Workshop 5 (Mapping the world)

Exercise 5: Bumble 1.3 – Mapper

Note: ***This is the last workshop that takes you through development in detail. After you have completed this workshop you should have all the necessary tools to make Bumble (& any other animate object in V-World) more intelligent & purposeful. This workshop puts the responsibility on Bumble to use his perceptions & relevant decision-making to progressively build the kind of map that was given in the previous workshop. Remember though, this is just one way to build the map & it is based firmly on Nute's ideas. It would be possible to use 'extra perceptions', such as ray casting (say) to extend Bumble's capabilities.***

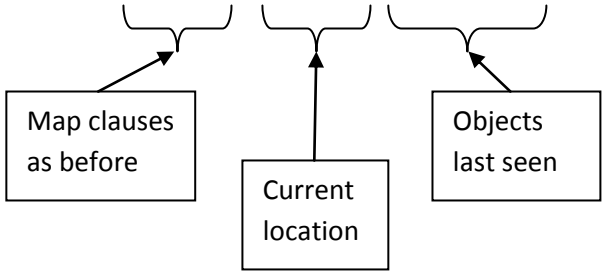
In Exercise 4, we explored search methods for finding paths to objects in virtual worlds, but those methods use a predefined map of the world. The next step in improving Bumble is to modify the agent so that it constructs a map of its world as it moves around.

Modify your code for Bumble 1.2 by changing

:- dynamic [agent/2,tried/0,last/1,pushed/0,hungry/0,hurt/0].

to

```
:- dynamic
[agent/2,tried/0,pushed/0,hungry/0,hurt/0,mymap/3,here/2,last_seen/3].
```



and by changing

```
agent(Perceptions,Action) :-
        set_bumble_states(Perceptions),
        bumble(Perceptions,Action).
```

to

```
agent(Perceptions,Action) :-
      set_bumble_states(Perceptions),
      build_bumble_map(Perceptions),      % new in 1.3
      bumble(Perceptions,Action).
```

Now you need to define the predicate build_bumble_map/1 to update Bumble's map (stored in clauses for mymap/3) and Bumble's current location on his map (stored in a single fact for here/2) each turn before bumble/2 is called. We will also use last_seen/3 to record the locations where transient objects (collectable objects, animate objects, etc.) were last seen.

At Bumble's first turn in a world, no map will exist. In this case, Bumble should "initialise" his location as (0,0) and enter information about the area of the world he can see. But how can Bumble tell that this is his first move? Well, he knows it must be his first move because

he doesn't have a map and he doesn't know where he is! So the first clause for our predicate should look something like this:

```
build_bumble_map(Perceptions):-
      \+ here(_,_), !,          % first time, no map
      assert(here(0,0)),        % always starts at (0,0)
      assert(mymap(0,0,o)),     % must be open space
      update_bumble_map(0,0,Perceptions).
```

After the first move, Bumble will get its current location from the predicate here/2 and its last location. This will be easy if Bumble did not move or did not go through a door. If Bumble did not move, then there is no need to update the map.

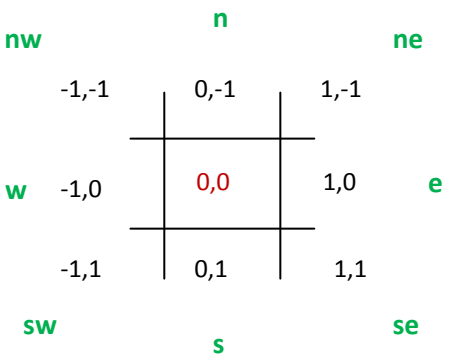
```
build_bumble_map([_,_,_,Dir,_,_,_,_,_]):-
      Dir=same, ! .          % no move - cut search - nothing
```

If Bumble did move but did not go through a door, then we can easily compute Bumble's position. For example, if Bumble's last position was northwest of Bumble's current position, then Bumble moved southeast and we need to add 1 to each coordinate in the position stored in here/2 in the last move. *(In WIN-PROLOG, the X coordinate of screen position goes up as we move to the right, and the Y coordinate goes up as we move down the screen. You can of course use another system if you wish so long as you are consistent.).* **Note from GW: this is actually inconsistent with the previous workshop, but some V-World access predicates use this convention. I decided therefore that it is logical to go along with it.**

```
build_bumble_map([ , , ,Dir,V1,V2,V3,V4,V5]):-
      \+ tried, !, % bumble did not go through a door last time
      here(X,Y),
      member(Dir,[nw,n,ne,e,se,s,sw,w],Pos),
      member((XX,YY), [(1,1),(0,1),(-1,1),(-1,0),(-1,-1),
                       (0,-1),(1,-1),(1,0)]Pos),
      NewX is X+XX,
      NewY is Y+YY,
      retractall(here(_,_)),
      assert(here(NewX,NewY)),
      update_bumble_map(NewX,NewY,[_,_,_,Dir,V1,V2,V3,V4,V5]).
```

Important note:

Although **nw** (for example) is (-1,-1), the direction of Bumble must be in the opposite, i.e. **se** direction, so in the second member/3 clause above, the X,Y coordinates are the numbers relating to the direction Bumble must be going.



If Bumble **went through a door, then tried will** succeed and Last will not have same as its value. In this case, you must determine from here(X,Y) and Last the coordinates for the door Bumble went through. Then you must look around in Bumble's new location to locate the door. Then you must use that information to compute Bumble's new location and store it as here(NewX,NewY). It is left as an exercise for you to define this last clause for build_bumble_map/1.

It remains to define update_bumble_map/1. Each location in Bumble's visual field will correspond to some set of values to be added to Bumble's current location. You can compute structures of three items (X,Y,Obj) where Obj is the object at coordinates (X,Y) for all locations within Bumble's visual field all at once using findall/3:

```
update_bumble_map(BX,BY,[_,_,_,_,
          [NWNW,NNW,NN,NNE,NENE],          % V1
          [WNW,NW,N,NE,ENE],               % V2
          [WW,W,_,E,EE],                   % V3
          [WSW,SW,S,SE,ESE],               % V4
          [SWSW,SSW,SS,SSE,SESE]]):-       % V5
     here(BX,BY),
     findall((X,Y,Obj),
     (member(Obj,[NWNW,NNW,NN,NNE,NENE,
          WNW,NW,N,NE,ENE,WW,W,E,EE,
          WSW,SW,S,SE,ESE,SWSW,SSW,SS,SSE,SESE],Pos),
     member((XX,YY),[(-2,-2),(-1,-2),(0,-2),
          (1,-2),(2,-2),(-2,-1),(-1,-1),
          (0,-1),(1,-1),(2,-1),(-2,0),
          (-1,0),(1,0),(2,0),(-2,1),(-1,1),
          (0,1),(1,1),(2,1),(-2,2),(-1,2),
          (0,2),(1,2),(2,2)],Pos),
     X is BX+XX,
     Y is BY+YY ),
     List),
     update_bumble_map_x(List).
```

update_bumble_map_x/1 should process each member of List using rules like this to update each triple (X,Y,Obj) in List.

1. If Obj is cant_see, then don't add the coordinates to the map.

2. If mymap(X,Y,_) succeeds, then you don't need to update the map. However, if Obj is some object that you want to keep track of (perhaps it is an animate object you may need to find again later,) then you may want to assert last_seen(Obj,X,Y). If you think there is only one of this Obj and you already have saved its location using last_seen/3, then you may want to delete its old location.

3. If mymap(X,Y,_) fails and Obj is a permanent object like a wall, a tree, a cross, or a door, then you should add mymap(X,Y,Obj) to the map.

4. If mymap(X,Y,_) fails and Obj is not a permanent object, then you should add mymap(X,Y,o) to your map. But again, you may want to save the locations of some non-permanent objects (collectables, animate objects, etc.) using last_seen/3.