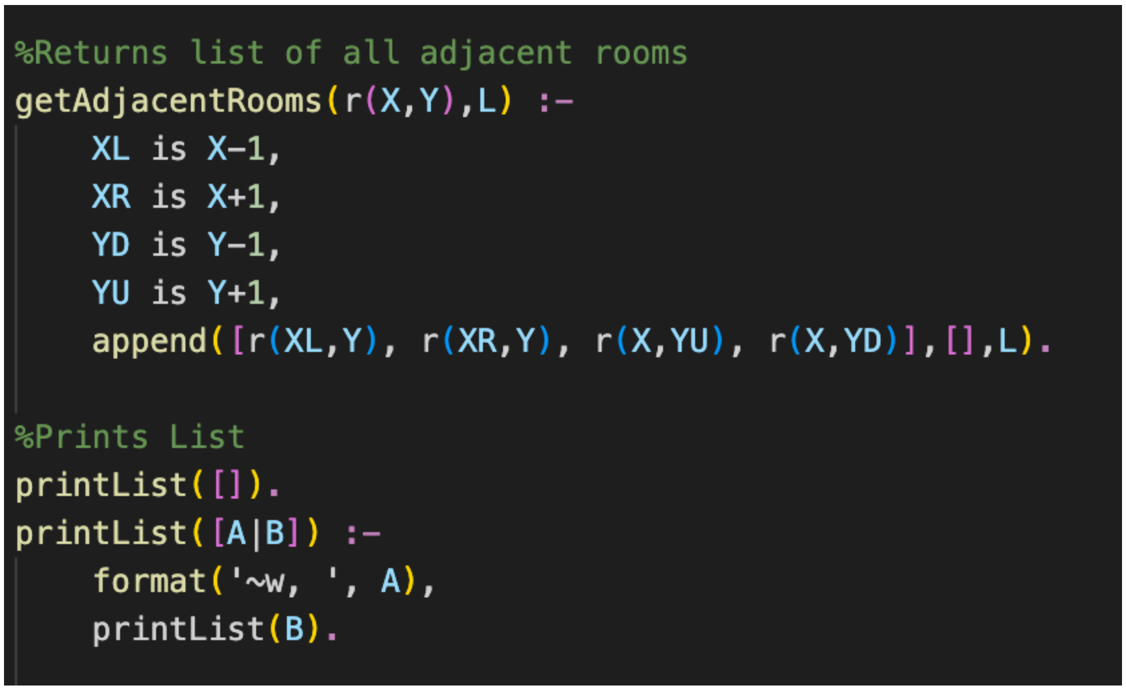
```
%Shoot Wumpus definiton
shootWumpus :- stench(r(X,Y)) -> wumpus(r(Z,T)), format('You shot the Wumpus at [~d,~d]!!! ', [Z,T]).
```

Breezes work similarly in this program, appearing in the rooms adjacent to the ones with pits.



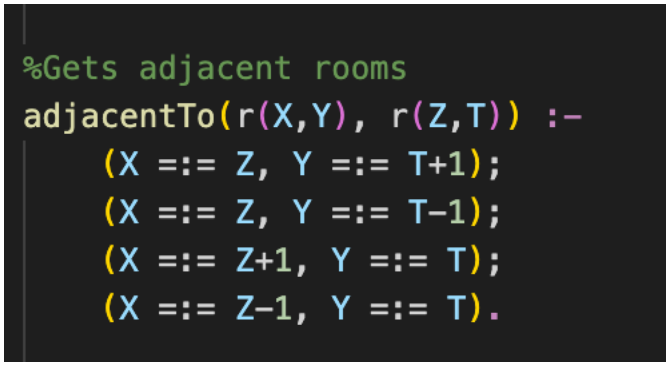
```
%If adjacent to a Pit, perceive breeze
breeze(r(X,Y)) :- r(X,Y), pit(r(Z,T)), adjacentTo(r(X,Y), r(Z,T)).
```

Other useful stuff:



