Programmierung WS 18
Hausaufgaben - Blatt 9

Julian Giesen (MNR 388487)
Levin Gäher (MNR 395035)
Gruppe 12

HA 2

```
-- a)
data VariableName = X | Y deriving Show

getValue :: VariableName -> Int
getValue X = 5
getValue Y = 13

-- b)
data Expression = Constant Int | Variable VariableName | Add Expression Expression |
    Multiply Expression Expression deriving Show

-- c)
evaluate :: Expression -> Int
evaluate (Constant c) = c
evaluate (Variable v) = getValue v
evaluate (Add ex1 ex2) = (evaluate ex1) + (evaluate ex2)
evaluate (Multiply ex1 ex2) = (evaluate ex1) * (evaluate ex2)

-- d)
tryOptimize :: Expression -> Expression
tryOptimize (Add (Constant c1) (Constant c2)) = Constant (c1 + c2)
tryOptimize (Multiply (Constant c1) (Constant c2)) = Constant (c1 * c2)
tryOptimize ex = ex

-- e)
evaluatePartially :: Expression -> Expression
evaluatePartially (Add ex1 ex2) = tryOptimize (Add (evaluatePartially ex1)
    (evaluatePartially ex2))
evaluatePartially (Multiply ex1 ex2) = tryOptimize (Multiply (evaluatePartially ex1)
    (evaluatePartially ex2))
evaluatePartially ex = ex

-- Example Provided
exampleExpression = Add
                ( Add
                   ( Constant 20)
                   ( Constant 17))
                ( Add
                   ( Variable X )
                   ( Multiply
                      ( Add
                         ( Constant 14)
                         ( Constant 7))
                      ( Constant 2)))
```

HA 4

f :: a - a - [a] - [a] g :: [a] - Int - [Int] - [Int] h :: (a - a) - a - [a] - [a] // List of f applies to ys and then x at the end

HA 6

```
-- a)
data Optional a = Empty | Present a deriving Show

mapOptional :: (a -> b) -> Optional a -> Optional b
mapOptional _ Empty = Empty
mapOptional f (Present o) = Present (f o)

-- b)
filterOptional :: (a -> Bool) -> Optional a -> Optional a
filterOptional f Empty = Empty
filterOptional f (Present o) = if f o then Present o else Empty

-- c)
foldOptional :: (a -> b) -> b -> Optional a -> b
foldOptional _ x Empty = x
foldOptional f _ (Present o) = f o

-- d)
data Product = Article String Int deriving Show

isHumanEatable :: Product -> Bool
isHumanEatable (Article "Dog Food" _) = False
isHumanEatable _ = True

adjustPrice :: Product -> Product
adjustPrice (Article n p) = if p < 1000 then Article n (p*2) else Article n p

stringify :: Product -> String
stringify (Article n p) = "The Article named '" ++ n ++ "' costs " ++ show(p) ++ "
    Cents"

-- e)
filterHumanEatable :: Product -> Optional Product
filterHumanEatable a = filterOptional isHumanEatable (Present a)

adjustPriceO :: Optional Product -> Optional Product
adjustPriceO a = mapOptional adjustPrice a

stringifyO :: Optional Product -> String
stringifyO a = foldOptional stringify "This article is unavailable" a

toPriceTag :: Product -> String
toPriceTag a = stringifyO (adjustPriceO (filterHumanEatable a))
```

HA 8

a)

```
data Rosebush = Rose | Cut | Stalk (Rosebush) | Fork (Rosebush) (Rosebush) deriving
    Show

generate :: Rosebush
generate = Stalk
   (Fork
      (Stalk
         (Fork
            (Stalk
               generate
            )
            (Stalk
               Rose
            )
         )
      )
      (Stalk
         (Stalk
            (Fork
               (Stalk
                  Rose
               )
               (Stalk
                  (Stalk
                     (Stalk
                     generate
                     )
                  )
               )
            )
         )
      )
   )
```

b)

```
   cut :: Int -> Rosebush -> Rosebush
   cut 0 _ = Cut
   cut _ Cut = Cut
   cut _ Rose = Rose
   cut n (Stalk s) = Stalk (cut(n-1) s)
   cut n (Fork x y) = Fork (cut(n-1) x) (cut(n-1) y)


   --given
   spaces :: Int -> String
   spaces 0 = ""
   spaces n = " " ++ spaces (n-1)

   dashes :: Int -> String
   dashes 0 = ""
   dashes n = "-" ++ dashes (n-1)

   zipOutput :: [String] -> Int -> [String] -> Int -> [String]
   zipOutput [] _ [] _         = []
   zipOutput (x:xs) n1 [] n2  = (x ++ " " ++ (spaces n2)):(zipOutput xs n1 [] n2)
   zipOutput [] n1 (y:ys) n2  = ((spaces n1) ++ " " ++ y):(zipOutput [] n1 ys n2)
   zipOutput (x:xs) n1 (y:ys) n2 = (x ++ " " ++ y):(zipOutput xs n1 ys n2)
```

```
data Output = Output [String] Int Int

display :: Rosebush -> Output
display Rose = Output ["*"] 1 1
display Cut = Output ["X"] 1 1
display (Stalk p) = Output (line:o) i n
where line = (spaces (i-1)) ++ "|" ++ (spaces (n-i))
Output o i n = display p
display (Fork p1 p2) = Output (line:(zipOutput o1 n1 o2 n2)) (n1+1) (n1+1+n2)
where line = (spaces (i1-1)) ++ "+" ++ (dashes (n1-i1)) ++ "+"
++ (dashes (i2-1)) ++ "+" ++ (spaces (n2-i2))
Output o1 i1 n1 = display p1
Output o2 i2 n2 = display p2

concatenate :: [String] -> String
concatenate []   = ""
concatenate (x:xs) = (concatenate xs) ++ x ++ "\n"

showMe rosebush = putStr (concatenate o)
where Output o _ _ = display rosebush
```