Programming Languages

Reexam (Project 2) (deadline: Friday February 24th, 2017 at 23.59)

This project is mandatory for the completion of the DM552 course.

1 The 2048-game with parameters (w, h)

In this exercise, you are going to implement a version of the game **2048** which is played on a grid of size $w \times h$. Please try out the game at `http://gabrielecirulli.github.io/2048/` first, and then read the following description.

In the file `Game.hs` we have provided the following type class *Game*

$$
\begin{aligned}
&\textbf{type } \mathit{Player} = \mathit{Bool}\\
&\textbf{type } \mathit{Value} = \mathit{Double}\\
&\textbf{type } \mathit{Weight} = \mathit{Double}\\
&\textbf{class } (\mathit{Show}\ (\mathit{Move}\ g), \mathit{Show}\ (\mathit{GameState}\ g)) \Rightarrow \mathit{Game}\ g \textbf{ where}\\
&\quad \textbf{type } \mathit{Move}\ g\\
&\quad \textbf{type } \mathit{GameState}\ g\\
&\quad \mathit{startState} :: g \to \mathit{GameState}\ g\\
&\quad \mathit{value} :: g \to \mathit{Player} \to \mathit{GameState}\ g \to \mathit{Value}\\
&\quad \mathit{moves} :: g \to \mathit{Player} \to \mathit{GameState}\ g \to [(\mathit{Weight}, \mathit{Move}\ g)]\\
&\quad \mathit{move} :: g \to \mathit{Player} \to \mathit{GameState}\ g \to \mathit{Move}\ g \to \mathit{GameState}\ g\\
&\quad \mathit{showGame} :: g \to \mathit{GameState}\ g \to \mathit{String}
\end{aligned}
$$

Your task is to implement the game 2048 as an instance of the *Game* type class. As a starting point the file `Game2048.hs` contains the following instance, where you should fill out appropriate function bodies instead of $\bot$:

$$
\begin{aligned}
&\textbf{type } \mathit{Width} = \mathit{Int}\\
&\textbf{type } \mathit{Height} = \mathit{Int}\\
&\textbf{data } \mathit{Game2048} = \mathit{Game2048}\ \mathit{Width}\ \mathit{Height}\\
&\textbf{data } \mathit{Direction} = L \mid R \mid U \mid D \textbf{ deriving } \mathit{Show}\\
&\textbf{data } \mathit{Action} = \mathit{NewTile}\ (\mathit{Int}, \mathit{Int})\ \mathit{Int} \mid \mathit{Slide}\ \mathit{Direction} \textbf{ deriving } \mathit{Show}\\
&\textbf{instance } \mathit{Game}\ \mathit{Game2048} \textbf{ where}\\
&\quad \textbf{type } \mathit{GameState}\ \mathit{Game2048} = [[\mathit{Int}]]\\
&\quad \textbf{type } \mathit{Move}\ \mathit{Game2048} = \mathit{Action}\\
&\quad \mathit{showGame} = \bot\\
&\quad \mathit{startState} = \bot\\
&\quad \mathit{move} = \bot\\
&\quad \mathit{moves} = \bot\\
&\quad \mathit{value} = \bot
\end{aligned}
$$

- The original game 2048 alternates between actions performed by the computer, and actions performed by the human player. In the Haskell-implementation required in this assignment, the current player is represented by the type *Bool*, and we refer to the computer player as player *False* and the human player as player *True*.

- The game starts with an empty grid, and we are only interested in games where the computer (player *False*) makes the first move, since there are no available moves for the human otherwise. In Haskell, we represent the grid by nested lists of *Int*s, $[[Int]]$, where each sublist corresponds to rows of the grid. An empty position in the grid is represented by the value 0 and the non-empty positions are represented by positive integers.

- **The computer's move** is to pick an empty position, and write either the value 2 or 4 in this position. We represent this kind of move by the data constructor *NewTile* (x, y) v where (x, y) is a tuple of coordinates in the grid, and v is either 2 or 4.

- **The player's move** is to pick a *direction* - left, right, up or down - and all the tiles in the grid will "slide" in this direction. If the slide operation does not change the game state, it is not allowed as a move. We represent this kind of move by the data constructor *Slide* d where d is either L, R, U or D.

  We will describe in detail how the slide operation is defined:

  - When the move *Slide L* (slide left) is performed, the following happens to each *row* in the grid:
    1. First, all 0's are removed. Example: [2,0,2,2] becomes [2,2,2].
    2. All consecutive numbers which are equal are merged by adding them, from left to right. Example: [2,2,2] becomes [4,2]. [2,2,2,2] becomes [4,4]. [2,2,4] becomes [4,4].
    3. The row list is padded with zeros at the end such that it again is of length w. Example: [2,2] becomes [2,2,0,0] for $w = 4$.
  - *Slide R* is defined similarly, but note the differences:
    2. All consecutive numbers which are equal are merged by adding them, from **right to left**.
    3. The row list is padded with zeros **at the beginning** such that it again is of length w.
  - *Slide U* and *Slide D* (up and down) is defined analogously to *Slide L* and *Slide R*, with the difference that they are defined in terms of the columns of the grid, and not the rows of the grid.

```
1)                      2) Move: NewTile (1,1) 2   3) Move: Slide L          4) Move: NewTile (3,2) 2
   8   32   64  512        8   32   64  512           8   32   64  512          8   32   64  512
   -    -    2   32        -    2    2   32           4   32    -    -          4   32    -    -
   -    -    -    8        -    -    -    8           8    -    -    -          8    -    -    2
   -    -    -    2        -    -    -    2           2    -    -    -          2    -    -    -
5) Move: Slide U        6) Move: NewTile (2,1) 2   7) Move: Slide R          8) Move: NewTile (0,1) 2
   8   64   64  512        8   64   64  512           -    8  128  512          -    8  128  512
   4    -    -    2        4    -    2    2           -    -    4    4          2    -    4    4
   8    -    -    -        8    -    -    -           -    -    -    8          -    -    -    8
   2    -    -    -        2    -    -    -           -    -    -    2          -    -    -    2
```

Figure 1: Examples of the moves in 2048 (game with parameters $(w, h) = (4, 4)$).

More specifically, solve the tasks in the following steps:

1. Define *startState* (*Game2048 w h*) which returns the empty board for a game with parameters w and h.

2. Define *move g p s m* which returns the new state of the game, after move m has been applied to state s by player p (g is again of the form *Game2048 w h*).

3. Define *moves g p s* which returns the list of feasible moves in a game with parameters g in state s, where the current player is p. Along with each move, a weight of type *Double* is given (look at the type signature of *moves*).

   - Case $p = \mathit{False}$: If n is the number of free positions in the grid, the list returned by *moves* contains elements of the form $(0.9 \;/\; n, \mathit{NewTile}\ (i,j)\ 2)$ and $(0.1 \;/\; n, \mathit{NewTile}\ (i,j)\ 4)$. The weight parameter describes the probability of the given move in the official 2048 Game (after having chosen a free position uniformly at random, the value 2 is used with probability 90% and 4 with probability 10%).
   - Case $p = \mathit{True}$: If n is the number of directions available to the player (recall that only directions which change the game state are available), the list returned by *moves* contains elements of the form $(1 \;/\; n, \mathit{Slide}\ d)$ where d is a *Direction*.

   HINT: It can be helpful to use the *Prelude* function $\mathit{fromIntegral} :: (\mathit{Num}\ b, \mathit{Integral}\ a) \Rightarrow a \to b$ to convert an *Int* to a *Double*.

4. Define *value g p s* which returns a *Double* value describing the current state of the game.

   It is a heuristic function which describes how "favorable" the current state of the game is to player *True*.

   For a game with parameters *Game2048 w h*, we define the constant $m = \min\{w, h\}$ and 4 matrices with dimensions $w \times h$ (notice that $G^{(2)}, G^{(3)}, G^{(4)}$ can be generated from $G^{(1)}$ by mirroring in horizontal and vertical directions):

$$G^{(1)} = \begin{bmatrix} m & m-1 & \dots & m-w+1 \\ m-1 & m-2 & \dots & m-w \\ \dots & \dots & \dots & \dots \\ m-h+1 & m-h & \dots & m-w-h+2 \end{bmatrix} \quad G^{(2)} = \begin{bmatrix} m-w+1 & \dots & m-1 & m \\ m-w & \dots & m-2 & m-1 \\ \dots & \dots & \dots & \dots \\ m-w-h+2 & \dots & m-h & m-h+1 \end{bmatrix}$$

$$G^{(3)} = \begin{bmatrix} m-h+1 & m-h & \dots & m-w-h+2 \\ \dots & \dots & \dots & \dots \\ m-1 & m-2 & \dots & m-w \\ m & m-1 & \dots & m-w+1 \end{bmatrix} \quad G^{(4)} = \begin{bmatrix} m-w-h+2 & \dots & m-h & m-h+1 \\ \dots & \dots & \dots & \dots \\ m-w & \dots & m-2 & m-1 \\ m-w+1 & \dots & m-1 & m \end{bmatrix}$$

