Wesley Blount
PLS
Graduate Assignment

Tic Tac Toe in Haskell

Download

If you don't already have it, the best place to download Haskell from is their main site at https://www.haskell.org

Run Files

- "ghc - -make filename" produces an executable and " ./filename" runs it
- "runhaskell filename.hs" simply runs it in the command line

Overview

For this project I decided to try out a little basic game theory and create a tic tac toe bot using Haskell and the minimax optimization function. The program simply displays a board and prompts the user to select a move. The program responds by selecting an acceptable move and the process repeats until there is a winner, draw, or input error (I didn't add checks for valid input…)

Syntax Basics (just the 1 or 2 things that might need clarification)

Function decelerations for functions that take arguments:

- func2 :: [Char] -> Char -> Int -> Bool
- reads: func2 takes a list of type Char, a Char, an Int, and returns a Bool

Guards (|)

- Sort of like conditionals, but used only for returning.
- Blocks a return if it doesn't evaluate to true
- ex: isPositive takes and Int and returns a Bool. Guards check before allowing a return. *The '=' is not an assignment. It marks a return.*

```
isPositive :: Int -> Bool
isPositive x
    | x > -1 = True
    | otherwise False
```

(x:xs) - x is head of list, xs is tail of list. A colon cons's them together

Function Descriptions

Aside from the main function and a few helpers for printing stuff out, the program consists of 11 functions which are used together to do all the work. Haskell is known as a 'lazy' language, which means functions are not evaluated until they are used, so their ordering within the file does no matter. I have ordered them from higher level to lower level for organization within the actual document. Below are descriptions of a few of the more complex ones

Wesley Blount
PLS
Graduate Assignment

bestMove
This function cycles through each available move in the board and runs minmax on each move. It returns the index of the first move that returns a positive value (not a loss). While this will keep it from loosing, an improvement would be to have it check explicitly for moves that lead to wins first, then settle for one that leads to a draw if none are found. It takes the current board and returns a new board after choosing a move.

minmax
To make it 'smart' I read up on the minimax algorithm and came up with my own implementation for Haskell. Minimax works by recursively spanning out all possible games from the point it was called. When a branch reaches a final state (such as winner, looser, or draw), it passes that result back up the tree. Each level of the tree alternates between selecting the most desired outcomes (max) or selecting the least desired outcomes (min), just as alternating opponents will only select moves that do not benefit the other. The value that is returned to the caller will represent the best case scenario following a potential move, assuming the opponent plays optimally. This will inform the caller if they really want to make that move or not. For this program, X is the maximizer and seeks a 10, O is the minimizer and seeks -10, and a draw is 0.

While heuristic is the obvious base case, another is reached if the last cell examined is not an available move, thus minmax can't be called on it. Because it's time to recur back up the list, we need some value returned to use for our comparisons, which is where *returnBound* comes in. It returns the min or max Int possible based on if the caller is minimizer or maximizer.

heuristic
Returns the value of a given board state. If it finds a win from X it returns 10, and if it finds a win from O it returns -10. If there is no winner but the board is full, it returns 0 to signify a draw. Heuristic is used with final as a base case for minmax.