Prolog Design

Implementing a Prolog program requires a different method of approach. Whilst you can lay out you program in similar ways with ‘method-like’ structure, the way you create them are different. Prolog has a lot of built in predicates, my implementation limits the use of these to a small amount to show my ability of writing Prolog programs.

This was the second language and paradigm I chose to tackle.

Pre-implementation

Before diving into the code, I knew it would be best to layout a plan for the flow.

Layout Pseudocode

My high-level implementation goes as follows:

1. **Introduction - One predicate with many console prints.**
    a. Explain the game - format/2.
    b. Display building numbers - Separate predicate.
    c. Get user input to continue (break up sections of intro so it’s not overwhelming) - read or get_char.
2. **Recursive game loop - Call until winner found.**
    a. Set up building arrays.
    b. Display the buildings - Loop rows.
        i. Print each row in the building matrixes.
    c. Check for winner.
        i. P1 win if P1 building is all 1s or P2 is all 0s - Head tail every room.
        ii. P2 win if P2 building is all 1s or P1 is all 0s - Head tail every room.
    d. Take turn.
    e. Recursively loop by calling game loop

3. **Take turn - Pass in player**
    a. Get player input for 1 or 0 (water or fire).
    b. Get player input for 1-20 (room number).
    c. Get the row and column needed to update
        i. Input is 1-20 where 1 is effectively (0,0) on a 5 x 4 graph and 20 is (5, 4). When player enters room 1, the element in the array is [3][0].
    d. Spread water or fire
4. **Spread water or fire - Same predicate twice for each player**
    a. If player = 1 and water, spread water on P1 building, update P1 building.
    b. If player = 1 and fire, spread fire on P2 building, update P1 building.
    c. If player = 2 and water, spread water on P2 building, update P1 building.
    d. If player = 2 and fire, spread fire on P1 building, update P1 building.
5. **Recursively spread water**
    a. Check if we're not on the bottom row.
    b. If on bottom are, fail and move into new predicate to just change that element to a 1.
    c. If not, check if there is a 1 above or below, if so:
    d. Recursively
        i. Go through each row below.
        ii. Replace the element in the same row with a 1.
        iii. Update building array by storing in a new building array and set that to be the corresponding building.
6. **Recursively spread fire**
    a. Same logic as water but zeros and top row.

Implementation

Creating my Prolog solution, I knew I would be making features for the game across multiple days. After creating a feature, I would test and then push to a private GitHub repository. This

also allows the ability to roll back to prior versions if later ones become messy and broken.

Commit 1 - Intro & Display Buildings

I began with the basics - print the rules and the buildings to the players.

```
+ % Player one building array
+ playerOneBuilding([[ 1, 0, 0, 0, 1],    % 16. 17. 18. 19. 20.
+                    [ 0, 1, 0, 1, 0],    % 11. 12. 13. 14. 15.
+                    [ 0, 1, 0, 1, 0],    % 6.  7.  8.  9.  10.
+                    [ 0, 0, 1, 0, 0]]).   % 1.  2.  3.  4.  5.
+
+ % Player two building array
+ playerTwoBuilding([[ 1, 0, 0, 0, 1],    % 16. 17. 18. 19. 20.
+                    [ 0, 1, 0, 1, 0],    % 11. 12. 13. 14. 15.
+                    [ 0, 1, 0, 1, 0],    % 6.  7.  8.  9.  10.
+                    [ 0, 0, 1, 0, 0]]).   % 1.  2.  3.  4.  5.
+
+ % play/0
+ play :-
+     intro,
+     game_loop.
+
+ % intro/0
+ % Into the game
+ intro :-
+     format('~nYOUR BUILDING IS ON FIRE~n'),
+     format('And so is your opponents...~n'),
+     format('You hate your opponet...~n'),
+     format('You can save your building... or destroy your opponents...~n'),
+     format('1 = Water~n'),
+     format('0 = Fire~n'),
+     format('Press enter to start~n'),
+     display_both_buildings,
+     format('Press enter to continue~n'),
+     format('These are the building numbers~n'),
+     display_building_numbers,
+     format('~nPress enter to continue~n').
+
+ % display_building_numbers/0
+ display_building_numbers :-
+     format('~n 16.  17.  18.  19.  20.~n'),
+     format(' 11.  12.  13.  14.  15.~n'),
+     format(' 6.   7.   8.   9.   10.~n'),
+     format(' 1.   2.   3.   4.   5.~n').
+
+ % game_loop/0
+ game_loop :-
+     format('~nLets begin~n').
+     % display_both_buildings.
+
+ % display_both_buildings/0
+ % Print the building of both players
+ display_both_buildings :-
+     playerOneBuilding(PlayerOneBuilding),   % Set player one building array
+     playerTwoBuilding(PlayerTwoBuilding),   % Set player two building array
+     format('~nPlayer One Building:~n'),
+     display_building(PlayerOneBuilding),    % Print player one building
+     format('~nPlayer Two Building:~n'),
+     display_building(PlayerTwoBuilding).    % Print player two building
+
+ % display_building(+buildingArray)
+ % Print the building recursively by looping through each row
+ display_building([]).
+
+ display_building([BuildingFloor | RemainingFloors]) :-
+     print_row(BuildingFloor),               % Print the current row
+     nl,                                     % New line
+     display_building(RemainingFloors).                % Loop back to print the remaining rows in the
  building
+
+ % print_row(+row)
+ % Print a row of the building recursively by looping through each element (room number)
+ print_row([]).
+
+ print_row([RoomNumber | RemainingRooms]) :-
+     write(RoomNumber),                      % Print the room number
+     write(' '),                             % Print a space between the room number
+     print_row(RemainingRooms).              % Loop back to print the rest of the numebr in the row
+
+
+ :- play.
```

Line 7-17: Define the two building arrays.

