Functional Programming
Assignment 2: Data structures

Ruud Koot

In this exercise we will implement a simple game called *Butter, Cheese and Eggs* (also known as *Tic-Tac-Toe* or *Noughts-and-Crosses* on the other side of the sea). You may have played this game before, but if you're a little foggy on the rules then you can have a look at `https://en.wikipedia.org/wiki/Tic-tac-toe`.

1 Rose trees

A *rose tree* or *multi-way tree* is a tree data structure in which each node can store one value and each node can have an arbitrary number of children. Rose trees can be represented by the algebraic data type:

$$\mathbf{data}\ \mathit{Rose}\ \alpha = \mathit{MkRose}\ \alpha\ [\mathit{Rose}\ \alpha]$$

Exercise 1. *Write functions $root :: Rose\ \alpha \to \alpha$ and $children :: Rose\ \alpha \to [Rose\ \alpha]$ that return the value stored at the root of a rose tree, respectively the children of the root of a rose tree.*

Exercise 2. *Write functions $size :: Rose\ \alpha \to Int$ and $leaves :: Rose\ \alpha \to Int$ that count the number of nodes in a rose tree, respectively the number of leaves (nodes without any children).*

2 Game state

The *state* of a (turn-based board) game will generally consist of the player whose turn it is and what is currently on the board. The current player in a two-person game can be represented by the data type:

$$\mathbf{data}\ \mathit{Player} = \mathit{P1} \mid \mathit{P2}$$

Exercise 3. *Write a function $nextPlayer :: Player \to Player$ that given the player whose move it is currently, will return the player who will make a move during the next turn.*

The board in *Butter, Cheese and Eggs* consists of nine fields, each either containing a cross or a circle, or being blank:

$$\mathbf{data}\ \mathit{Field} = X \mid O \mid B$$

Exercise 4. *Write a function $symbol :: Player \to Field$ that gives the symbol a particular player uses. (By centuries-old tradition the first player always uses a cross.)*

A *row* consists of three horizontally, vertically or diagonally adjacent fields:

$$\mathbf{type}\ \mathit{Row} = (\mathit{Field}, \mathit{Field}, \mathit{Field})$$

We can then compose the board from three horizontal rows:

$$\mathbf{type}\ \mathit{Board} = (\mathit{Row}, \mathit{Row}, \mathit{Row})$$

Exercise 5. *This representation gives us easy access to the horizontal rows, but not to the vertical and diagonal ones. Write two functions verticals* :: *Board* $\rightarrow$ (*Row*, *Row*, *Row*) *and diagonals* :: *Board* $\rightarrow$ (*Row*, *Row*) *that do. Your implementation of diagonals should report the "topleft-to-bottomright" diagonal as the first element of the output tuple.*

Exercise 6. *Define a constant emptyBoard* :: *Board that represents the empty board.*

Exercise 7. *Write a function printBoard* :: *Board* $\rightarrow$ *String that nicely formats a board as a string. For example, printBoard someBoard should return the string* `"O| | \n-+-+-\n |X| \n-+-+-\n | | \n"`.

3 Game trees

A *game tree* is a rose tree where all the nodes represent game states and all the children of a node represent the valid moves than can be made from the state in the parent node.

Exercise 8. *Write a function moves* :: *Player* $\rightarrow$ *Board* $\rightarrow$ [*Board*] *that, given the current player and the current state of the board, returns all possible moves that player can make expressed as a list of resulting boards. (For now, you should continue making moves, even if one of the players has already won.). Here are some hints on how to write this function in a convenient and concise way:*

1. *Write a helper function traverseFst* :: $(\alpha \rightarrow [d]) \rightarrow (\alpha, b, c) \rightarrow [(d, b, c)]$. *There should only be one "reasonable" non-trivial implementation of a function that has this type. Write analogous functions traverseSnd and traverseThd (think about their types first).*
2. *Write a function traverseAll* :: $(\alpha \rightarrow [\alpha]) \rightarrow (\alpha, \alpha, \alpha) \rightarrow [(\alpha, \alpha, \alpha)]$ *that combines the above three functions.*
3. *Implement a function movesRow* :: *Player* $\rightarrow$ *Row* $\rightarrow$ [*Row*] *that, given a player and a row, returns all possible moves that the player can make in that row, using the function traverseAll.*

Exercise 9. *Write a function hasWinner* :: *Board* $\rightarrow$ *Maybe Player that, given a board, returns which player has won or Nothing if none of the players has won (either because the game is still in progress, or because it is a draw).*

Exercise 10. *Write a function gameTree* :: *Player* $\rightarrow$ *Board* $\rightarrow$ *Rose Board that computes the game tree. (Here you should make sure that you stop making moves once one of the players has won.)*