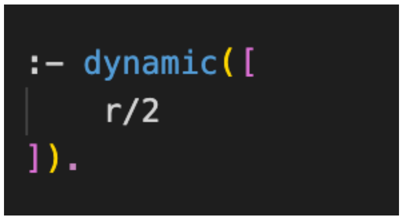
```
%Returns list of all adjacent rooms
getAdjacentRooms(r(X,Y),L) :-
    XL is X-1,
    XR is X+1,
    YD is Y-1,
    YU is Y+1,
    append([r(XL,Y), r(XR,Y), r(X,YU), r(X,YD)],[],L).

%Prints List
printList([]).
printList([A|B]) :-
    format('~w, ', A),
    printList(B).
```

Those were used to get a list of adjacent rooms and print them.



```
%Gets adjacent rooms
adjacentTo(r(X,Y), r(Z,T)) :-
    (X =:= Z, Y =:= T+1);
    (X =:= Z, Y =:= T-1);
    (X =:= Z+1, Y =:= T);
    (X =:= Z-1, Y =:= T).
```

To get each adjacent room to later be checked for attributes.



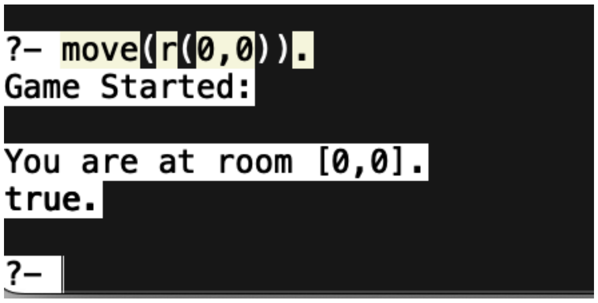
```
:- dynamic([
    r/2
]).
```

r/2 was made dynamic so it may be changed at runtime.

Limitations of the solution:

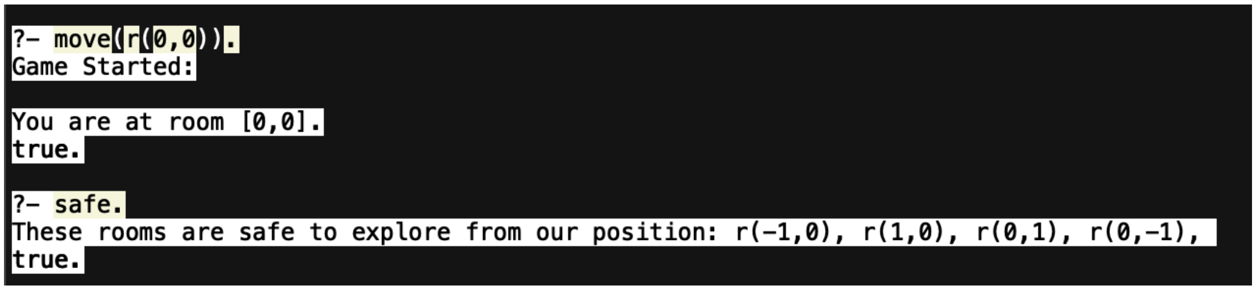
The results seen in the screenshots above are consistent, meaning that they always yield the same results given the same starting configurations. However, there are some ways to make the code fail:

Using the move/1 predicate, it is possible to go out of bounds, as such:



```
?- move(r(0,0)).
Game Started:

You are at room [0,0].
true.

?- 
```

It is also possible, then, to have the safe/0 predicate return impossible rooms:



```
?- move(r(0,0)).
Game Started:

You are at room [0,0].
true.

?- safe.
These rooms are safe to explore from our position: r(-1,0), r(1,0), r(0,1), r(0,-1),
true.
```

There is no point in going out of bounds, as nothing is generated there. Nevertheless, it is possible.

This problem can be solved with more time by implementing wall-like features to limit the scope of the game.
Otherwise, every other feature works as intended and the results remain consistent with our expectations.

References:

Alexander Roque's github: https://github.com/alexroque91

Wumpus-world-prolog on Alexander Roque's github: https://github.com/alexroque91/wumpus-world-prolog