Line 26-38: Print the rules. Here I used `format/2` as I knew from my prior experience with Prolog, it provides better flexibility for outputting and allows for the use of string interpolation (the ability to pass in strings to print within one line). '/2' is the arity, the number of parameters, here the second parameter is optional for this command so you could treat the arity as one.

Line 41-45: Print the building room numbers.

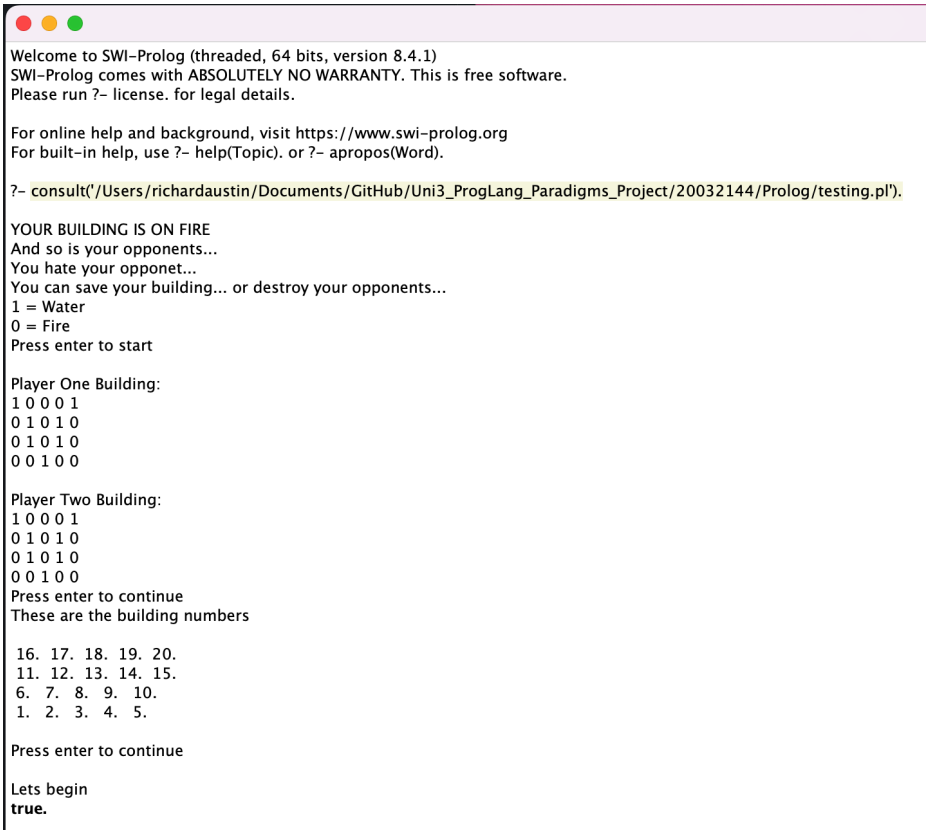
Line 54-60: Set Player 1 and 2's building arrays in memory and call display for both.

Line 64-69: The Prolog fun begins. 64 is a base case where the array is empty. 66 uses a head and tail functionality, where I named the variables appropriately for the game. Loop through each floor (row) of the building and call the print. 69 moves onto the next floor, etc.

Line 73-78: The same logic for 'display_building' but where that would get a row from an array, this gets an element from that row. Instead of format, i used `write` as I just need o to print the singular value, being the room value. Using format would require `format('~w', [RoomNumber])` so write is the more elegant solution here.

Line 81: Upon load, play will run. Prolog runs any code with the `:-` and no predicate name at the start of the program. Without this, the user has to type `play.` to start whereas this removes the need for that.

Testing in SWI Prolog.



I opened Prolog and consulted in the file. When consulted, the game starts (’play’ is called). It works as intended. The introduction prints, the buildings display, and the building numbers are shown.

Commit 2 - Check Winner

Before the next player takes a turn, we should check if a player has won.

```
31        format('1 = Water~n'),
32        format('0 = Fire~n'),
33        format('Press enter to start~n'),
34  +     get_char(_), % User input "_" as we dont care a
      variable
35        display_both_buildings,
36        format('Press enter to continue~n'),
37  +     get_char(_),
38        format('These are the building numbers~n'),
39        display_building_numbers,
40  +     format('~nPress enter to continue~n'),
41  +     get_char(_).
```

