universität innsbruck

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 11, 10 points		Deadline: Wednesday, January 19, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_11.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Evaluation Strategies and Kinds of Recursion* **5 p.**

1. Given the four functions:

```
double x = x * 2
square x = x * x
add2times x y = x + double y
func x y = square x + add2times y x
```

   Evaluate each of the following expressions step-by-step under the three evaluation strategies call-by-value, call-by-name, and call-by-need. (3 points)

   (a) `add2times (5+2) 8`

   (b) `double (square 5)`

   (c) `func (2+2) 4`

2. For each of the following functions, specify which kind of recursion they use: (1 point)

   (a)
```
squareList [] = []
squareList (x:xs) = x*x : squareList xs
```

   (b)
```
doubleTimes x 0 = x
doubleTimes x y = doubleTimes (x+x) (y-1)
```

   (c)
```
add2List [] = 0
add2List (x:xs) = x + add2List xs
```

   (d)
```
average :: [Double] -> Double
average xs = aux xs 0 (fromIntegral (length xs))
  where aux [] s c = s / c
        aux (x:xs) s c = aux xs (s+x) c
```

3. Implement two variants of a function that takes a string and produces an upper case version of it: `stringToUpperTail` using tail recursion and `stringToUpperGuarded` using guarded recursion. For example `stringToUpperTail "Hello" = stringToUpperGuarded "Hello" = "HELLO"`. (1 point)

Solution 1

First notice that only for pattern matching the evaluation proceeds from left to right, i.e., evaluating the Haskell expression `let f 0 0 = 0 in f (error "left") (error "right")` leads to the error `"left"` and cannot result in the `"right"` error. By contrast, for the builtin arithmetic functions the argument evaluation is not strictly restricted from left to right, e.g., invoking `error "left" + error "right"` and `error "left" * error "right"` will result in different error messages. Therefore, in the following solution we freely chose to evaluate the arguments of builtin arithmetic functions from right to left. Having chosen left to right is also perfectly fine.

1. (a) `add2times (5+2) 8`

   - call-by-value

     ```
     add2times (5+2) 8 = add2times 7 8
                       = 7 + double 8
                       = 7 + (8 * 2)
                       = 7 + 16
                       = 23
     ```

   - call-by-name

     ```
     add2times (5+2) 8 = (5 + 2) + double 8
                       = (5 + 2) + (8 * 2)
                       = (5 + 2) + 16
                       = 7 + 16
                       = 23
     ```

   - call-by-need

     Same solution as call-by-name.

   (b) `double (square 5)`

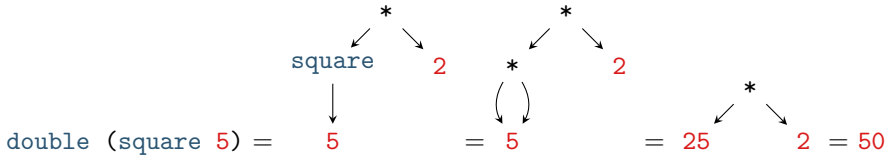
   - call-by-value

     ```
     double (square 5) = double (5*5)
                       = double 25
                       = 25 * 2
                       = 50
     ```

   - call-by-name

     ```
     double (square 5) = (square 5) * 2
                       = (5 * 5) * 2
                       = 25 * 2
                       = 50
     ```