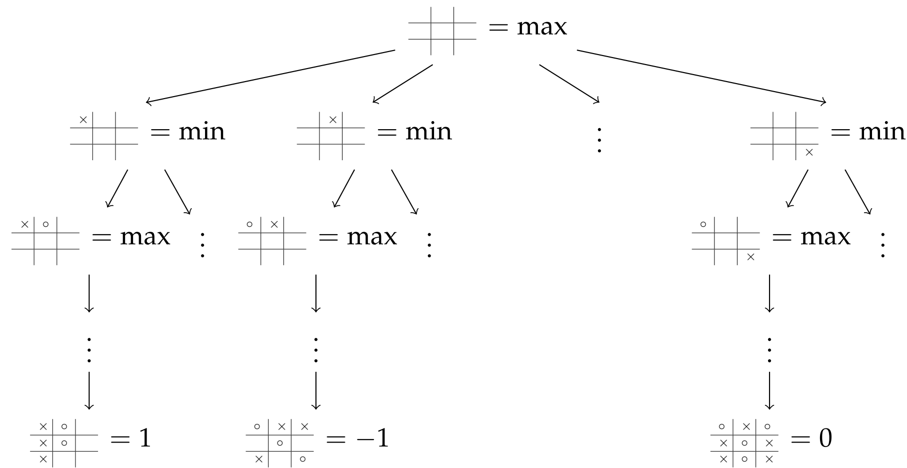
4 Game complexity

Game theorists have defined various measures of how hard a particular game is, called the *complexity* of a game. One of those measures is the *game tree complexity*, which is the number of leaves in the game tree.

Exercise 11. *Define a constant gameTreeComplexity* :: *Int that computes the game tree complexity of Butter, Cheese and Eggs.*

5 Minimax

We can use a game tree to implement an intelligent computer opponent (AI) for us to play against. This can be done using the *minimax* algorithm. The name of this algorithm stems from the fact that if we would assign a score to each leaf of the game tree (1 if we win, 0 if it's a draw, and -1 if we lose) then for each internal node where we make a move, we always make a move that *maximizes* our score (and take this as the score for the internal node), while for internal nodes where the opponent makes a move, we can assume they make a move that *minimizes* our score (and take this as the score for the internal node).



Exercise 12. *Write a function minimax* :: *Player* → *Rose Board* → *Rose Int that computes the minimax tree for a given player and game tree. Here are some hints:*

1. *You must treat leaves of the tree (nodes that do not have any children) differently from the internal nodes of the tree (nodes that do have children).*
2. *The first argument of minimax is the Player you are calculating the minimax tree for. This argument is kept constant throughout the whole computation. It is useful to introduce a helper function minimax′ that takes another Player argument. This is the player that is allowed to make a move, and alternates at every level of tree as you recurse through it.*

If you correctly and elegantly implemented the *minimax* function then its implementation should use the *minimum* and *maximum* functions. These functions find the minimum and maximum of arbitrary lists of numbers and will thus always have to look at every element in the lists. However, we know that the lists we encounter will only contain the numbers 1, 0 or -1. Thus if *minimum* (respectively *maximum*) encounters the element -1 (respectively 1) we already know what the optimum is going to be and do not have to continue looking any further.

Exercise 13. *Write lazier versions of minimum and maximum (and name them minimum′ and maximum′) that stop looking for a smaller, respectively larger, number in their input list once they encounter the optimum of* -1*, respectively* 1.

If you replace the calls to *minimum* and *maximum* with *minimum′* and *maximum′* in the *minimax* function, then the function will stop looking for an optimum once it has already found one. By "magic" of lazy evaluation the program will not only stop looking for a more optimal optimum that can never exist, it will also not bother generating whole parts of the minimax and game trees. This will make the program run several times faster.

Exercise 14. *Write a function makeMove* :: *Player* → *Board* → *Maybe Board that makes an optimal move (if it is still possible to make a move).*

6 Wrapping up

The starting framework contains a *main* function that—assuming you have implemented all of the above exercise correctly—allows you to play the game against a human or computer opponent. Of course—again assuming you have implemented all exercises correctly—you will never be able to beat the computer opponent.

Important note: When handing in the assignment make sure you name your module *Assignment2*, otherwise DOMjudge will not be able to compile your code. If you want to compile your program yourself, you will have to name the module *Main*, though. If you want to run you program from the interpreter, then the module name does not matter. Like most Haskell programs, the game will run much faster if compiled (with the `-O` flag) instead of interpreted.

7 Further research

Here are some suggestion for "further research", if you are finished early with the assignment and are feeling bored. You do not have to hand them in.

1. Generalize the game to k^n (i.e., n-dimensional Butter, Cheese and Eggs on a $k \times \cdots \times k$ board). It will be inconvenient to represent your board as nested tuples in this case. Instead, consider using an Array.
2. Use the same techniques to play other games such as Connect Four, Othello, Checkers, Chess, or Go; to solve 15 Puzzles or Rubik's Cubes; or to optimize Starcraft build orderings. For these games you are unlikely to be able to fully traverse the whole game tree to determine who wins, so you will probably want to use alpha–beta pruning and an evaluation function on a *partial game tree* instead of minimax on a complete game tree.
3. You can use the various rotational and reflective symmetries of the game board to reduce the size of the of the game tree by two orders of magnitude. Additionally, some sequences of moves will eventually result in the same game board, so it can be advantageous to represent the game as a directed acyclic graph instead of a tree. Use these techniques to speed up the AI even further.