Exercise 8: Prolog Adventure and Riddle

# Exercise 8.1: Adventure

Most of my code and added predicates are at the bottom, below the /\*\*\*\*/.

Note, I was unable to run the program on SWISH so i'm assuming my predicates work. I did run them through chatGPT to see if they can “compile” it and see if the functions logic works as intend.

* /\*
* This is a little adventure game. There are three
* entities: you, a treasure, and an ogre. There are
* six places: a valley, a path, a cliff, a fork, a maze,
* and a mountaintop. Your goal is to get the treasure
* without being killed first.
* \*/
* /\*
* First, text descriptions of all the places in
* the game.
* \*/
* description(valley, 'You are in a pleasant valley, with a trail ahead.').
* description(path, 'You are on a path, with ravines on both sides.').
* description(cliff, 'You are teetering on the edge of a cliff.').
* description(fork, 'You are at a fork in the path.').
* description(maze(\_), 'You are in a maze of twisty trails, all alike.').
* description(mountainroad, 'You are on a long dirt path up to the peak of the mountain.').
* description(mountaintop, 'You are on the mountaintop.').
* description(gate, 'You are at a gate blocking the path to the mountaintop.').
* /\*
* report prints the description of your current
* location.
* \*/
* report :-
* at(you,X),
* description(X,Y),
* write(Y), nl.
* /\*
* These connect predicates establish the map.
* The meaning of connect(X,Dir,Y) is that if you
* are at X and you move in direction Dir, you
* get to Y. Recognized directions are
* forward, right, and left.
* \*/
* connect(valley,forward,path).
* connect(path,right,cliff).
* connect(path,left,cliff).
* connect(path,forward,fork).
* connect(fork,left,maze(0)).
* connect(fork,forward,gate).
* connect(gate,forward,mountainroad).
* connect(mountainroad,forward,mountaintop),
* connect(maze(0),left,maze(1)).
* connect(maze(0),right,maze(3)).
* connect(maze(1),left,maze(0)).
* connect(maze(1),right,maze(2)).
* connect(maze(2),left,fork).
* connect(maze(2),right,maze(0)).
* connect(maze(3),left,maze(0)).
* connect(maze(3),right,maze(3)).
* /\*
* move(Dir) moves you in direction Dir, then
* prints the description of your new location.
* \*/
* move(Dir) :-
* at(you,Loc),
* connect(Loc,Dir,Next),
* retract(at(you,Loc)),
* assert(at(you,Next)),
* report,
* !.
* /\*
* But if the argument was not a legal direction,
* print an error message and don't move.
* \*/
* move(\_) :-
* write('That is not a legal move.\n'),
* report.
* /\*
* Shorthand for moves.
* \*/
* forward :- move(forward).
* left :- move(left).
* right :- move(right).
* /\*
* If you and the ogre are at the same place, it
* kills you.
* \*/
* ogre :-
* at(ogre,Loc),
* at(you,Loc),
* write('An ogre sucks your brain out through\n'),
* write('your eye sockets, and you die.\n'),
* retract(at(you,Loc)),
* assert(at(you,done)),
* !.
* /\*
* But if you and the ogre are not in the same place,
* nothing happens.
* \*/
* ogre.
* /\*
* If you and the treasure are at the same place, you
* win.
* \*/
* treasure :-
* at(treasure,Loc),
* at(you,Loc),
* write('There is a treasure here.\n'),
* write('Congratulations, you win!\n'),
* retract(at(you,Loc)),
* assert(at(you,done)),
* !.
* /\*
* But if you and the treasure are not in the same
* place, nothing happens.
* \*/
* treasure.
* /\*
* If you are at the cliff, you fall off and die.
* \*/
* cliff :-
* at(you,cliff),
* write('You fall off and die.\n'),
* retract(at(you,cliff)),
* assert(at(you,done)),
* !.
* /\*
* But if you are not at the cliff nothing happens.
* \*/
* cliff.
* /\*
* Main loop. Stop if player won or lost.
* \*/
* main :-
* at(you,done),
* write('Thanks for playing.\n'),
* !.
* /\*
* Main loop. Not done, so get a move from the user
* and make it. Then run all our special behaviors.
* Then repeat.
* \*/
* main :-
* write('\nNext move -- '),
* read(Move),
* call(Move),
* ogre,
* lightning,
* treasure,
* cliff,
* main.
* /\*
* This is the starting point for the game. We
* assert the initial conditions, print an initial
* report, then start the main loop.
* \*/
* go :-
* retractall(at(\_,\_)), % clean up from previous runs
* retractall(inventory(\_)),
* assert(at(you,valley)),
* assert(at(ogre,maze(3))),
* assert(at(lightning,mountainroad)),
* assert(at(treasure,mountaintop)),
* assert(inventory([])),
* assert(at(key,maze(2))),
* write('This is an adventure game. \n'),
* write('Legal moves are left, right, or forward.\n'),
* write('End each move with a period.\n\n'),
* report,
* main.
* /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
* /\*
* If they pass the gate with the key.
* \*/
* lightning :-
* at(lightning,Loc),
* at(you,Loc),
* inventory(Inv),
* member(key,Inv),
* write('As you pass through the gate, lightning strikes the metal key and you die.\n'),
* retract(at(you,Loc)),
* assert(at(you,done)),
* !.
* /\*
* But if yo udont have the key and you dont pass the gate.
* \*/
* lightning.
* /\*
* Pickup and Putdown.
* \*/
* pickup(Obj) :-
* at(you,Loc),
* at(Obj,Loc),
* retract(at(Obj,Loc)),
* inventory(Inv),
* retract(inventory(Inv)),
* assert(inventory([Obj|Inv])),
* write('You picked up the '), write(Obj), write('.\n'),
* !.
* pickup(\_) :-
* write('Nothing to pick up here.\n').
* putdown(Obj) :-
* inventory(Inv),
* member(Obj,Inv),
* at(you,Loc),
* retract(inventory(Inv)),
* delete(Inv, Obj, NewInv),
* assert(inventory(NewInv)),
* assert(at(Obj,Loc)),
* write('You put down the '), write(Obj), write('.\n'),
* !.
* putdown(\_) :-
* write('You dont have that to put down.\n').
* /\*
* Delete items from list.
* \*/
* delete([H|T], H, T).
* delete([H|T], X, [H|R]) :-
* delete(T, X, R).
* /\*
* Use item.
* \*/
* use(Item) :-
* at(you,Loc),
* inventory(Inv),
* member(Item,Inv),
* can\_use(Item, Loc),
* !.
* can\_use(key, gate) :-
* write('You use the key to unlock the gate.\n'),
* retract(at(key,gate)),
* assert(at(key,used)),
* !.
* can\_use(\_, \_) :-
* write('That item serves no purpose here.\n'),
* fail.
* use(\_) :-
* write('That item serves no purpose here.\n').

# Exercise 8.2: Riddle

// see if the head is empty, if so, return true

riddle(X, \_) :- // check head of list, take in X and \_ the rest

length(X, XL), // get length of X, unify X and XL

XL = 0. // if XL = 0 then X = 0 then clause = true

// see if tail is empty, if so, return true

riddle(\_, Y) :- // same as X but checking if “the rest” is length 0.

length(Y, YL),

YL = 0.

riddle([], \_). // see if head is empty list, if so return true

riddle(\_, []). // see if tail is empty list, if so return true