CS 410/510 Advanced Functional Programming

Assignment 3

Functional application development

Katie Casamento

Spring 2023
Portland State University

1 Introduction

This final assignment is an open-ended project based on the topics that we've been covering for the past several weeks. Your overall goal in this assignment is to implement a variant of the game of tic-tac-toe, using Haskell with the `brick` library that we saw in assignment 2.

2 Program requirements

These are the requirements for how your program should behave from the user's perspective.

2.1 Command-line arguments

Your program must support two command-line arguments, which the user should be able to pass in either order:

- `-s` to control the size of the board, which must be a positive (nonzero) integer
- `-p` to control which player goes first, which must be either `X` or `O`

The board should always be square, so for example `-s 15` indicates a 15×15 board. If the user does not explicitly pass an argument, the default argument values should be `-s 3` and `-p X`.

2.2 User interface

While the game is being played, the user interface should use a `brick` table view similar to the Minesweeper grid in assignment 2. An empty cell should show an empty space, and a cell with an `X` or `O` in it should show the corresponding letter.

On each player's turn, the user should be able to select a cell on the board with the arrow keys and the Enter key. Each player should alternate turns until the game has ended (one player has won or the game has ended in a tie).

If the user tries to select a cell that already contains an `X` or `O`, the input should just be ignored.

When the game ends, your program should exit the table view and print a message to the console saying `X wins`, `O wins`, or `tie game`.

2.3 Game rules

On an $n \times n$ board, there are $2n+2$ different ways for a player to win:

- n rows
- n columns
- 2 diagonals

Note that there are always only 2 diagonals regardless of how big the board is: for example, this board below does **not** count as a win for `O` because the line doesn't go from one corner to the opposite corner.

X	O			
	X	O		
			O	X
				O

In other words, to win on an $n \times n$ board, a player always needs a line of exactly n cells.

The game should end immediately after a player makes a winning move. If neither player wins, the game should end with a tie after the last empty cell is filled with an `X` or `O`.

3 Code requirements

These are the requirements for how you should write the code of your program.

3.1 Academic honesty requirements

Breaking these requirements will count as an academic honesty violation.

- **You must not submit any shared code that another student is also submitting for this assignment**. You are encouraged to **discuss the concepts** of the assignment and **share sample code that you do not intend to submit**, but you **must not share code that you are going to submit**.

- If you take any code from the internet or any other source, you must **cite where it came from** and **comply with its license**. If the code doesn't come with a license, by default that means the code cannot be legally copied or shared.

3.2 Grade requirements

Breaking these requirements will cause deductions to your assignment grade.

- Your code must compile and run with at least one GHC version that is 9.2.7 or greater. (You don't need to test for compatibility with multiple GHC versions, just make sure you're not using anything older than 9.2.7.)
- You must not write any calls to the functions `head`, `tail`, or `fromJust`, or the `!` or `!!` operators, or any other functions that have any possibility of a runtime crash.
- You must not write any calls to the `error` function or any uses of `undefined`.
- You must not write any module imports that include the word `Unsafe`.
- All of the code that you write must run in finite time for all possible inputs. You will not be graded on performance, but your code must not go into any infinite recursion.

Apart from these requirements, you have total freedom in how you write your code: you may import any packages, set any compiler flags, and use any code style.

4 Getting started

I recommend starting with a copy of the assignment 2 codebase and modifying it for the needs of this project. (The assignment 2 license allows you to copy it!)

I also recommend deeply reviewing the provided assignment 2 code as we work through the topic of monads in lecture. Focus in particular on the code that uses `IO` and `EventM`; the `State` code will be mostly irrelevant to this project.

The `Prelude` documentation explains everything that's imported automatically in a Haskell module, and you can find the documentation for all other built-in modules in the `base` module listing. You can also find documentation on Hackage for any third-party packages that you use, including `brick`.

I will give a lot of advice for this project over the next several lectures, so make sure to keep up and review the videos when you forget things!