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.

# Pre-implementation

## Layout Pseudocode

My high-level implementation goes as follows:

**Introduction.**

Explain the game.

Display building numbers.

Get user input to continue (break up sections of intro so it’s not overwhelming).

**Recursive game loop**

Set up building arrays.

Display the buildings.

Print each row in the building matrixes.

Check for winner.

P1 win if P1 building is all 1s or P2 is all 0s - Head tail every room.

P2 win if P2 building is all 1s or P1 is all 0s - Head tail every room.

Take turn.

Recursively lop back to game loop

**Take turn**

Get player input for 1 or 0 (water or fire).

Get player input for 1-20 (room number).

Get the row and column needed to update

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].

Spread water or fire

**Spread water or fire**

If player = 1 and water, spread water on P1 building, update P1 building.

If player = 1 and fire, spread fire on P2 building, update P1 building.

If player = 2 and water, spread water on P2 building, update P1 building.

If player = 2 and fire, spread fire on P1 building, update P1 building.

**Recursively spread water**

Check if we’re not on the bottom row.

If on bottom are, fail and move into new predicate to just change that element to a 1.

If not, check if there is a 1 above or below, if so:

Recursively

Go through each row below.

Replace the element in the same row with a 1.

Update building array by storing in a new building array and set that to be the corresponding building.

**Recursively spread fire**

Same logic as water but zeros and top row.

# Implementation

## Commit 1 - Intro & Display Buildings

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

**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 but the second parameter is optional for this command.

**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.

## Commit 2 - Check Winner

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

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.

**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’ (\_) as Prolog ignores \_ 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.**

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

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

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. '

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.

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.

**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:** Four versions of ‘player\_turn’. This is a good showcase of Prolog. 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. In these predicates, I made the mistake of trying to set a variable to a new value but I fix this later.

## Commit 4 - Spread Water & Fire

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.

**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.

## 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.

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.

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.

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.

## 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.

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.

# 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.

## 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.

## Player Two Win Water

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

## 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.