Starting with a small change, I added the code for user input to simply press enter to continue. While there are better solutions with `read/1`, however on Mac, I has issues with getting read to work and so I opted for `get_char/1` and underscored the input, as I don't need to assign it as it's an input we don't care about.

```
88  + %! check_winner(+PlayerOneBuilding, +PlayerTwoBuilding)
89  + % Check if player one has won
90  + % If fail, check_winner will run the next attempt below, for p2
91    check_winner(PlayerOneBuilding, PlayerTwoBuilding) :-
92        check_player_one_win(PlayerOneBuilding, PlayerTwoBuilding),
93  +     format('Player 1 wins~n'), !.
94
95  + % Check if player two has won
96  + % If fail, check_winner will fail and the game will continue on the next attempt
97    check_winner(PlayerOneBuilding, PlayerTwoBuilding) :-
98        check_player_two_win(PlayerOneBuilding, PlayerTwoBuilding),
99  +     format('Player 2 wins~n'), !.
100
101 + % If neither player has won, the game will continue
102   check_winner(_PlayerOneBuilding, _PlayerTwoBuilding) :- !.
103
104 +
105 + %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Check player win
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
106 +
107 +
108 + %! check_player_one_win(+PlayerOneBuilding, +PlayerTwoBuilding)
109 + % Check if player one has won by extinguishing their own building
110 + check_player_one_win(PlayerOneBuilding, _PlayerTwoBuilding) :-
111       building_extinguished(PlayerOneBuilding).
112
113 + % Check if player one has won by burning down their opponents building
114 + check_player_one_win(_PlayerOneBuilding, PlayerTwoBuilding) :-
115       building_in_flames(PlayerTwoBuilding).
116
117 + %! check_player_two_win(+PlayerOneBuilding, +PlayerTwoBuilding)
118 + % Check if player two has won by extinguishing their own building
119 + check_player_two_win(_PlayerOneBuilding, PlayerTwoBuilding) :-
120       building_extinguished(PlayerTwoBuilding).
121
122 + % Check if player two has won by burning down their opponents building
123 + check_player_two_win(PlayerOneBuilding, _PlayerTwoBuilding) :-
124       building_in_flames(PlayerOneBuilding).
125
126 +
127 + %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Building put out or on fire
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
128 +
129 +
130 + %! is_room_water(+buildingArray)
131   % Check if building is all 1's (water)
132 + is_room_water([]).                         % Base case
133 + is_room_water([1 | Rest]) :-               % If the current room is 1 (water)
134 +     is_room_water(Rest).                    % Loop back to check the rest of the building
135 +
136 + % Check if a matrix contains all ones
137 + building_extinguished([]).                 % Base case
138 + building_extinguished([Row | Rest]) :-     % Separate the matrix (building) into the current
      row and the rest of the matrix
139 +     is_room_water(Row),                     % Check if the current row is all 1's (water)
140 +     building_extinguished(Rest).            % Loop back to check the rest of the building
141
142 + %! is_room_fire(+buildingArray)
143   % Check if building is all 0's (fire)
144 + is_room_fire([]).                          % Same as above but for 0's (fire)
145 + is_room_fire([0 | Rest]) :-
146 +     is_room_fire(Rest).
147 +
148 + % Check if a matrix contains all ones
149   building_in_flames([]).
150 + building_in_flames([Row | Rest]) :-
151 +     is_room_fire(Row),
152 +     building_in_flames(Rest).
153 +
154 +
155 + %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Take turn
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
156 + %! take_turn(+PlayerOneBuilding, +PlayerTwoBuilding)
157 +
```

Line 91-102: `check_winner` takes in the two building arrays. The first version tests for Player 1 winning. The second version for Player 2. Third version if neither win. If line 92 fails, P1 didn't win so the program moves onto the next attempt of check_winner. It fails on P1 win so it tries for P2 win as Prolog wants to be true. If P2 win fails, it runs the line 102 where it cuts out of the predicate which is true, so the game will continue. Without the fail case on 102 and with no winner, `check_winner` would fail and the program would halt. The two parameters here are 'grunted' (_) aka underscored, as Prolog ignores underscores and neither building array is used here.

Line 110-124: Check P1 win by seeing if all elements in P1's building array is 1. Check P1 win by seeing if all elements in P2's building array is 0. And likewise for P2.

Line 132-152: Loop with head tail again. This time with naming schemes with 'Row' and 'Rest'. The predicate 'building_extinguished' get a row from the given array, then calls `is_room_water` which does the same functionality, but checks if every element in the row is a 1. `building_extinguished` checks the next row if true. Again, same functionality for fire, but checks for 0.

Testing in SWI Prolog.

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- consult('/Users/richardaustin/Documents/GitHub/Uni3_ProgLang_Paradigms_Project/20032144/Prolog/testing.pl').

YOUR BUILDING IS ON FIRE
And so is your opponents...
You hate your opponet...
You can save your building... or destroy your opponents...
1 = Water
0 = Fire
Press enter to start
|:

Player One Building:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

Player Two Building:
1 0 0 0 1
0 1 0 1 0
0 1 0 1 0
0 0 1 0 0
Press enter to continue
|:

These are the building numbers

 16.  17.  18.  19.  20.
 11.  12.  13.  14.  15.
 6.   7.   8.   9.   10.
 1.   2.   3.   4.   5.

Press enter to continue
|:

Lets begin
Player 2 wins

Player One Building:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

Player Two Building:
1 0 0 0 1
0 1 0 1 0
0 1 0 1 0
0 0 1 0 0

Ayup
true.

?-
```

Loading it into Prolog and it works! Pressing the enter key when asked shows the next section, the building visually shows up and check winner seemingly works.

Commit 3 - Take Turn & Start Spread

```
+     format('If you stack two 1s on top of each other, then the water will flow down to all rooms directly below\n'),
+     format('If you stack two 0s on top of each other, then the fire will flow up to all rooms directly above\n'),
+     format('Press enter to continue~n'),
+     get_char(_),
```

Minor addition comes by the way of extra game rule information for the players.

```
+     check_winner(PlayerOneBuilding, PlayerTwoBuilding), !,
+     whos_turn(Player),
+     take_turn(Player, PlayerOneBuilding, PlayerTwoBuilding),
      format('~nAyup~n').

+ game_loop :-
+     game_loop.
+
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %! check_winner(+PlayerOneBuilding, +PlayerTwoBuilding)
  % Check if player one has won
  % If fail, check_winner will run the next attempt below, for p2
  check_winner(PlayerOneBuilding, PlayerTwoBuilding) :-
      check_player_one_win(PlayerOneBuilding, PlayerTwoBuilding),
+     format('Player 1 wins~n'),
+     halt.

