Lecture 7

Section 5 of Why Functional programming Matters

2nd glue: lazy evaluation

example from AI: computer playing games

we assume
data type Position
function moves :: Position -> [Positions]

```
reptree :: (a ->[a]) -> a -> Tree a
reptree f a = Node a (map (reptree f) (f a))
```

```
gametree :: (Position ->[Position]) -> Position -> Tree Position
gametree p = reptree moves p
```

Min-max and alpha-beta procedure

we consider the gametree
evaluate statically the positions in the tree, static :: position -> Int

value of a node=position measures how promising the position is for the computer
max when computer wins and min when it loses

The computer wants to always do the best move = take the move that leads to the position with maximum value

when the adversary moves, he/she should take the move that leads to the position with minimum value maptree static :: Tree Position -> Tree Int

max,min : [Int] -> Int

maximize,minimize :: Tree Int -> Int

maximize (Node n sub) = max (map minimize sub)

minimize (Node n sub) = min (map maximize sub)

for leaves

maximize (Node n []) = n   static evaluation of the position

minimize (Node n []) = n

```
evaluate :: (Position ->[Position]) -> Position -> Int
evaluate = maximize . maptree static . gametree
```

it doesn’t work for infinite trees

even finite game trees may be too big

we prune the tree to a fixed depth:

```
prune :: Int -> Tree a -> Tree a
prune 0 (Node a x) = Node a []
prune (n+1) (Node a x) = Node a (map (prune n) x)
```

```
evaluate = maximize . (maptree static) . (prune 5) . gametree
```

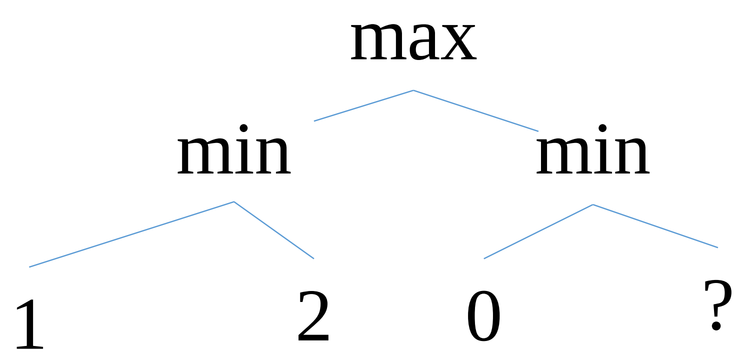
evaluate = maximize . (maptree static) . (prune 5) . gametree

higher-order functions reptree and maptree allow us to construct and manipulate possibly infinite trees

lazy evaluation allow us to separate che construction of the game tree and the computations on it

without lazy evaluation, in place of (prune 5). gametree
we should have a unique function that computes the tree counting 5 levels

(maptree static) . (prune 5) . gametree costructs the parts of the tree that maximize needs !   Again, thanks to lazy evaluation we improve maximize so that it may not need parts of the pruned tree



1 is the min of the first subree and 0 or less will be the result of the 2nd subtree => drop the ? it won’t matter

incorporate this idea in maximize and minimize maximize = max . maximize'   and similarly for minimize

maximize', minimize' :: Tree Int -> [Int]

maximize' (Node n [])= n : []

maximize' (Node n l) = map minimize l
= map (min . minimize') l
= map min (map minimize' l)
=mapmin (map minimize' l)
where mapmin = map min

map minimize' l :: [[Int]]   and map min (map minimize' l) :: [Int] that is the list of the minima of those lists, but some minima may be useless smarter mapmin:

```
mapmin :: [[Int]] -> [Int]
mapmin (nums : rest) = (min nums) : (omit (min nums) rest)

omit  pot [] = []
omit pot (nums : rest)  | minleq nums pot = omit pot rest  --nums dropped
                        | otherwise       = (min nums) : (omit (min nums) rest)

minleq [] pot = False
minleq (n:rest) | n <=pot   = True
                |otherwise = minleq rest pot

evaluate = max . maximize’. (maptree static) . (prune 8) . gametree
```