   - call-by-need

   (c) `func (2+2) 4`

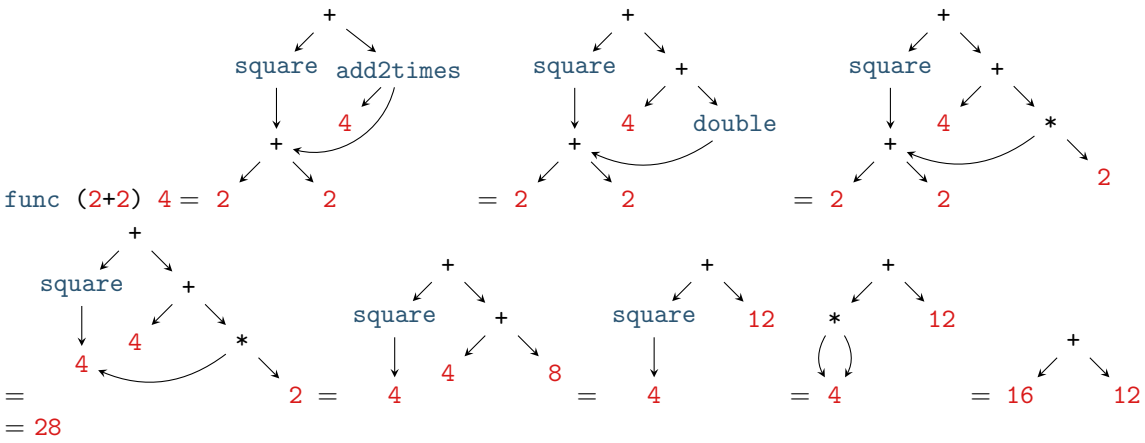
   - call-by-value

     ```
     func (2 + 2) 4 = func 4 4
                    = square 4 + add2times 4 4
                    = square 4 + (4 + double 4)
                    = square 4 + (4 + (4 * 2))
                    = square 4 + (4 + 8)
                    = square 4 + 12
                    = (4 * 4) + 12
                    = 16 + 12
                    = 28
     ```

   - call-by-name

```
func (2 + 2) 4 = square (2 + 2) + add2times 4 (2 + 2)
               = square (2 + 2) + (4 + double (2 + 2))
               = square (2 + 2) + (4 + ((2 + 2) * 2))
               = square (2 + 2) + (4 + 4 * 2)
               = square (2 + 2) + (4 + 8)
               = square (2 + 2) + 12
               = ((2 + 2) * (2 + 2)) + 12
               = ((2 + 2) * 4) + 12
               = (4 * 4) + 12
               = 16 + 12
               = 28
```

- call-by-need



2. (a)
```
squareList [] = []
squareList (x:xs) = x*x : squareList xs
```
Guarded Recursion

(b)
```
doubleTimes x 0 = x
doubleTimes x y = doubleTimes (x+x) (y-1)
```
Tail Recursion

(c)
```
add2List [] = 0
add2List (x:xs) = x + add2List xs
```
Linear Recursion (no Tail Recursion)

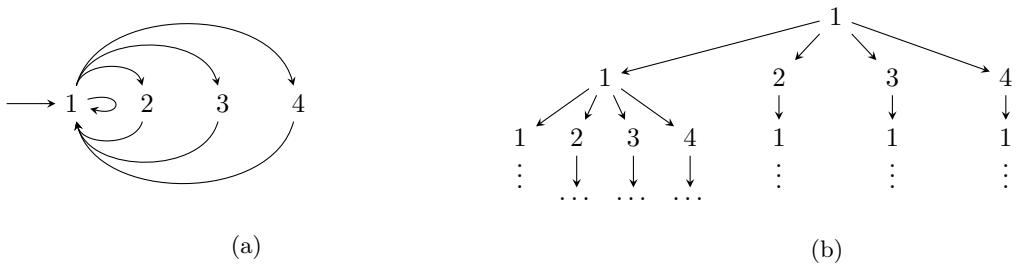
(d)
```
average :: [Double] -> Double
average xs = aux xs 0 (fromIntegral (length xs))
  where aux [] s c = s / c
        aux (x:xs) s c = aux xs (s+x) c
```
Tail Recursion

3.
```
stringToUpperTail :: String -> String
stringToUpperTail = reverse . aux []
  where
    aux acc [] = acc
    aux acc (x:xs) = aux (toUpper x : acc) xs

stringToUpperGuarded :: String -> String
stringToUpperGuarded [] = []
stringToUpperGuarded (x:xs) = toUpper x : stringToUpperGuarded xs
```

Exercise 2 *Lazyness and Infinite Data Structures* **5 p.**

A rooted graph consists of a set of edges between nodes – of the form (source, target) – and additionally has a distinguished node called root. For instance, Figure 1a contains a rooted graph with distinguished node 1 and edges $\{(1,1),(1,2),(1,3),(1,4),(2,1),(3,1),(4,1)\}$.
One way of representing (possibly infinite) rooted graphs is to use (possibly infinite) trees, the so-called *unwinding* of a graph. For example the rooted graph of Figure 1a can be represented by the unwinding shown in Figure 1b.