  % Check if player two has won
  % If fail, check_winner will fail and the game will continue on
  check_winner(PlayerOneBuilding, PlayerTwoBuilding) :-
      check_player_two_win(PlayerOneBuilding, PlayerTwoBuilding),
+     format('Player 2 wins~n'),
+     halt.
```

I added in the `check_winner`, `whos_turn`, and `take_turn` calls into the game loop while also adding the addition of the `halt` when a player wins.

```
+ :- dynamic(current_turn/1).
+ current_turn(1).
```

At the very top, I created a dynamic predicate for `current_turn`. Prolog variables are not mutable so defining this, allows me to modify the clause of current_turn, beginning with 1.

```
170 + whos_turn(Player) :-
171 +     current_turn(Player),
172 +     switch_turn.
173 +
174 + %! switch_turn
175 + switch_turn :-
176 +     retract(current_turn(Player)),
177 +     NextPlayer is 3 - Player,
178 +     assert(current_turn(NextPlayer)).
179 +
```

Get the current player turn from memory. Switch the player by retracting the current value from memory (get the value but remove it). I change the player by doing 3 - the player number. 3 - 1 = 2 and 3 - 2 = 1. A constant loop between 1 and 2 this way is an elegant solution. Once player number has switched, store the new player number in memory.

```
181 + %! take_turn(+Player, +PlayerOneBuilding, +PlayerTwoBuilding)
182 + take_turn(Player, PlayerOneBuilding, PlayerTwoBuilding) :-
183 +     format('~nIts player ~ws turn~n', [Player]),
184 +     % chose_fire_or_water(FireOrWater),
185 +     format('~nPress 1 for water or 0 for fire: ~n'),
186 +     get_char(FireOrWaterChar),
187 +     atom_number(FireOrWaterChar, FireOrWater),
188 +     trace,
189 +     format('Which room do you chose (1-20): '),
190 +     get_char(RoomNumberChar),
191 +     atom_number(RoomNumberChar, RoomNumber),
192 +     update_building(RoomNumber, Row, Col),
193 +     player_turn(Player, PlayerOneBuilding, PlayerTwoBuilding, FireOrWater, Row, Col),
194 +     display_both_buildings.
195 +
196 + %! chose_fire_or_water(-FireOrWater)
197 + chose_fire_or_water(FireOrWater) :-
198 +     format('~nPress 1 for water or 0 for fire: ~n'),
199 +     get_char(FireOrWaterChar),                 % Get the char the player inputs (other ways
      bugger it up on Mac for some reason)
200 +     atom_number(FireOrWaterChar, FireOrWater),  % Convert the char to an int
201 +     check_one_or_zero(FireOrWater).
202 +
203 + % If player doesnt enter 1 or 0, the first chose_fire_or_water will fail and will try again here
204 + % Which tells the player they inputted wrong and loops back to the first chose_fire_or_water
205 + chose_fire_or_water(FireOrWater) :-
206 +     format('~nIncorrect input~n'),
207 +     chose_fire_or_water(FireOrWater).
208 +
209 + %! check_one_or_zero(+FireOrWater)
210 + % Check if player inputs 1 or 0 on the attempt below
211 + % If fail, the first instant of chose_fire_or_water will run the fail case above
212 + check_one_or_zero(1) :- !.
213 +
214 + % Check if player inputs 0
215 + check_one_or_zero(0) :- !.
216 +
217 + %! update_building(+RoomNumber, -Row, -Col)
218 + % Get the row and column of the room number
219 + update_building(RoomNumber, Row, Col) :-
220 +     Row is 3 - (RoomNumber - 1) // 5,
221 +     Col is (RoomNumber - 1) mod 5.
222 +
223 + %! player_turn(+Player, +PlayerOneBuilding, +PlayerTwoBuilding, +FireOrWater, +Row, +Col)
224 + player_turn(1, PlayerOneBuilding, _, 1, Row, Col) :-
225 +     spread_water(PlayerOneBuilding, Row, Col, NewPlayerOneBuilding),
226 +     PlayerOneBuilding = NewPlayerOneBuilding.
227 +
228 + player_turn(1, _, PlayerTwoBuilding, 0, Row, Col) :-
229 +     spread_fire(PlayerTwoBuilding, Row, Col, NewPlayerTwoBuilding),
230 +     PlayerTwoBuilding = NewPlayerTwoBuilding.
231 +
232 + player_turn(2, _, PlayerTwoBuilding, 1, Row, Col) :-
233 +     spread_water(PlayerTwoBuilding, Row, Col, NewPlayerTwoBuilding),
234 +     PlayerTwoBuilding = NewPlayerTwoBuilding.
235 +
236 + player_turn(2, PlayerOneBuilding, _, 0, Row, Col) :-
237 +     spread_fire(PlayerOneBuilding, Row, Col, NewPlayerOneBuilding),
238 +     PlayerOneBuilding = NewPlayerOneBuilding.
239 +
```

Line 183: Format is used here with the variable of Player. This is why I use format, for the extra functionality it provides.

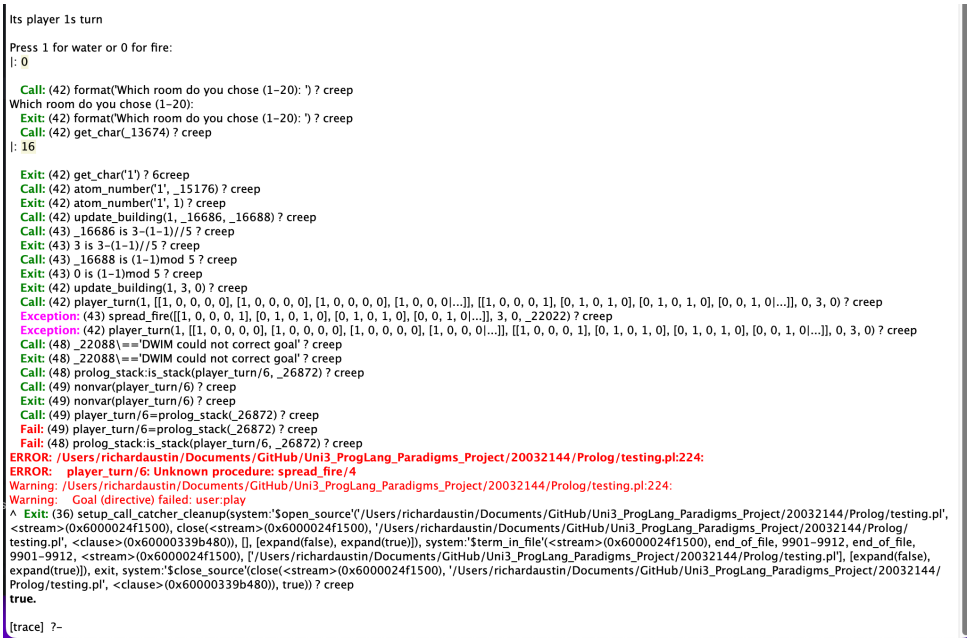
Line 186-191: A messy constraint, again due to macOS issues, I used `get_char` to get the inputs from a player where I then used the built in predicate `atom_number` that simply takes the atom value as a character and coverts it to a number.

Line 224-238: This is known as ‘rule based programming’ with the implementation of ‘pattern matching’ via four versions of `player_turn.`. Instead of multiple if statements (if Player == 1 and FireOrWater == 1, four times) I created each version of `player_turn` for it’s purpose. Player 1 save their building. Player 1 destroy P2. Etc. Very clean and easy to read code thanks to the use of the logic paradigm. In these predicates, I made the mistake of trying to set a variable to a new value but I fix this later.

Testing in SWI Prolog.

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- consult('/Users/richardaustin/Documents/GitHub/Uni3_ProgLang_Paradigms_Project/20032144/Prolog/testing.pl').

YOUR BUILDING IS ON FIRE
And so is your opponents...
You hate your opponet...
You can save your building... or destroy your opponents...
1 = Water
0 = Fire
Press enter to numbers
|:
If you stack two 1s on top of each other, then the water will flow down to all rooms directly below
If you stack two 0s on top of each other, then the fire will flow up to all rooms directly above
Press enter to continue
|:

Player One Building:
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0

Player Two Building:
1 0 0 0 1
0 1 0 1 0
0 1 0 1 0
0 0 1 0 0
Press enter to continue
|:

These are the building numbers

 16.  17.  18.  19.  20.
 11.  12.  13.  14.  15.
 6.   7.   8.   9.   10.
 1.   2.   3.   4.   5.

Press enter to start
|:

Lets begin

Player One Building:
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0

Player Two Building:
1 0 0 0 1
0 1 0 1 0
0 1 0 1 0
0 0 1 0 0

Its player 1s turn

Press 1 for water or 0 for fire:
|: 0
```

This time I knew functionality wasn't implemented yet but I had made progress I wanted to test. Game starts and asks player to chose fire or water (0 or 1). At this point, I added in a `trace` to go through the code step by step.



I stepped through the logic and entered the room I chose. As seen in the above screenshot, it fails on spreading fire as this feature is not yet implemented. A success regardless to get to this stage so I committed and moved onto the spreading functionality.

Commit 4 - Spread Water & Fire

```
+ %! spread_water(+Building, +Row, +Col, -NewBuilding)
+ % If on the bottom row, this predicate will run and replace only the room they chose
+ spread_water(Building, Row, Col, NewBuilding) :-
+     check_already_on_bottom_row(Row),
+     replace([Row, Col], Building, 1, NewBuilding).
+
+ % Spread water to the room below recursively if not on the bottom row
+ spread_water(Building, Row, Col, NewBuilding) :-
+     not(check_already_on_bottom_row(Row)),                % Check if we're not on the bottom row
+     replace([Row, Col], Building, 1, TempBuilding), % Spread water to the current room
+     NextRow is Row + 1,
+     spread_water(TempBuilding, NextRow, Col, NewBuilding).
+
+
+ %! check_already_on_bottom_row(+Row)
+ % Check if the room is on the bottom row
+ % Cant spread water down if already on the bottom row
+ check_already_on_bottom_row(Row) :-
+     Row = 3.
+
+ %! replace(+[RowIndex, ColIndex], +Building, +ReplaceWith, -NewBuilding)
+ replace([RowIndex, ColIndex], Building, ReplaceWith, NewBuilding) :-
+     replace_in_row(RowIndex, ColIndex, Building, ReplaceWith, NewBuilding).
+
+ %! replace_in_row(+RowIndex, +ColIndex, +Building, +ReplaceWith, -NewBuilding)
+ replace_in_row(0, ColIndex, [Row|Rest], ReplaceWith, [NewRow|Rest]) :-
+     replace_in_col(ColIndex, Row, ReplaceWith, NewRow).
+
+ replace_in_row(RowIndex, ColIndex, [Row|Rest], ReplaceWith, [Row|NewRest]) :-
+     RowIndex > 0,
+     NextRowIndex is RowIndex - 1,
+     replace_in_row(NextRowIndex, ColIndex, Rest, ReplaceWith, NewRest).
+
+ %! replace_in_col(+ColIndex, +Row, +ReplaceWith, -NewRow)
+ replace_in_col(0, [_|Rest], ReplaceWith, [ReplaceWith|Rest]).
+
+ replace_in_col(ColIndex, [X|Rest], ReplaceWith, [X|NewRest]) :-
+     ColIndex > 0,
+     NextColIndex is ColIndex - 1,
+     replace_in_col(NextColIndex, Rest, ReplaceWith, NewRest).
+
+
+ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Spread fire
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
+
+ %! spread_fire(+Building, +Row, +Col, -NewBuilding)
+ % If on the top row, this predicate will run and replace only the room they chose
+ spread_fire(Building, Row, Col, NewBuilding) :-
+     check_already_on_top_row(Row),
+     replace([Row, Col], Building, 0, NewBuilding).
+
+ % Spread fire to the room above recursively if not on the top row
+ spread_fire(Building, Row, Col, NewBuilding) :-
+     not(check_already_on_top_row(Row)),
+     replace([Row, Col], Building, 0, TempBuilding),
+     NextRow is Row - 1,
+     spread_fire(TempBuilding, NextRow, Col, NewBuilding).
+
+ %! check_already_on_top_row(+Row)
+ check_already_on_top_row(Row) :-
+     Row = 0.
```

The functionality of this section is to check spread water down a column or fire fire up.

Line 245-261: spread_water's first case checks if we're on the bottom row. If the player selected room 1-5, that room is on the bottom. You can't spread water down to any rooms below if you're on the bottom floor. Therefore, just change that room from 0 to 1. The second case is not on the bottom, hence replace every element directly below. As the building needs to be updated, I created the variable, 'TempBuilding' to temporarily store the updated building. Finally, recursively spread water with the next row.

Line 264-282: `replace` calls `replace_in_row`, which subsequently calls `replace_in_col`.

'replace'

substitutes the room at the row and column calculated in the `update_building` predicate with a 0 or 1 (fire or water).

'replace_in_row'

uses that row and column,

if the row index is greater than 0 (not at the top),

decrement the row index (move to the next row),

loop back to replace the room in the next row

'replace_in_col' does the same functionality as the prior, but for columns.

The logic in this section of code is called 'pattern matching'. It matches whether the row/col is 0 or not.

Line 289-302: Again, the same logic as the water functionality in spread_water, but checking the top and replacing with zeros.

At this stage, all that is left is to: check if a water or fire is above or below before spreading, display the updated the player buildings, and test fully.

Testing in SWI Prolog.

Commit 5 - Full Functionality Working

Although I implemented similar functionality earlier, I struggled to figure out how to update the building arrays so the game would continue with those new arrays.

```
1  + :- dynamic(current_turn/1).
2  + :- dynamic(buildingOne/1).
3  + :- dynamic(buildingTwo/1).
```

Adding to the previously defined dynamic variable, I realised I should do the same for the building arrays.

Now I had the spark to use dynamic variables, I can assert the buildings to memory and remove when best.

```
220 + player_turn(1, 0, _, BuildingTwo, Row, Col) :-
221 +     spread_fire(BuildingTwo, Row, Col, NewBuildingTwo),
222 +     retract(buildingTwo(BuildingTwo)),
223 +     assert(buildingTwo(NewBuildingTwo)).
224 +
225 + % Player 1 chose to spread water
226 + player_turn(1, 1, BuildingOne, _, Row, Col) :-
227 +     spread_water(BuildingOne, Row, Col, NewBuildingOne),
228 +     retract(buildingOne(BuildingOne)),
229 +     assert(buildingOne(NewBuildingOne)).
230 +
231 + % Player 2 chose to spread fire
232 + player_turn(2, 0, BuildingOne, _, Row, Col) :-
233 +     spread_fire(BuildingOne, Row, Col, NewBuildingOne),
234 +     retract(buildingOne(BuildingOne)),
235 +     assert(buildingOne(NewBuildingOne)).
236 +
237 + % Player 2 chose to spread water
238 + player_turn(2, 1, _, BuildingTwo, Row, Col) :-
239 +     spread_water(BuildingTwo, Row, Col, NewBuildingTwo),
240 +     retract(buildingTwo(BuildingTwo)),
241 +     assert(buildingTwo(NewBuildingTwo)).
```

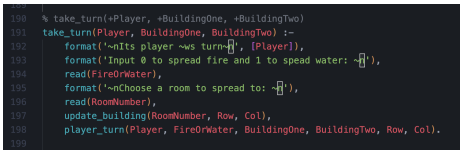
Recalling my earlier mistake of setting the player building to = the new building, the solution is now obvious. Remove the building currently in memory and then assert the updated version. Now in every `player_turn`, I do exactly that. But I can't retract anything if there isn't a building array in memory.

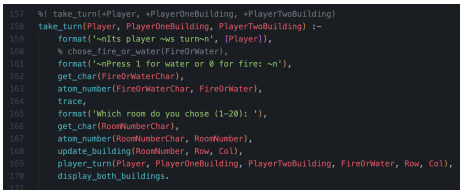
```
 89 + game_loop :-
 90 +     do_we_have_buidlings(BuildingOne, BuildingTwo),
 91 +     display_both_buildings(BuildingOne, BuildingTwo),
 92 +     check_winner(BuildingOne, BuildingTwo), !,
 93 +     whos_turn(Player),
 94 +     take_turn(Player, BuildingOne, BuildingTwo),
 95 +     game_loop.
 96 +
 97 + % do_we_have_buidlings(+BuildingOne, +BuildingTwo)
 98 + do_we_have_buidlings(NewBuildingOne, NewBuildingTwo) :-
 99 +     clause(buildingOne(NewBuildingOne), true),
100 +     clause(buildingTwo(NewBuildingTwo), true).
101 +
102 + do_we_have_buidlings(BuildingOne, BuildingTwo) :-
103 +     assert(buildingOne(BuildingOne)),
104 +     assert(buildingTwo(BuildingTwo)).
```

Therefore I created the `do_we_have_buidlings` predicate.

Line 98-104: The idea is to check if we already have the new, updated buildings in memory, from `player_turn`. At the start of the game, we do not and so the second version (line 102-104) will assert the starting building arrays to memory. If we do have new building arrays (the game is on at least it's second loop), the first test will succeed.

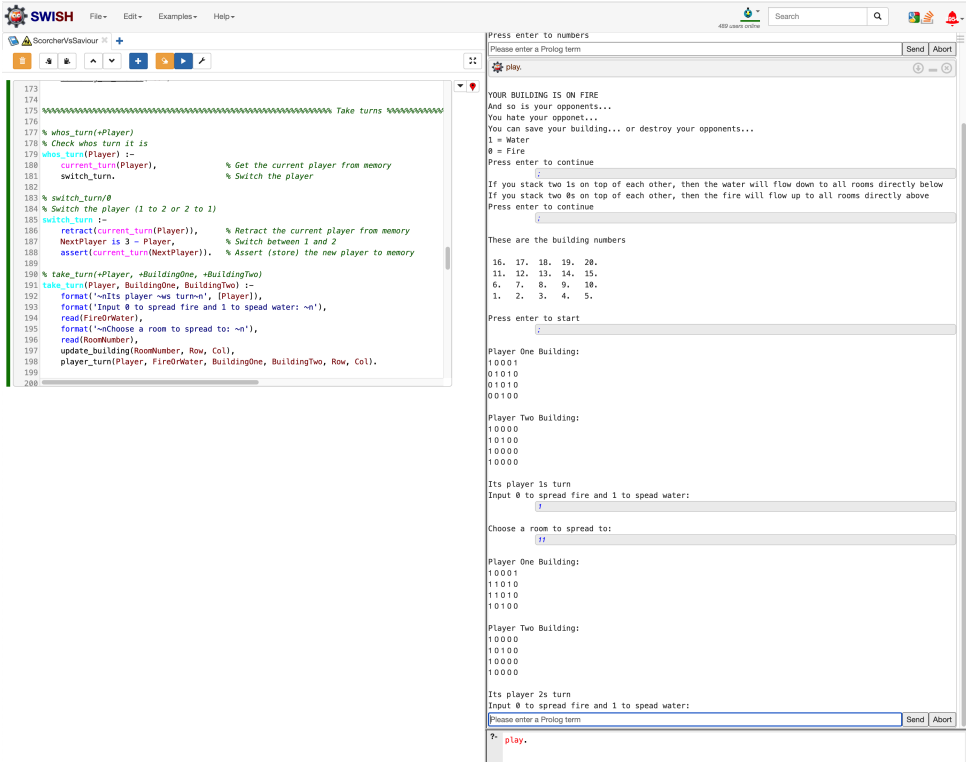
At this stage, the game had become more complex and the issues with Prolog and Mac became a hinderance. Due to this, I started testing with the online Prolog compiler. The complexity of `get_char` and `atom_number` could now be removed and replaced with the far simpler `read/1`. The only downside being that the user now has to enter something on the keyboard to run, they can't just press enter, but overall, it's an improvement.





A comparison of `take_turn/3` shows the considerably cleaner implementation of getting user input, on the left vs the right.

Testing in SWI Prolog.



Testing, now in the online compiler. Typing a semicolon to continue, I selected 1 to save my building with room 11 as choosing 11 should start a flow of water down, and it does. Now this doesn't feature the correct functionality whereby spreading only occurs if the same number is above or below it (two stacked), so that'll be the next implementation.

Commit 6 - Game Complete

The game is almost complete but with one key exception -only spread fire or water if the room above or below is the same value.

```
302 + % room_above_below(+Building, +Row, +Col, -Value)
303 + % Check if the room below is the chosen value (1 or 0)
304 + room_above_below(Building, Row, Col, Value) :-
305 +     NextRow is Row + 1,
306 +     nth0(NextRow, Building, RowList),
307 +     nth0(Col, RowList, Value), !.
308 +
309 + % Chcek if the room above is the chosen value (1 or 0)
310 + room_above_below(Building, Row, Col, Value) :-
311 +     NextRow is Row - 1,
312 +     nth0(NextRow, Building, RowList),
313 +     nth0(Col, RowList, Value), !.
314 +
```

In `room_above_below` I used the built in predicate `nth0/3` which retrieves the Nth element from a list, where N is the zero-based index of the element you want to access. In this case, the next row with the given column (floor in a building). I then call this predicate in `spread_fire` and `spread_water`.

Testing in SWI Prolog.

```
Press enter to start
          ;

Player One Building:
1 0 0 0 1
0 1 0 1 0
0 1 0 1 0
0 0 1 0 0

Player Two Building:
1 0 0 0 0
1 0 1 0 0
1 0 0 0 0
1 0 0 0 0
```

Starting arrays.

```
Its player 1s turn
Input 0 to spread fire and 1 to spead water:
1

Choose a room to spread to:
11

Player One Building:
1 0 0 0 1
1 1 0 1 0
1 1 0 1 0
1 0 1 0 0

Player Two Building:
1 0 0 0 0
1 0 1 0 0
1 0 0 0 0
1 0 0 0 0
```

```
Its player 2s turn
Input 0 to spread fire and 1 to spead water:
0

Choose a room to spread to:
7

Player One Building:
1 0 0 0 1
1 0 0 1 0
1 0 0 1 0
1 0 1 0 0

Player Two Building:
1 0 0 0 0
1 0 1 0 0
1 0 0 0 0
1 0 0 0 0
```

```
Its player 1s turn
Input 0 to spread fire and 1 to spead water:
0

Choose a room to spread to:
1

Player One Building:
1 0 0 0 1
1 0 0 1 0
1 0 0 1 0
1 0 1 0 1

Player Two Building:
1 0 0 0 0
1 0 1 0 0
1 0 0 0 0
0 0 0 0 0
```

Here I tested with P1, water in 11. Two one's stacked. It spread to all below. Success.

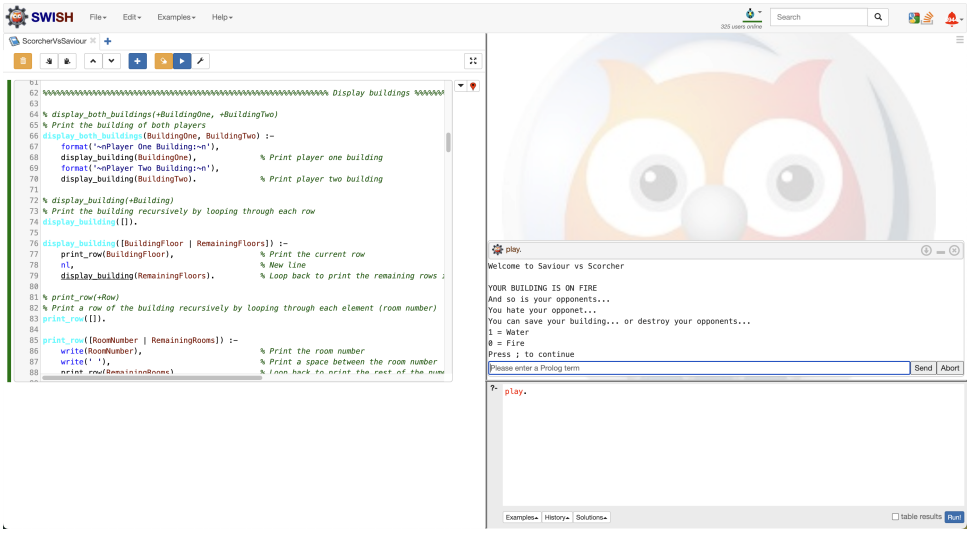
P2 spread fire to P1's 7. Two 0's stacked, it changes room room 12 to a 0. Success.

P1 spread fire to P2's 1. Only changes that room as there isn't a 0 above. Success.

Working Game

Upon completion, I tested the game fully with a friend. I test and play it in online SWI Prolog compiler as Prolog and macOS seem to have some issues.

Game Running



On the online compiler, I typed 'play.' in the console and the game started.

Player One Win Water

Here Player 1 has won by putting out their building as seen by all 1's in their matrix.

```
Its player 1s turn
Input 0 to spread fire and 1 to spead water:
          1

Choose a room to spread to:
          19

Player One Building:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1

Player Two Building:
1 0 0 0 0
1 1 0 1 0
1 1 0 0 0
1 1 0 0 1
Player 1 wins
** Execution aborted **
```

Player One Win Fire

Here Player 1 has won by enflaming Player 2's building as seen by all 0's in P2's matrix.

```
Its player 1s turn
Input 0 to spread fire and 1 to spead water:
          0

Choose a room to spread to:
          1

Player One Building:
0 1 0 0 1
0 0 0 1 0
1 0 0 1 0
1 0 0 1 0

Player Two Building:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Player 1 wins
** Execution aborted **
```

Player Two Win Water

Here Player 2 has won by putting out their building as seen by all 1's in their matrix.

```
Its player 2s turn
Input 0 to spread fire and 1 to spead water:
          1

Choose a room to spread to:
          1

Player One Building:
1 1 0 0 1
1 0 0 0 1
1 0 1 0 1
1 0 1 0 1

Player Two Building:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Player 2 wins
** Execution aborted **
```

Player Two Win Fire

Here Player 2 has won by enflaming Player 1's building as seen by all 0's in P1's matrix.

```
Its player 2s turn
Input 0 to spread fire and 1 to spead water:
          0

Choose a room to spread to:
          1

Player One Building:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

Player Two Building:
1 1 1 0 1
1 0 0 0 0
1 0 0 1 0
0 0 0 1 0
Player 2 wins
** Execution aborted **
```

Conclusion

Prolog is a language unlike many other. It is a language I have prior experience with and that is clearly shown in my final code. I displayed excellent Prolog, logical paradigm practices with recursion, storing and retracting variables, and correct documentation. That last part I am particularly proud of, while it doesn't affect the running program, it was paramount to returning to the code to develop further and overall code readability -something that is helpful for such language. I enjoyed returning to writing code in this logical way as it just makes sense. I have displayed how failing predicates try again on a next attempt -I think this way of programming is great. Whilst I had some turmoil with Prolog and Mac working together, the game ended up working exactly has intended via an online compiler. With more time, I would refactor and trim down some predicates but overall, it's an elegant solution that is easy to read.