Programming Languages

Exam Project (Project 1 of 2) (deadline: Monday November 13th, 2017 at 23.59)

This project is mandatory for the completion of the DM552 course. You will get a grade on the 7-step-scale based on your work in both projects.

1 The Kalaha game with parameters (n, m) (40%)

Kalaha is played by two people on a board with n pits on each side and two stores, called kalahas.

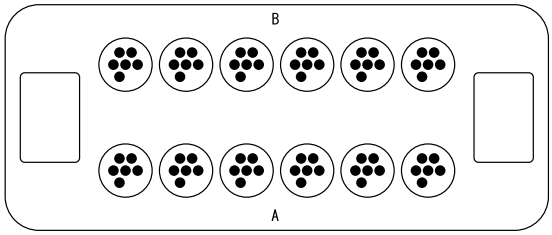

Figure 1: The Kalaha board with parameters $n, m = 6$.

In each pit, there are initially m stones. A move is made by taking all stones from a pit on your own side and sowing them one-by-one in counterclockwise direction. Your own kalaha is included in the sowing, but the opponents kalaha is skipped.

There are three possible outcomes of a turn:

1. The sowing ends in your own kalaha: It is your turn to move again.
2. The sowing ends in an empty pit on your own side: All stones in the opposite pit (on the opponents side) along with the last stone of the sowing are placed into your kalaha and your turn is over.
3. Otherwise (the sowing ends on the opponents side or in a nonempty pit on your own side): Your turn is over.

If all pits on your side become empty, the opponent captures all of the remaining stones in his pits. These are placed in the opponents kalaha and the game is over. You win the game when you have $nm + 1$ or more stones in your kalaha. If both players end up with nm stones, the game is tied.

1.1 Representation of a Kalaha-game in Haskell

Please download the zip-file Kalaha.zip - inside there is a file called Kalaha.lhs with the types for all the functions required in this project. All the work in this project should be done in this file - both the report and the code. It is a *Literate Haskell* file, meaning that all code should be written inside a `\begin{code}...\end{code}` environment. The file can be loaded by GHC and GHCi like normal Haskell files. You can compile your report to a PDF by running `make clean && make`. You are allowed to import and use the `Prelude` and the `Data.List` module, but everything else you should implement yourself.

- To represent the players in a game of Kalaha, we use the data type

  type *Player* = *Bool*

  The first player (player A on the figures) is represented by *False*. The other player is represented by *True*.

- To represent the parameters of a Kalaha game, we define the types

  data *Kalaha* = *Kalaha PitCount StoneCount*
  type *PitCount* = *Int*
  type *StoneCount* = *Int*

- The state of a *Kalaha n m*-game is represented a list of length $2n + 2$. The elements in the list is the number of stones in each pit/store.

  type *KPos* = *Int*
  type *KState* = [*Int*]

  On the figure below, the interpretation of the list is illustrated for *Kalaha* 6 6

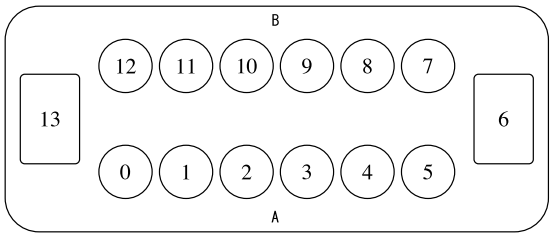

Figure 2: The Kalaha board with parameters $n, m = 6$. The pits for Player A has indexes $0 \ldots n-1$, the store for Player A has index n, the pits for Player B has indexes $n+1 \ldots 2n$ and the store for player B has index $2n+1$.

1.2 Functions to be implemented

More specifically, solve the tasks in the following steps:

1. Implement the function

   startStateImpl :: *Kalaha* → *KState*

   which returns the initial state of the Kalaha-game. Example outputs:

   - *startStateImpl* (*Kalaha* 6 6) = $[6, 6, 6, 6, 6, 6, 0, 6, 6, 6, 6, 6, 6, 0]$
   - *startStateImpl* (*Kalaha* 6 4) = $[4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0]$
   - *startStateImpl* (*Kalaha* 4 4) = $[4, 4, 4, 4, 0, 4, 4, 4, 4, 0]$

2. Implement the function

   valueImpl :: *Kalaha* → *KState* → *Double*

   For a given game g, and state s, *valueImpl g s* returns $v_{True} - v_{False}$, where v_p is current value of player p's store, and converts it to a *Double*.

3. Implement the function

   $movesImpl :: Kalaha \to Player \to KState \to [KPos]$

   For a given game g, player p and state s, *movesImpl g p s* returns the pits of player p which has a positive number of elements.

4. Implement the function

   $moveImpl :: Kalaha \to Player \to KState \to KPos \to (Player, KState)$

   For a given game g, player p, state s and move m (which is the index of a pit), *moveImpl g p s m* implements the logic of a move in Kalaha, as described in the introduction. It should return a tuple of a player and a state (p', s'), where p' is the next player, and s' is the changed state.

   Please explain the efficiency of your implementation - a time complexity of $O(n)$ can be achieved, instead of the $O(nm)$ for a 'naive' implementation.

5. Implement the function

   $showGameImpl :: Kalaha \to KState \to String$

   For a given game g, and state s, *showGameImpl g s* returns a pretty output of the given Kalaha-state.

   As an example, the output of

   $putStrLn\ (showGameImpl\ (Kalaha\ 6\ 6)\ [0,5,2,2,1,0,31,0,0,11,0,4,0,16])$

   is

```
    0  4  0 11  0  0
16                   31
    0  5  2  2  1  0
```

   It outputs 3 lines:

   - Line 1 is the pits of player *True* (in reverse)
   - Line 2 is the stores of both players
   - Line 3 is the pits of player *False*

   For a general game *Kalaha n m*, the function first computes $maxLen = length\ (show\ (2*n*m))$ (which is the maximum length of the decimal representation of a count in such a Kalaha-game.

   The function should pad small numbers with spaces, such that each number is represented by a string of length *maxLen*. In addition to the padding, the numbers on each line should be separated by a space.

Testing your program:
Apart from your own testing, please also test your program by running the *main*-function in `KalahaTest.hs`. You need QuickCheck to run the test. To install QuickCheck on a computer with Haskell Platform, write `cabal install quickcheck` in the Terminal.

2 Trees (20%)

You are given the algebraic data type *Tree*

```
data Tree m v = Node v [(m, Tree m v)]
```

An example tree is

```
testTree :: Tree Int Int
testTree = Node 3 [(0, Node 4
      [(0, Node 5 []), (1, Node 6 []), (2, Node 7 [])])
   , (1, Node 9
      [(0, Node 10 [])])
   ]
```

1. Provide a function

   ```
   takeTree :: Int → Tree m v → Tree m v
   ```

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

   ```
   takeTree 0 testTree ≡ Node 3 []
   takeTree 1 testTree ≡ Node 3 [(0, Node 4 []), (1, Node 9 [])]
   ```

3 The Minimax algorithm (40 %)

The goal of this section is to implement an algorithm that can play any game, which can be represented as an element of type *Game* - the Kalaha is one of these games.

```
data Game s m = Game {
  startState  :: s,
  showGame    :: s → String,
  move        :: Player → s → m → (Player, s),
  moves       :: Player → s → [m],
  value       :: Player → s → Double}
```

```
kalahaGame :: Kalaha → Game KState KPos
kalahaGame k = Game {
  startState = startStateImpl k,
  showGame   = showGameImpl k,
  move       = moveImpl k,
  moves      = movesImpl k,
  value      = const (valueImpl k)}
```

Figure 3: The Game data type and the kalaha game defined with this type

The task is split into separate parts:

1. Write a function which lazily generates the game tree of the game. A node in a game tree corresponds to a tuple of a player p and a node value v, (p, v) - the player p is the player to make the next move.

   The children of a node are all the nodes of the form (p', v'), where p' is the next player to make a move, and the value v' of the next node. Implement the function

   ```
   tree :: Game s m → (Player, s) → Tree m (Player, Double)
   ```

   which outputs the complete game tree. You should use the *value*, *move* and the *moves* function in your implementation.

2. Implement a function

$$minimax :: Tree\ m\ (Player, Double) \to (Maybe\ m, Double)$$

which takes a game tree (as generated in the previous exercise), and returns the optimal *next move*, along with the minimax-value associated with this move.

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

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, v), it returns $(Just\ m, v')$ 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 (p, v), it returns $(Nothing, v)$.

3. Implement a function

$$minimaxAlphaBeta :: AlphaBeta \to Tree\ m\ (Player, Double) \to (Maybe\ m, Double)$$
$$\textbf{type}\ AlphaBeta = (Double, Double)$$

which is an optimized version of the minimax algorithm, applying a technique called (α, β)-pruning.

The *minimaxAlphaBeta* function is similar to the *minimax* function, except that it might not look at all children of an internal node. The algorithm takes an extra parameter, (α, β), which initially is $(-\infty, \infty)$. When iterating through the list of children for a node, it keeps updating this tuple (α, β), and stops looking more elements of the list if at some point $\alpha \geqslant \beta$. We refer to the pseudo-code on the last page of this project description, for a precise description of how the pruning should work.

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` and `http://en.wikipedia.org/wiki/Alpha-beta_pruning`.

Testing: You are given QuickCheck based tests in the file GameStrategiesTest.hs. Please test your implementation using these tests.

When having implemented *minimax* and/or *minimaxAlphaBeta*, you can test play the algorithms with the Kalaha, by runhaskell MainTestRun.hs "Kalaha 6 6" "(Minimax, 2)" "(AlphaBeta, 5)". Here, a minimax-strategy with depth 2 is playing against an AlphaBeta-strategy with depth 5.

4 Report and evaluation

Your report should be written in the file Kalaha.lhs, along with the code required for the project. You should explain your implementation of all the functions, and how you tested your functions.

Correctness of your code is the most important, but your final grade will also be influenced by the quality of your report, and the style of your code.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← 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 ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 4: Imperative pseudo-code for minimax.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5: Imperative pseudo-code for minimax with (α, β)-pruning