(a) (b)

Figure 1: A graph and its unwinding

In this exercise graphs and (infinite) trees are represented by the following Haskell type definitions:

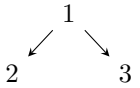
```
type Graph a = [(a, a)]
type RootedGraph a = (a, Graph a)
data Tree a = Node a [Tree a] deriving (Eq, Show)
```

1. Implement a function `unwind :: Eq a => RootedGraph a -> Tree a` that converts a rooted graph into its tree representation. (1 point)

2. Implement a function `prune :: Int -> Tree a -> Tree a` such that `prune n t` results in a pruned tree where only the first `n` layers of the input tree are present. For example invoking `prune 2` on the infinite tree in Figure 1b drops all parts that are depicted by … and ⋮, and `prune 0` would return a tree that just contains the root node 1.

   Consider the tree that results from unwinding the rooted graph `(z, [(x,z), (z,x), (x,y), (y,x)])`, a figure of eight: 

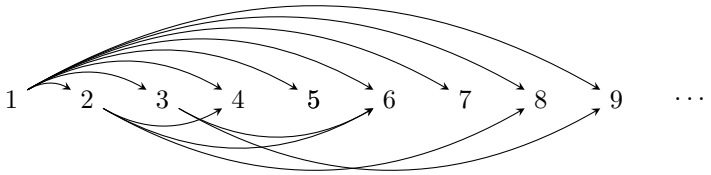
. What is the result of `prune 4` on this tree? (1 point)

3. Implement a function `narrow :: Int -> Tree a -> Tree a` that restricts the number of successors for each node of a tree to a given maximum (by dropping any surplus successors). For example, when calling the function `narrow 1` on the tree 

, the result would be the tree 

. (1 point)

4. Define an infinite tree `mults :: Tree Integer` that represents the graph where every natural number, starting from 1 points to all its multiples: (1 point)



5. Describe the results of evaluating each of the following three expressions: `narrow 4 $ prune 2 mults`, `narrow 1 mults`, and `prune 1 mults`. (1 point)

Solution 2

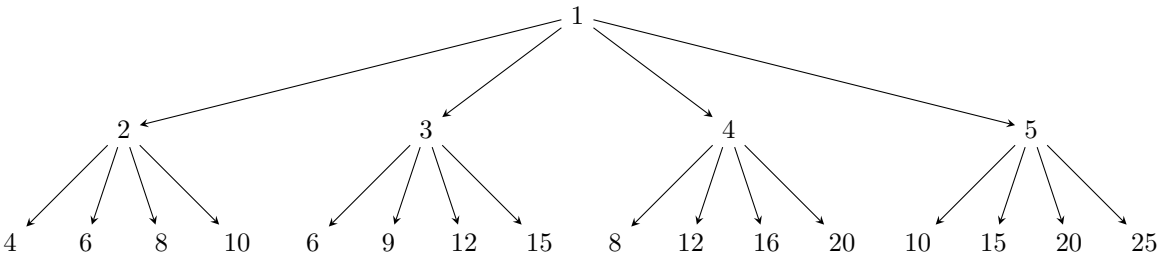
1.
```
unwind :: Eq a => RootedGraph a -> Tree a
unwind (n, g) = Node n (map (\ s -> unwind (s, g)) successors)
  where successors = map snd (filter ( (== n) . fst) g)
```

2.
```haskell
prune :: Int -> Tree a -> Tree a
prune 0 (Node x ts) = Node x []
prune n (Node x ts) = Node x (map (prune (n - 1)) ts)
```

3.
```haskell
narrow :: Int -> Tree a -> Tree a
narrow n (Node x ts) = Node x $ map (narrow n) $ take n ts
```

4.
```haskell
mults :: Tree Integer
mults = go 1
  where go i = Node i $ map go $ map (i*) [2..]
```

5.
- The expression `narrow 4 $ prune 2 mults`



- The expression `narrow 1 mults` yields an infinite tree that structurally resembles the infinite list of powers of 2: $1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 8 \longrightarrow 16 \longrightarrow 32 \longrightarrow \cdots$
- The expression `prune 1 mults` yields the following infinite tree: