Programming Languages (Project 1)

Jeff Gyldenbrand (jegyl16)

November 17, 2017

Abstract

The goal of this project is to implement an array of unique complementary functions. When these functions are assembled in a correct order they allow the programmer to construct a program, which posseses the capability, to brute force a victory in Kalaha. This array of functions, as perscribed by they assignment, are to be implemented in the programming language called haskell.

Contents

1 The Kalaha game with parameters (n, m)

```haskell
module Kalaha where

import Data.List

type PitCount   = Int
type StoneCount = Int
data Kalaha     = Kalaha PitCount StoneCount deriving (Show, Read, Eq)

type KPos       = Int
type KState     = [Int]
type Player     = Bool
```

1.1 The function `startStateImpl`

This function is giving with a ‘Kalaha’-game: Kalaha, which takes a pit-count and stone-count as parameters.

By replicating the pit-count and stone-count, and then append a zero, we get half the list, so we simply add the list to itself to return the initial state of the Kalaha-game.

```haskell
startStateImpl :: Kalaha -> KState
startStateImpl (Kalaha pitC stoneC) = replicate' ++ replicate'
  where
    replicate' = replicate pitC stoneC ++ [0]
```

1.2 The function `movesImpl`

In **movesImpl** which, besides the ‘Kalaha’-game parameters, now take as parameters a player; False or True, and a state for the game. Then returns the pits which has a positive count of stones of a given player.

This is done by making two guards: one for each player. If it is player False we simply use the **findIndices** function which, in our case, returns all indices greater than zero. This we can do because we split the game state into a tuble, and define player False to be the first element in the tuble. We use init to exclude player False’s own kalaha pit.

To find same indices only for player True, we define our own **findIndices’** function, that recursively searches the other element in the before mentioned tuble.

```haskell
movesImpl :: Kalaha -> Player -> KState -> [KPos]
movesImpl (Kalaha pitC stoneC) p gState
  | p == False = findIndices (>0) (pFalse)
  | p == True = findIndices' (pitC+1) (pTrue)
    where
      pFalse = init(fst(splitAt (pitC+1) gState))
      pTrue = init(snd(splitAt (pitC+1) gState))
      findIndices' _ [] = []
      findIndices' pitC (x:gState)
        | (x>0) = pitC : findIndices' (pitC+1) gState
        | otherwise = findIndices' (pitC+1) gState
```

1.3 The function **valueImpl**

In **valueImpl** which, besides the 'Kalaha'-game parameters, now takes as parameters the game state. Then returns True's kalaha pit subtracted from False's kalaha pit as a double.

As in **movesImpl** this is done by splitting the game state into two elements in a tuble. Then assigning True's kalaha pit to the last element in the first element of the tuble, and False's kalaha pit to the last element in the second element of the tuble. Then we simply subtract the two kalaha pits.

```haskell
valueImpl :: Kalaha -> KState -> Double
valueImpl (Kalaha pitC stoneC) gState = fromIntegral (pitTrue - pitFalse)
  where
    pitFalse = last(fst(splitAt(pitC+1) gState))
    pitTrue = last(snd(splitAt(pitC+1) gState))
```

1.4 The function **moveImpl**

In **moveImpl** we take all the same parameters as in **movesImpl** only now we want to return the the logic of a move, meaning the next player and the new state, in a tuble.

To accomplish this, several help functions is developed: **letsMove**, **pickUpStones**, **incVal**, **stealOpposite**, **collector**, **otherRules**, **outOfStones**, and **sweapBoard**.

All these help functions are needed in order to define a ruleset of the move, and will be explained in more details in the beginning of each. As for the main function it will read the given parameters, and will call the function **pickUpStones** to pick up the stones at giving index, before making the first move called by the function **letsMove**

```haskell
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player,KState)
moveImpl (Kalaha pitC stoneC) p gState pitIndex = nextTurn
 where
   nextTurn = letsMove (Kalaha pitC stoneC) p updatedState (pitIndex+1) pitVal
   pitVal = gState!!pitIndex
   updatedState = pickUpStones pitIndex gState
```

The main help-function 'letsMove'

letsMove is where the primary action is happening. As parameters it takes a kalaha game, a player, a newGameState, the current index and its value. With guards we check for some condition, that will determine the next legal move:

1. if we have zero stones left in the hand, we check for the other rules. This is explained in more detailts in the help-function **endCheck**.
2. As long as we have stones in the hand we move.. and to prevent a move to reach past the last index in the list, we jump back at the beginning (index zero), and drop a stone.
3. If player false lands in player trues kalaha, we skip it, and land at player falses start pit, and drop a stone.
4. if player true lands in player falses kalaha, we skip it, and land at player trues start pit, and drop a stone.
5. if none of above rules are violated, we move to next pit and drop a stone.

```haskell
letsMove (Kalaha pitC stoneC) p gState pIndex stones
 | (stones == 0) = outOfStones (Kalaha pitC stoneC) p (pIndex-1) gState     -- 1
 | (stones > 0) && (pIndex > pitC*2+1) =                                   -- 2
   letsMove (Kalaha pitC stoneC) p beyond 1 (stones-1)
 | (p == False) && (pIndex == 2*pitC+1) =                              -- 3
   letsMove (Kalaha pitC stoneC) p skipTrue 1 (stones-1)
 | (p == True) && (pIndex == pitC) =                                   -- 4
   letsMove (Kalaha pitC stoneC) p skipFalse (pIndex+2) (stones-1)
 | otherwise = letsMove (Kalaha pitC stoneC) p nextMove (pIndex+1) (stones-1)--5
 where
  nextMove = incVal pIndex gState 1
  beyond = incVal 0 gState 1
  skipTrue = incVal 0 gState 1
  skipFalse = incVal (pIndex + 1) gState 1
```

The help-function 'pickUpStones'

We start a move by picking up all the stones from a chosen pit. We look at the current game state and changes the giving index's value to zero.

We can't just change it directly, so with a little work-a-round by splitting the game state into a tuble, appending a zero (which is the replacement for the actual value) and creating a new list, we get the new game state which is returned to

```
pickUpStones pitIndex gState = newgState
 where
   removeIndexValue = init(fst(splitAt(pitIndex + 1) gState))
   restOfList = snd(splitAt(pitIndex + 1) gState)
   newgState = (removeIndexValue ++ 0 : restOfList)
```

The help-function 'incVal'

This function takes as argument a pit index, game state and a value n. When ever this function is invoked, the value at giving index is incremented by n, and returns the new game state to the function it is invoked by.

```
incVal pitIndex gState n = updateState
 where
   pitValue = gState!!pitIndex
   pitsFalse = init(fst(splitAt(pitIndex + 1) gState))
   pitsTrue = snd(splitAt (pitIndex + 1) gState)
   updateState = (pitsFalse ++ (pitValue + n) : pitsTrue)
```

The help-function 'stealOpposite

In the case where a player drops hes last stone in one of hes own empty pits, he will steal all the stones from the opposite pit (from the opponent) plus one (that last stone he dropped in hes own pit). All these stones will go into the players kalaha.

So based on whether we are player False or True, we invoke function **stealFromTrue** or **stealFromFalse** which invokes the functions **incVal** and **pickUpStones** which updates the state of the game with help from:

1. **totalStones** which adds the opposit pits stones with one (hence stealing the stones from the opponent plus our last dropped stone)
2. **emptyLast** which make sure to empty the pit, where we dropped the last stone.
3. **emptyOpposite** which empty the opposite pit for stones.

```
stealOpposite (Kalaha pitC stoneC) player index gState
 | player == False = stealFromTrue
 | player == True = stealFromFalse
 where
   totalStones = (gState!!(index+((2*pitC)-(index*2)))) + 1
   emptyLast = pickUpStones index gState
```

```haskell
   emptyOpposite = pickUpStones (index+((2*pitC)-(index*2))) emptyLast

   stealFromTrue = incVal pitC emptyOpposite totalStones
   stealFromFalse = incVal ((pitC*2)+1) emptyOpposite totalStones
```

The help-function 'collector'

This function is called by sweapBoard to empty a players remaining stones one pit at a time, and add it to the this players kalaha.

```haskell
collector (Kalaha pitC stoneC) player index gState
 | player == False = nextFalse
 | otherwise = nextTrue
 where
  stones = gState!!index
  newState = pickUpStones index gState
  nextFalse = incVal pitC newState stones
  nextTrue = incVal (pitC*2+1) newState stones
```

The help-function 'otherRules'

In this function we check for the other rules:

1. if player False lands in an empty pit
2. if player True lands in an empty pit
3. if player False lands in player own kalaha
4. if player True lands in player own kalaha
5. nothing happens, and returns state and its next players turn

```haskell
otherRules (Kalaha pitC stoneC) p pitIndex gState
 | (p == False) && (gState!!pitIndex == 1) && (pitIndex < pitC) = case1
 | (p == True) && (gState!!pitIndex == 1) && (pitIndex > pitC) && (pitIndex < ((pitC*2)
    +1)) = case2
 | (p == False) && (pitIndex == pitC) = case3
 | (p == True) && (pitIndex == pitC*2+1) = case4
 | otherwise = (not p, gState)
  where
  emptyFalseOpposite = (stealOpposite (Kalaha pitC stoneC) False pitIndex gState)
  emptyTrueOpposite = (stealOpposite (Kalaha pitC stoneC) True pitIndex gState)
  case1 = (True, emptyFalseOpposite)
  case2 = (False, emptyTrueOpposite)
  case3 = (False, gState)
  case4 = (True, gState)
```

The help-function 'outOfStones'

In this function we check if the pits of a player are empty. In that case the oppenent will collect all hes own stones to hes kalaha, and the game will end.

We do this by looking for the case where we get the value 'Nothing' from finding indices greater than zero in both player true and false's pits. So if a players pits are empty, we invoke the `sweapBoard` function for the opposite player which then collects hes own stones.

```haskell
outOfStones (Kalaha pitCount stoneCount) player pitIndex gameState
 | (findIndex (>0) (fst(splitAt pitCount gState)) == Nothing) = (not player, tCollect)
 | (findIndex (>0) (init(snd(splitAt (pitCount+1) gState))) == Nothing) = (not player,
    fCollect)
 | otherwise = otherRules (Kalaha pitCount stoneCount) player pitIndex gameState
```

```
  where
   gState = snd(otherRules (Kalaha pitCount stoneCount) player pitIndex gameState)
   listOfindexes = findIndices (>0) gState
   tCollect = sweapBoard (Kalaha pitCount stoneCount) True gState listOfindexes ((
    length listOfindexes) -1)
   fCollect = sweapBoard (Kalaha pitCount stoneCount) False gState listOfindexes ((
    length listOfindexes) -1)
```

The help-function 'sweapBoard'

So the sweapBoard function is invoked by outOfStones, and recursively extracts the values in the pits into the correct players kalaha, by invoking the function collector. Ultimately it will return the final game state.

```
sweapBoard (Kalaha pitC stoneC) p gState indexList pitIndex
 | (pitIndex<0) = gState
 | (extractVal == pitC) = sweapBoard (Kalaha pitC stoneC) p gState indexList (pitIndex
    -1)
 | (extractVal == pitC*2+1) = sweapBoard (Kalaha pitC stoneC) p gState indexList (
    pitIndex-1)
 | otherwise = sweapBoard (Kalaha pitC stoneC) p execute indexList (pitIndex-1)
 where
    extractVal = indexList!!pitIndex
    execute = collector (Kalaha pitC stoneC) p extractVal gState
```

1.5 The function showGameImpl

For a given game and state we want a pretty output of the game state. For this we define tree lines, one for player true, one for the two kalahas, and one for player false. We use unlines to join the three lines with newlines appended, and also we map unwords to the three lines to seperate them by spaces.

With some help function **pad** and **part**, we align numbers perfectly. And by defining some empty space from the length of our list, we push line1 and line2 to their proper place.

```
showGameImpl :: Kalaha -> KState -> String
showGameImpl g@(Kalaha pitC stoneC) gameState =
  unlines $ map unwords [line1, line2, line3]
  where
    maxLen = length $ show $ 2*pitC*stoneC
    empty = replicate maxLen ' '

    gameState' = map (pad maxLen) $ map show gameState
    (line1, line3) = (empty : (reverse $ part(pitC+1, 2*pitC+1) gameState'), empty : (
    part(0,pitC) gameState'))
    line2 = last gameState' : (replicate pitC empty ++ [gameState'!!pitC])

pad :: Int -> String -> String
pad pitC s = replicate (pitC - length s) ' ' ++ s

part :: (Int, Int) -> [a] -> [a]
part (x,y) l = drop x $ take y l
```

2 Trees

```haskell
data Tree m v  = Node v [(m,Tree m v)] deriving (Eq, Show)
```

Test tree This is a test tree giving from the assignement. We use this to develop the **takeTree** function

```haskell
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[])])
    ]
```

2.1 The function `takeTree`

As a lazy evaluated game tree is to be constructed later in the assignment, a function which allows to block off any children at a specified depth. As such a function, which takes an interger and list as parameters, implemented. This implementation, will recusivly run thorugh the tree, creating a copy of said tree. This recursion, will for each stack element, decrement the allowed depth. When a given stack recieves depth count being zero, it will not create any further stacks, hence the $n == 0$ check.

```haskell
takeTree :: Int -> Tree m v -> Tree m v
takeTree n (Node v list)
 | (n==0) = (Node v [])
 | otherwise = (Node v (map tree' list))
  where
   tree' (m,t) = (m, takeTree (n-1) t)
```

3 The Minimax algorithm

```haskell
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)}

startTree :: Game s m -> Player -> Tree m (Player,Double)
startTree g p = tree g (p, startState g)
```

3.1 The function `tree`

This function, **tree**, outputs the complete game tree for a given game, player and game state. The output of the function is in one long line, as to make it more readable, the function **showTree** is implemented.

Function **showTree** is a modification of the function from class exercise giving by the teachers assisten: Henrik (henpe15): “Induction proofs, functions on trees, and Monoids” http://imada.sdu.dk/~henpe15/dm552-17/haskell/ex/w6_1.lhs

```
tree :: Game s m -> (Player, s) -> Tree m (Player, Double)
tree game (player, state) = Node (player, treeValue) treeMove
    where
    treeValue = value game player state
    treeMoves = moves game player state
    treeMove = [(m, tree game (move game player state m)) | m <- treeMoves]

showTree :: (Show v, Show m) => Tree m v -> [String]
showTree (Node v []) = [show v]
showTree (Node v l@(_:xs)) = case xs of
  [] -> show v : concatMap sT' l
  _  -> show v : concatMap sT (init l) ++ sT' (last xs)
  where
    sT  (m,x) = lines ("+- " ++ show m ++ " -> " ++ drop 3 (unlines $ map ("|  " ++) (
    showTree x)))
    sT' (m,x) = lines ("+- " ++ show m ++ " -> " ++ drop 3 (unlines $ map ("   " ++) (
    showTree x)))
```

3.2 The function **minimax**

The function **minimax**, is a bruteforce algorithm designed to predict the outcome of the game. It does so by walking through the tree generate by the **tree** function. When the tree is given, the player is checked, as the player defines the applicabel rule on the given nodes children. These rules are can be deduced due to the nature of how the tree evaluates the best score.

The score on every node, is defined by the **valueImpl** function, which subtracts the score of player **false** from player **true** as shown below:

$$score = v_{true} - v_{false} \tag{1}$$

Due to the natue of this mathematical function, it is impled that, the score is going to be positiv when player true has be the biggest score. If player false were to obtian the biggest kalaha, the score would become negativ.

As such, when the given player is player **true**, the biggest score is to be searched for as to maximise the cahnce if winnig. Otherwise, when it is player false, the smallest score is to be searched for.

This can be seen on minimax as pattern maching, when the player is player true, alle the children of the given node are canned for the biggest score. When the player is player false, the can searches for the smallest score.

Function **minimax** is a modification of the function from class exercise giving by the teachers assisten: Henrik (henpe15): “Primary topic: Minimax (+AlphaBeta), tree algorithms” from URL: http://imada.sdu.dk/~henpe15/dm552-17/haskell/ex/w7_1.pdf

```
minimax   :: Tree m (Player, Double) -> (Maybe m, Double)
minimax (Node (_,treeValue) []) =  (Nothing, treeValue)
minimax (Node (True, v) ch) = maximumSnd [ (Just path, snd $ minimax subtree) | (path,
    subtree) <- ch]
minimax (Node (False, v) ch) = minimumSnd [ (Just path, snd $ minimax subtree) | (path,
     subtree) <- ch]

maxSnd :: (a,Double) -> (a, Double) -> (a, Double)
maxSnd a@(_,v1) b@(_,v2) | v1 >= v2 = a| otherwise = b

```

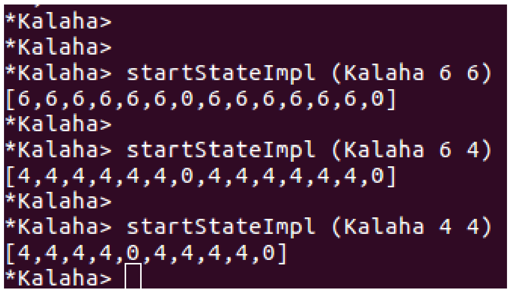
```haskell
minSnd :: (a, Double) -> (a, Double) -> (a, Double)
minSnd a@(_,v1) b@(_,v2) | v1 <= v2 = a| otherwise = b

maximumSnd :: [(a, Double)] -> (a, Double)
maximumSnd [] = error "undefined for empty list"
maximumSnd (x:xs) = foldl maxSnd x xs

minimumSnd :: [(a, Double)] -> (a, Double)
minimumSnd [] = error "undefined for empty list"
minimumSnd (x:xs) = foldl minSnd x xs
```

3.3 The function `minimaxAlphaBeta`

```haskell
type AlphaBeta = (Double,Double)

minimaxAlphaBeta :: AlphaBeta -> Tree m (Player, Double) -> (Maybe m, Double)
minimaxAlphaBeta = undefined
```

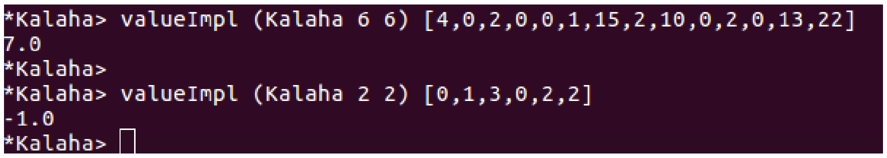
4 Testing and sample executions

We test the function **startStateImpl** for tree cases: one for a kalaha game with six pits and six stones in each, one for six pits and four stones, and finally one for four pits and four stones in each. See picture below for result.



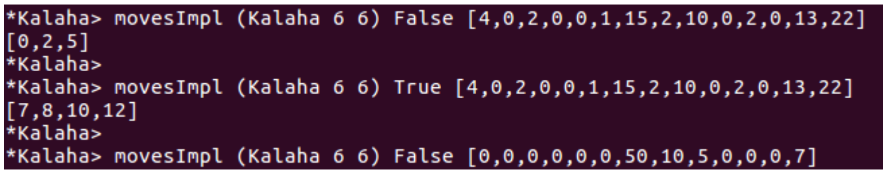
```
*Kalaha>
*Kalaha>
*Kalaha> startStateImpl (Kalaha 6 6)
[6,6,6,6,6,6,0,6,6,6,6,6,6,0]
*Kalaha>
*Kalaha> startStateImpl (Kalaha 6 4)
[4,4,4,4,4,4,0,4,4,4,4,4,4,0]
*Kalaha>
*Kalaha> startStateImpl (Kalaha 4 4)
[4,4,4,4,0,4,4,4,4,0]
*Kalaha>
```

As seen in the picture below, we test the function **valueImpl** with two different cases: one for a kalaha game with six pits, and one with only two pits. In both cases we see that player True's kalaha is subtracted from player False's kalaha, with the output 7.0 and -1.0



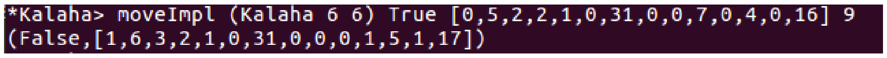
```
*Kalaha> valueImpl (Kalaha 6 6) [4,0,2,0,0,1,15,2,10,0,2,0,13,22]
7.0
*Kalaha>
*Kalaha> valueImpl (Kalaha 2 2) [0,1,3,0,2,2]
-1.0
*Kalaha>
```

As seen in the picture below, we test the function **movesImpl** with three different cases: one for player False, where we return hes pits which has positive elements, we test the same case only for player True, and one case where player False has zero positive elements, hence returning an empty list.

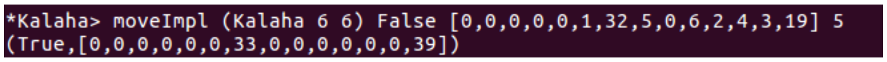


```
*Kalaha> movesImpl (Kalaha 6 6) False [4,0,2,0,0,1,15,2,10,0,2,0,13,22]
[0,2,5]
*Kalaha>
*Kalaha> movesImpl (Kalaha 6 6) True [4,0,2,0,0,1,15,2,10,0,2,0,13,22]
[7,8,10,12]
*Kalaha>
*Kalaha> movesImpl (Kalaha 6 6) False [0,0,0,0,0,0,50,10,5,0,0,0,7]
```

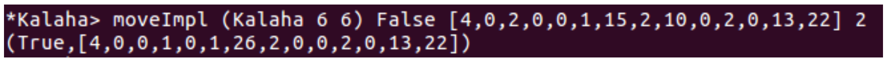
For the function **moveImpl** we look at three cases, one for a regular move for player True, as shown in the picture below, one for the case where all player False's pits become empty, so player True collects the remaining stones to hes kalaha, as shown in the picture, and lastly, the case where player False lands in one of hes own empty pits, and steals the stones from the opposite side, as shown in picture last picture.



```
*Kalaha> moveImpl (Kalaha 6 6) True [0,5,2,2,1,0,31,0,0,7,0,4,0,16] 9
(False,[1,6,3,2,1,0,31,0,0,0,1,5,1,17])
```

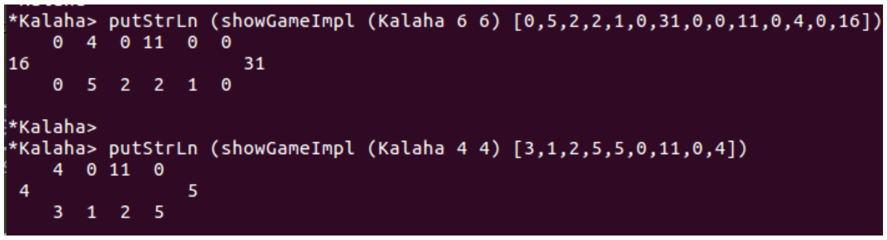


```
*Kalaha> moveImpl (Kalaha 6 6) False [0,0,0,0,0,1,32,5,0,6,2,4,3,19] 5
(True,[0,0,0,0,0,0,33,0,0,0,0,0,0,39])
```



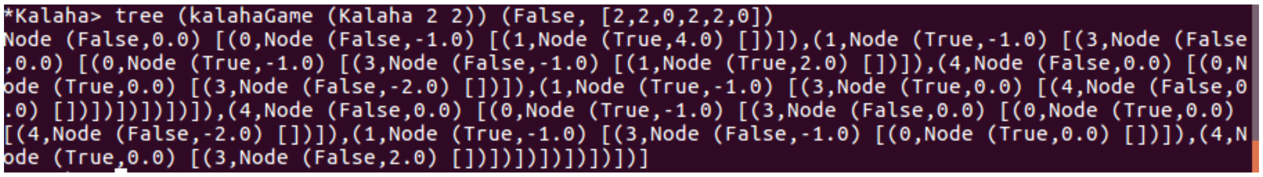
```
*Kalaha> moveImpl (Kalaha 6 6) False [4,0,2,0,0,1,15,2,10,0,2,0,13,22] 2
(True,[4,0,0,1,0,1,26,2,0,0,2,0,13,22])
```

In the function **showGameImpl** we test for two cases to make sure the output is pretty and aligned, so case one is for a kalaha with six pits, and second case for a kalaha with 4 pits. Both cases with number of one and two digits.



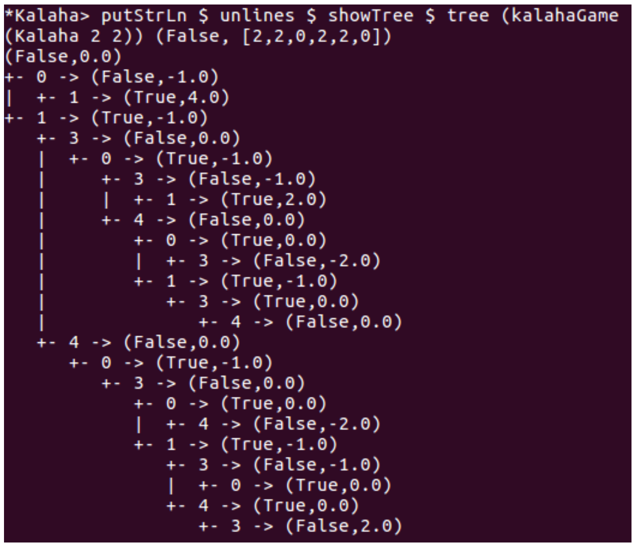
```
*Kalaha> putStrLn (showGameImpl (Kalaha 6 6) [0,5,2,2,1,0,31,0,0,11,0,4,0,16])
    0  4  0 11  0  0
16                   31
    0  5  2  2  1  0

*Kalaha>
*Kalaha> putStrLn (showGameImpl (Kalaha 4 4) [3,1,2,5,5,0,11,0,4])
    4  0 11  0
 4             5
    3  1  2  5
```

In the **treeImpl** we get the complete tree for all possible moves for a given player and kalaha state. We only test for a tree with two pits and two stones in each, as shown in the picture below, otherwise the output tree would be too big.



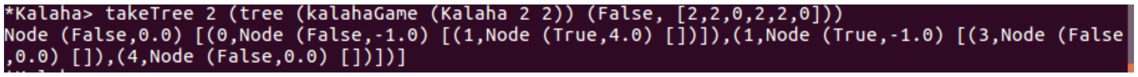
```
*Kalaha> tree (kalahaGame (Kalaha 2 2)) (False, [2,2,0,2,2,0])
Node (False,0.0) [(0,Node (False,-1.0) [(1,Node (True,4.0) [])]),(1,Node (True,-1.0) [(3,Node (False
,0.0) [(0,Node (True,-1.0) [(3,Node (False,-1.0) [(1,Node (True,2.0) [])]),(4,Node (False,0.0) [(0,N
ode (True,0.0) [(3,Node (False,-2.0) [])]),(1,Node (True,-1.0) [(3,Node (True,0.0) [(4,Node (False,0
.0) [])])])])])]),(4,Node (False,0.0) [(0,Node (True,-1.0) [(3,Node (False,0.0) [(0,Node (True,0.0)
[(4,Node (False,-2.0) [])]),(1,Node (True,-1.0) [(3,Node (False,-1.0) [(0,Node (True,0.0) [])]),(4,N
ode (True,0.0) [(3,Node (False,2.0) [])])])])])])])])
```

The function **showTree** gives us a more read-friendly view of the same tree, by invoking the function with putStrLn $ unlines $ showTree on our normal **treeImpl**

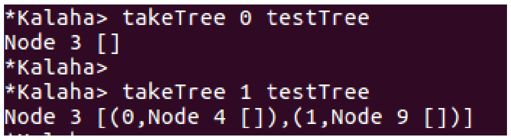


```
*Kalaha> putStrLn $ unlines $ showTree $ tree (kalahaGame
(Kalaha 2 2)) (False, [2,2,0,2,2,0])
(False,0.0)
+- 0 -> (False,-1.0)
|  +- 1 -> (True,4.0)
+- 1 -> (True,-1.0)
   +- 3 -> (False,0.0)
   |  +- 0 -> (True,-1.0)
   |     +- 3 -> (False,-1.0)
   |     |  +- 1 -> (True,2.0)
   |     +- 4 -> (False,0.0)
   |        +- 0 -> (True,0.0)
   |        |  +- 3 -> (False,-2.0)
   |        +- 1 -> (True,-1.0)
   |           +- 3 -> (True,0.0)
   |              +- 4 -> (False,0.0)
   +- 4 -> (False,0.0)
      +- 0 -> (True,-1.0)
         +- 3 -> (False,0.0)
            +- 0 -> (True,0.0)
            |  +- 4 -> (False,-2.0)
            +- 1 -> (True,-1.0)
               +- 3 -> (False,-1.0)
               |  +- 0 -> (True,0.0)
               +- 4 -> (True,0.0)
                  +- 3 -> (False,2.0)
```

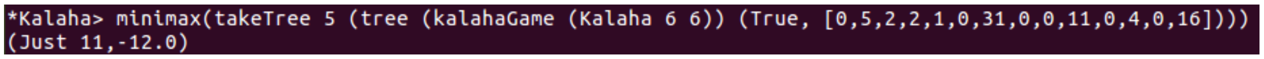
In the function **takeTree** we test with the same tree as in **treeImpl** only now we cut it off in depth two.


```
*Kalaha> takeTree 2 (tree (kalahaGame (Kalaha 2 2)) (False, [2,2,0,2,2,0]))
Node (False,0.0) [(0,Node (False,-1.0) [(1,Node (True,4.0) [])]),(1,Node (True,-1.0) [(3,Node (False
,0.0) []),(4,Node (False,0.0) [])])]
```

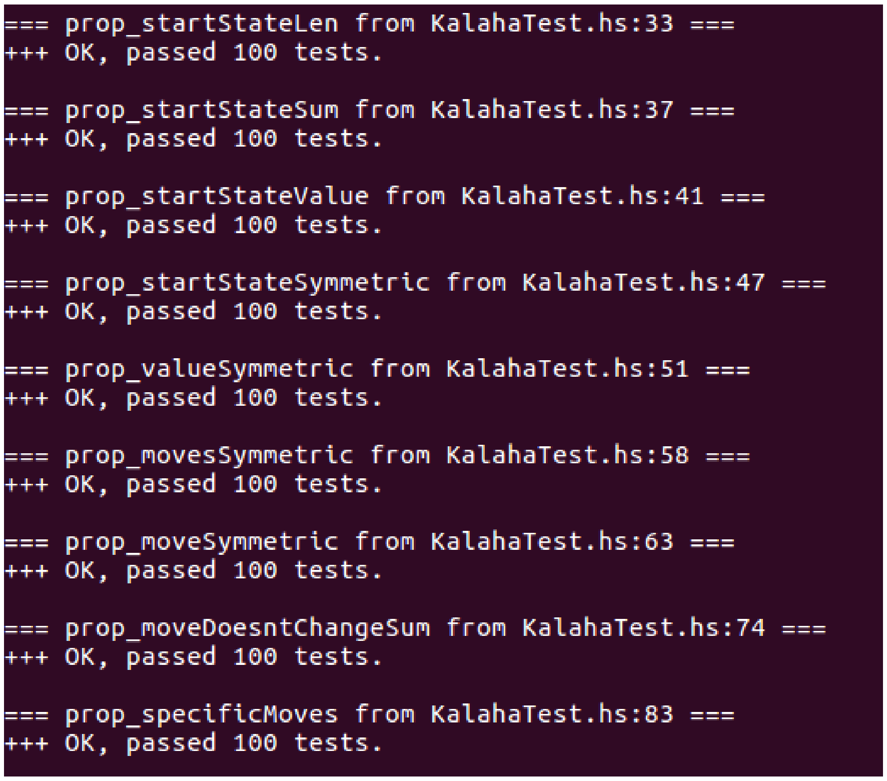
Just so see the output of depth zero and one, we cut the hard-coded test tree at zero with the result of an empty tree, and one, as shown in picture X.


```
*Kalaha> takeTree 0 testTree
Node 3 []
*Kalaha>
*Kalaha> takeTree 1 testTree
Node 3 [(0,Node 4 []),(1,Node 9 [])]
```