   We call these matrices **Gradient matrices**. For each of the gradient matrices given above, the current game state (the board of type $[[\mathit{Int}]]$) should be multiplied pointwise with the gradient matrix, and the products should be summed. If the game state was a mathematical matrix S, the calculations would be:

$$\sum_{i=1}^{w} \sum_{j=1}^{h} G^{(k)}_{i,j} S_{i,j}$$

   for $k \in \{1..4\}$. The *value* is defined to be the largest of these sums, i.e.

$$\max_{k \in \{1..4\}} \left\{ \sum_{i=1}^{w} \sum_{j=1}^{h} G^{(k)}_{i,j} S_{i,j} \right\}$$

5. Define *showGame g s* which outputs a string representation of the board.

   - Let s be the length of the longest string representation of a number in the matrix. Example: The length of the string representation of 2 is 1, and the length of 2048 is 4.
   - Empty fields should be denoted by a dash `-` instead of `0`.
   - Each string rep. of numbers in the matrix should be padded with spaces in the beginning, such that is exactly of length $\max\{5, s\}$.
   - Column of the matrix are separated by a single space.
   - Rows of the matrix are separated by a newline `\n`.

   Example of a game defined with parameters *Game2048* 4 4:

```
4     2    16     4
-     8    32     8
-    64   256    32
-    16   512  1024
```

Testing your program:
Apart from your own testing, please also test your program by running the *main*-function in `Game2048Test.hs`. You need QuickCheck to run the test. To install QuickCheck on a computer with Haskell Platform, write `cabal install quickcheck` in the Terminal. Be aware that the test does not test your *value*-function.

2 The Minimax algorithm

This section is not relevant for people following the DM509 (5 ECTS) course.

The goal of this section is to implement an algorithm that can play any game g which is an instance of the type class *Game g* - Game 2048 is one of these games. The task is split into separate parts:

1. Write a function which lazily generates the game tree of the game.
2. Write a function which cuts off the game tree at a certain depth
3. Write a function which, by inspection of the game tree, decides on a *next move* for player *True*. The algorithms to be implemented are

   - The *Minimax algorithm* which gives the best next move, assuming that the *Player False* tries to minimize the *value* function.
   - The *Expectimax algorithm* which gives the best next move, by assuming that *Player False* chooses each move with the probability given by the Weight-parameter.

It is crucial to this program design (splitting point 1 and 2 into separate parts) that Haskell is a *lazy language* - if not, the complete game tree would need to be stored in the memory, which is not reasonable.

Your work for this part of the exercise should be put in the file `GameStrategies.hs`.

2.1 The Game Tree - Implement the functions *tree* and *takeTree*

You are given the algebraic data type *Tree*, and a type synonym *GTree g* which is the type of a game tree for the game g:

data *Tree m v* = *Tree v* $[(m,$ *Tree m v*$)]$
type *GTree g* = *Tree* $($*Weight*, *Move g*$)$ $($*Player*, *GameState g*$)$

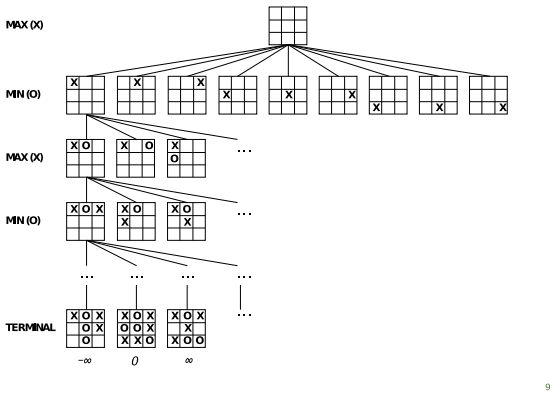

Figure 2: A game tree for Tic Tac Toe.

A node in a game tree corresponds to a tuple of a player p and a game state s, (p, s). The player p is the player to make the next move, and the state s is the current state of the game.

The children of a node are all the nodes of the form $(\neg\ p, s')$, where s' is a new state, which can be achieved by making a feasible move m from the previous state. Implement the function

tree :: *Game g* $\Rightarrow$ *g* $\rightarrow$ $($*Player*, *GameState g*$)$ $\rightarrow$ *GTree g*

which, given a game instance (e.g. *Game2048* 4 4), and a node (p, s) :: $($*Player*, *GameState g*$)$, outputs the complete game tree rooted at the node. You should use the *move* and the *moves* function (guaranteed to be implemented by the type class *Game*) to build the game tree.

You are furthermore going to provide a function

takeTree :: *Depth* $\rightarrow$ *Tree m v* $\rightarrow$ *Tree m v*

which cuts off children below a certain depth in the tree.

2.2 Properties of the value function

Each node in the game tree has a corresponding value, given by the *value* function (guaranteed to be implemented by the type class *Game*). This value is going to be interpreted in the following way: The value function is just a *heuristic* function, which should satisfy the following intuition: Given feasible moves m_1 and m_2, leading to children (p', s_1) and (p', s_2):

- If *value g p*$'$ $s_1 >$ *value g p*$'$ s_2, the state s_1 is more advantageous to Player *True* and the state s_2 is more advantageous to Player *False*.

The **best move for player** *True* is therefore the move that **maximizes** the value function, and the **best move for player** *False* is the move that **minimizes** the value function.

2.3 The Minimax and Expectimax Algorithms - Implement the functions *minimax* and *expectimax*

You are going to implement the following functions:

$$\begin{aligned} &\mathit{minimax} &&:: \mathit{Game}\ g \Rightarrow g \to \mathit{GTree}\ g \to (\mathit{Maybe}\ (\mathit{Move}\ g), \mathit{Value}) \\ &\mathit{expectimax} &&:: \mathit{Game}\ g \Rightarrow g \to \mathit{GTree}\ g \to (\mathit{Maybe}\ (\mathit{Move}\ g), \mathit{Value}) \end{aligned}$$

Imperative pseudocode for the Minimax-algorithm can be seen on the last page (Note that an ACTION corresponds to a *move* and evaluation of UTILITY corresponds to a *value*). The output of the *minimax* function can be described as follows:

- For internal nodes (p, s), it returns $(\mathit{Just}\ m, v)$ where m is the move leading to the child with the
  - maximal minimax-value v if p is *True*
  - minimal minimax-value v if p is *False*

  (the minimax-value v of the child is determined by a recursive call to *minimax*).
- For terminal nodes, it evaluates the *value* function at the node

Imperative pseudocode for the Expectimax-algorithm can be seen on the last page. The output of the *expectimax* function can be described as follows:

- For internal nodes (p, s):
  - if p is *True*: It returns $(\mathit{Just}\ m, v)$ where m is the move leading to the child with the maximal expectimax-value v if p is *True*
  - if p is *False*: It returns $(\mathit{Nothing}, v)$ where $v = \sum_{i=1}^{n} w_i v_i$ (n is the number of child nodes, w_i is the weight of the ith child and v_i is the expectimax value of the child)

  (the expectimax-value v of the child is determined by a recursive call to *expectimax*).
- For terminal nodes, it evaluates the *value* function at the node

Note: When using the *minimax*/*expectimax*-functions in conjunction with the game 2048 implemented in the previous section, we will always cut off the game tree at a certain depth, i.e. for $g = \mathit{Game2048}\ w\ h$, the call $\mathit{minimax}\ g\ (\mathit{takeTree}\ d\ (\mathit{tree}\ g\ (p, s)))$ would compute the best next-move when inspecting tree to depth d, starting from node (p, s).

Please refer to the next page for pseudo-code of the Minimax algorithm. You can also look at Wikipedia for further information.

- `http://en.wikipedia.org/wiki/Minimax`
- `http://en.wikipedia.org/wiki/Expectiminimax_tree`

Testing your program:
Apart from your own testing, please also test your program by running the *main*-function in `GameStrategiesTest.hs`. You need QuickCheck to run the test. To install QuickCheck on a computer with Haskell Platform, write `cabal install quickcheck` in the Terminal. Please also test your program by running the *testrun* program. This can be compiled by `ghc -O2 -fforce-recomp testrun`. Run the program without arguments to see argument descriptions for the tool. Example runs are

```
./testrun minimax 4 4 4
./testrun expectimax 4 4 4
```

3 Report and evaluation

You should hand in a succint report describing all implementation options. The report should also include a full listing of the program with appropriate documentation, as well as a section with example executions. The evaluation will take into account the following aspects: Program (50%), Execution, correctness and efficiency (20%) and Report (30%).

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return *v*

Figure 3: Imperative pseudo-code for minimax.

```
function expectiminimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        // Return value of minimum-valued child node
        let α := +∞
        foreach child of node
            α := min(α, expectiminimax(child, depth-1))
    else if we are to play at node
        // Return value of maximum-valued child node
        let α := -∞
        foreach child of node
            α := max(α, expectiminimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        foreach child of node
            α := α + (Probability[child] * expectiminimax(child, depth-1))
    return α
```

Figure 4: Imperative pseudo-code for expectiminimax. Note that in this exercise, there are no minimizing nodes - only maximixing nodes (player *True* and *chance* nodes (player *False*) - therefore we call the algorithm in this assignment *expectimax*.