Tic Tac Toe: Data and Architecture Design

Curry Lambda, 1234321

September 25, 2019

1 Introduction and Rules

This document describes the implementation we have already made of Tic-Tac-Toe, whilst concluding assignment number 2. Tic-Tac-Toe is a turn-based two-player puzzle game, traditionally played on a two-dimensional 3×3 grid.

The active player alternate every round, where it places one marker on the board. The game is finished when either player is able to make a continuous horizontal, vertical or diagonal line on the 3×3 grid.

2 Design

2.1 Data Structures

2.1.1 The Game State

The full state of our game is represented using a GameState data type:

```
data GameState = GameState { board   :: Board
                           , player2 :: PlayerType
                           ,  There are probably morefields here in
                               an actual design document
                           }
```

Player 1 will always be controlled by a human.

2.1.2 The Player and Playing field

The players and the state of each field of the grid will be represented by an enumeration type:

```
data Player = P1 | P2
data Field  = X | O | B
```

A board configuration will be represented by a nested 3-tuple.

```
type Row   = (Field, Field, Field)
type Board = (Row, Row, Row)
```

2.1.3 Player Movement

Tic Tac Toe is a game that has no independently moving particles. Moreover, movement is discrete: The player selects a move, and it appears on the screen. There is no smooth movement like we would have in something like `Pacman`.

The possible moves a player p can play a board b will be enumerated; this enables this function to be reused for computer players, Section 2.2.

```
moves :: Player -> Board -> [Board]
```

Hence, we choose not to represent player moves as first class objects. We use an index in the list of possible *next* boards for the active player as a *move*, whenever necessary.

2.2 Computer Adversary

Our implementation of Tic Tac Toe will support computer players.

```
data PlayerType = Human | Computer
```

When the game is being played in single-player mode, a rudimentary artificial inteligence takes place. We will use the `minimax` algorithm to explore the game tree, which is represented by a rose tree of `Board`s. After deciding which board is the next *best possible* mode, the computer will use `makeMove` and play that board.

```
data Rose a = a :> [Rose a]

minimax :: Player -> Rose Board -> Rose Int

makeMove :: Player -> Board -> Maybe Board
```

2.3 Interface

The players will interact with the game through a textual interface in the command line. Each board will be pretty printed on the screen with a function.

```
printBoard :: Board -> String
```

An example output would be:

```
X| |O
-+-+-
X|O|
-+-+-
O|X|
```

When the turn of a human player p comes up, the computer prints all possible moves on the screen, player p then types the number of the move they want to perform. This is easily doen with a polymorphic `askFor` function in Haskell.

```
askFor :: Show a => String -> [a] -> IO a
```

In order to easily display information on the screen, all the types involved will be instances of the `Show` typeclass. This enables the rendering framework to be resilient to future changes.

```
instance Show Player     ...
instance Show Field      ...
instance Show PlayerType ...
```

2.4 Implementation of the Minimum Requirements

Player The player can control one of the players in Tic-Tac-Toe.

Enemies The computer can control one of the players. It tries to win against the human player by using a minmax algorithm.

Randomness This example design document does not incorporate any randomness. You should make sure your design document does include randomness!

Animation This example design document does not incorporate anything on animation. You should make sure your design document does include animation!

Pause Since tic-tac toe is turn based, it can naturally be paused. I.e. no additional work is required here. Note that in your design document this will not be the case.

IO This example design document does not incorporate interaction with the file system. You should make sure your design document does include IO!

2.5 Implementation of the optional Requirements

We are not planning on incorporating any of the optional requirements.