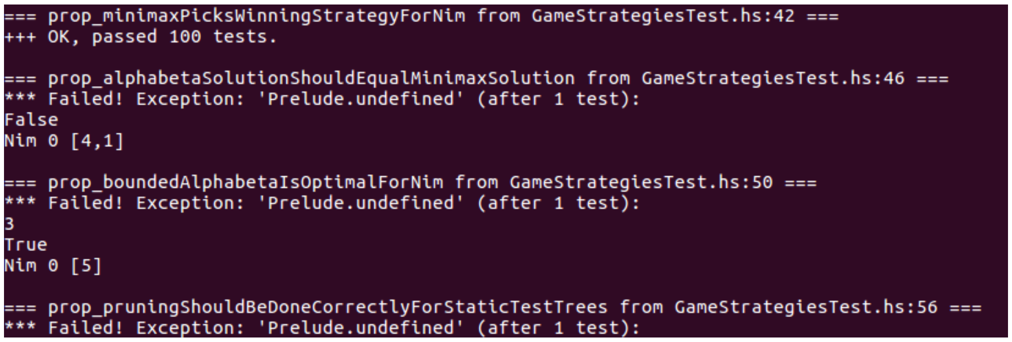
In the picture below we see the output of the minimax algorithm off a giving kalaha game, player and game state.


```
*Kalaha> minimax(takeTree 5 (tree (kalahaGame (Kalaha 6 6)) (True, [0,5,2,2,1,0,31,0,0,11,0,4,0,16])))
(Just 11,-12.0)
```

In the picture below we run the KalahaTest.lhs to confirm that everything is correct.


```
=== prop_startStateLen from KalahaTest.hs:33 ===
+++ OK, passed 100 tests.

=== prop_startStateSum from KalahaTest.hs:37 ===
+++ OK, passed 100 tests.

=== prop_startStateValue from KalahaTest.hs:41 ===
+++ OK, passed 100 tests.

=== prop_startStateSymmetric from KalahaTest.hs:47 ===
+++ OK, passed 100 tests.

=== prop_valueSymmetric from KalahaTest.hs:51 ===
+++ OK, passed 100 tests.

=== prop_movesSymmetric from KalahaTest.hs:58 ===
+++ OK, passed 100 tests.

=== prop_moveSymmetric from KalahaTest.hs:63 ===
+++ OK, passed 100 tests.

=== prop_moveDoesntChangeSum from KalahaTest.hs:74 ===
+++ OK, passed 100 tests.

=== prop_specificMoves from KalahaTest.hs:83 ===
+++ OK, passed 100 tests.
```

For the final test, we run the GameStrategiesTest.hs to confirm that our minimax algorithm is correct. As shown the alfa beta test fails giving that the algorithm never got implemented.


```
=== prop_minimaxPicksWinningStrategyForNim from GameStrategiesTest.hs:42 ===
+++ OK, passed 100 tests.

=== prop_alphabetaSolutionShouldEqualMinimaxSolution from GameStrategiesTest.hs:46 ===
*** Failed! Exception: 'Prelude.undefined' (after 1 test):
False
Nim 0 [4,1]

=== prop_boundedAlphabetaIsOptimalForNim from GameStrategiesTest.hs:50 ===
*** Failed! Exception: 'Prelude.undefined' (after 1 test):
3
True
Nim 0 [5]

=== prop_pruningShouldBeDoneCorrectlyForStaticTestTrees from GameStrategiesTest.hs:56 ===
*** Failed! Exception: 'Prelude.undefined' (after 1 test):
```