KØBENHAVNS UNIVERSITET

Advanced Programming

AP20 Assignment0 Groups 26

Assignment 0

csb847, hcx121

Datalogisk Institut9. september 2020

1 Design and Implementation

1.1 Simple arithmetic expressions

Here for convenience we add a pair of parentheses for each output we get from function **showExp** except for a positive x in $Cst\ x$.

Unless we ask **showExp** or **evalSimple** to process some undefined arithmetic expression, which would be reported by an error message, they would abort with the relevant Haskell runtime error when encounter errors.

1.2 Extended arithmetic expressions

For Let-expressions in **evalFull** function, if a variable is not used, we would just ignore the calculation of the value of this variable because of the lazy evaluation. Such implementation is simple and since we don't need the value, the errors in the calculation of the value should not affect the process of the whole expression, and we can get an answer without this variable.

We get a Maybe Integer from Var-expression, however during calculation we operate on Integers. So we implemented a function **getJust::Maybe Integer -> Integer** to extract Integer x from Maybe Integer $Just\ x$.

1.3 Returning explicit errors

Now we get an Either ArithError Integer value from the **evalErr** function, however during the evaluation process, we need to operate on Integer values, thus we implement a **getEither::Either ArithError Integer -> Integer** function to extract Integer value x from $Rightx$.

When there are two parameters x, y in the expression like in Div-expressions, Pow-expressions and so on, we have to check whether the expressions of these two parameters can be processed properly, if not, we do not do the calculation, but pass the error message, else we can do the calculation. To avoid repetitive work, we implement a function called **checkEither::Either ArithError Integer -> Bool**, it is used to check if the expression returns a Right value or a Left value, if Right, it returns True, else returns False. So if *checkEither x* and *checkEither y* are both True, we can proceed to the calculation in this expression, else we just pass the error message on.

When passing the error message, we have to know which parameter gets a Left value so that we can pass it on, and to avoid repetitive work, we implemented **showEither::Either ArithError Integer -> Either ArithError Integer -> Either ArithError Integer** to decide the error message. If the first parameter has an error message then we just pass it on and ignore the second one, else we pass the error message from the second parameter.

2 Assessment of the code

2.1 Simple arithmetic expressions

For showExp and evalSimple, we used 5 testing examples separately. For *showExp* we chose several polynomials. It turned out to be work well in these five cases.
For *evalSimple*,we also chose several polynomials,and tested if it is vaild with big number result.

2.2 Extended arithmetic expressions

In 2.2 we tested extendEnv and evalFull. For extendEnv function, we chose 5 examples to test if it can extend from an empty Env in different cases And for evalFull's testing, we chose and different environments, including extended and init environment.

2.3 Returning explicit errors

In this section, we tested evalErr function's output in different error cases.We also tested all subexpressions are to be evaluated left-toright.

2.4 Apart from the tests in the Test.hs

We also used online TA to assess our code and only fail in the test $Pow(Div\ (Cst\ 4)\ (Cst\ 0))\ (Cst\ 0)$ and $Pow\ (Pow\ (Cst\ 2)\ (Cst\ (-1)))\ (Cst\ 0)$ because of the calculation order.

For error cases, we tested $evalSimple(Div\ (Cst\ 1)(Cst\ 0))$ in ghci and get a result of an exception '*** Exception: divide by zero'; we also tested $evalSimple(Pow\ (Cst\ 1)(Cst\ (-1)))$ in ghci and get the result '*** Exception: Negative exponent'. For undefined arithmetic expression, we tested $showExp\ (Add1\ (Cst\ 1)\ (Cst\ 2))$ and $evalSimple\ (Add1\ (Cst\ 1)\ (Cst\ 2))$ and get errors. The $evalFull$ function also works well in similar tests.

In summary, we think our code can get a 9/10 score.

Appendix

Warmup.hs

```
module Warmup where

type Pos = (Int, Int)
data Direction = North | South | East | West

move :: Direction -> Pos -> Pos
move North (x,y) = (x, y+1)
move West  (x,y) = (x-1, y)
move South (x,y) = (x, y-1)
move East (x,y) = (x+1, y)


moves :: [Direction] -> Pos -> Pos
moves [] (x,y) = (x,y)
moves a (x,y) = moves (tail a) (move (head a) (x,y))

data Nat = Zero | Succ Nat
  deriving (Eq, Show, Read, Ord)

add :: Nat -> Nat -> Nat
add x Zero = x
add x (Succ y) = add (Succ x) y

mult :: Nat -> Nat -> Nat
mult Zero y = Zero
mult (Succ x) y = add (mult x y) y

-- Do not use these to define add/mult!
nat2int :: Nat -> Int
nat2int Zero = 0::Int
nat2int (Succ x) = nat2int x + 1::Int

int2nat :: Int -> Nat
int2nat 0 = Zero::Nat
int2nat x = Succ (int2nat (x-1))

data Tree = Leaf | Node Int Tree Tree
  deriving (Eq, Show, Read, Ord)

insert :: Int -> Tree -> Tree
insert x Leaf = Node x Leaf Leaf
insert x (Node a t1 t2)
  | x == a = Node a t1 t2
  | x < a = Node a (insert x t1) t2
  | x > a = Node a t1 (insert x t2)
```

Arithmetic.hs

```
-- This is a skeleton file for you to edit

{-# OPTIONS_GHC -W #-} -- Just in case you forgot...

module Arithmetic
  (
  showExp,
  evalSimple,
  extendEnv,
  evalFull,
  evalErr,
  showCompact,
  evalEager,
  evalLazy
  )

where

import Definitions

showExp :: Exp -> String
showExp (Cst x)
  | x >= 0 = show x
  | otherwise = "(" ++ show x ++ ")"
showExp (Add x y) = "(" ++ showExp x ++ "+" ++ showExp y ++ ")"
showExp (Sub x y) = "(" ++ showExp x ++ "-" ++ showExp y ++ ")"
showExp (Mul x y) = "(" ++ showExp x ++ "*" ++ showExp y ++ ")"
showExp (Div x y) = "(" ++ showExp x ++ "/" ++ showExp y ++ ")"
showExp (Pow x y) = "(" ++ showExp x ++ "^" ++ showExp y ++ ")"
showExp _ = error "You can only use Cst, Add, Mul, Div and Pow here in showExp."

evalSimple :: Exp -> Integer
evalSimple (Cst x) = x
evalSimple (Add x y) = evalSimple x + evalSimple y
evalSimple (Sub x y) = evalSimple x - evalSimple y
evalSimple (Mul x y) = evalSimple x * evalSimple y
evalSimple (Div x y) = (evalSimple x) `div` evalSimple y
evalSimple (Pow x y) = (evalSimple x) ^ evalSimple y
evalSimple _ = error "You can only use Cst, Add, Mul, Div and Pow here in evalSimple."

extendEnv :: VName -> Integer -> Env -> Env
extendEnv v n r = \x->if x == v then Just n else r x

-- Extract the Interger from (Just Interger) expression.
getJust :: Maybe Integer -> Integer
getJust (Just x) = x
getJust _ = undefined

-- Extract the value from (Right Integer).
-- Only value checked by checkEither would use this function.
-- Thus there is only Right value here.
getEither :: Either ArithError Integer -> Integer
getEither (Right x) = x

-- Check if the Either expression is like (Right something).
checkEither :: Either ArithError Integer -> Bool
checkEither (Right _) = True
checkEither (Left _) = False

-- Pass the error message.
-- If the first expression returns (Left somthing), then return it,
-- else return the result of the second expression.
showEither :: Either ArithError Integer -> Either ArithError Integer -> Either ArithError
    Integer
showEither a b =
  if not (checkEither a)
      then a
    else b

evalFull :: Exp -> Env -> Integer
evalFull (Cst x) _ = x
evalFull (Var v) r = if r v == Nothing then error "Variable is not bound." else getJust (r v)
evalFull (Add x y) r = (evalFull x r) + (evalFull y r)
evalFull (Sub x y) r = (evalFull x r) - (evalFull y r)
evalFull (Mul x y) r = (evalFull x r) * (evalFull y r) -- if the fst one is 0, don't need to
    calculate the snd one?
evalFull (Div x y) r = (evalFull x r) `div` (evalFull y r)
evalFull (Pow x y) r = (evalFull x r) ^ (evalFull y r)
-- Add () to all x, y to make it more clear.
evalFull (If e1 e2 e3) r
  | evalFull e1 r == 0 = evalFull e3 r
  | otherwise = evalFull e2 r
evalFull (Let v e1 e2) r = evalFull e2 (extendEnv v (evalFull e1 r) r)
evalFull (Sum v e1 e2 e3) r
  | evalFull e1 r > evalFull e2 r = 0
  | otherwise = (evalFull e3 (extendEnv v (evalFull e1 r) r)) + (evalFull (Sum v (Add e1 (Cst
      1)) e2 e3) r) -- the snd don't need a extendEnv because v is covered later
evalFull _ _ = error "Undefined arithmetic operations is called in evalFull."

evalErr :: Exp -> Env -> Either ArithError Integer
evalErr (Cst x) _ = Right x

evalErr (Var v) r =
  if r v == Nothing
    then Left (EBadVar v)
  else Right (getJust (r v))

evalErr (Add x y) r =
  if checkEither (evalErr x r) && checkEither (evalErr y r)
    then Right $ getEither (evalErr x r) + getEither (evalErr y r)
  else
    showEither (evalErr x r) (evalErr y r)

evalErr (Sub x y) r =
  if checkEither (evalErr x r) && checkEither (evalErr y r)
    then Right $ getEither (evalErr x r) - getEither (evalErr y r)
  else
    showEither (evalErr x r) (evalErr y r)

evalErr (Mul x y) r =
  if checkEither (evalErr x r) && checkEither (evalErr y r)
   then Right $ getEither (evalErr x r) * getEither (evalErr y r)
  else
    showEither (evalErr x r) (evalErr y r)
```

```
evalErr (Div x y) r =
  if checkEither (evalErr x r) && checkEither (evalErr y r)
    then
      if getEither (evalErr y r) == 0
        then Left EDivZero
      else Right $ (getEither (evalErr x r)) `div` (getEither (evalErr y r))
  else
    showEither (evalErr x r) (evalErr y r)

evalErr (Pow x y) r =
  if checkEither (evalErr x r) && checkEither (evalErr y r)
    then
      if getEither (evalErr y r) < 0
        then Left ENegPower
      else Right $ (getEither(evalErr x r)) ^ (getEither(evalErr y r))
  else
    showEither (evalErr x r) (evalErr y r)

evalErr (If e1 e2 e3) r =
  if checkEither (evalErr e1 r)
    then
      if getEither (evalErr e1 r) == 0
        then
          evalErr e3 r
      else
        evalErr e2 r
  else
    evalErr e1 r

evalErr (Let v e1 e2) r =
  if checkEither (evalErr e1 r)
    then evalErr e2 (extendEnv v (getEither(evalErr e1 r)) r)
  else
    evalErr e1 r

evalErr (Sum v e1 e2 e3) r =
  if checkEither (evalErr e1 r) && checkEither (evalErr e2 r)
    then
      if getEither (evalErr e1 r) > getEither (evalErr e2 r)
        then Right 0
      else
        if checkEither (evalErr e3 (extendEnv v (getEither(evalErr e1 r)) r)) && checkEither (
            evalErr (Sum v (Add e1 (Cst 1)) e2 e3) r)
          then Right $ (getEither (evalErr e3 (extendEnv v (getEither(evalErr e1 r)) r))) + (
              getEither (evalErr (Sum v (Add e1 (Cst 1)) e2 e3) r))
        else
          showEither (evalErr e3 (extendEnv v (getEither(evalErr e1 r)) r)) (evalErr (Sum v (
              Add e1 (Cst 1)) e2 e3) r)
  else
    showEither (evalErr e1 r) (evalErr e2 r)



-- optional parts (if not attempted, leave them unmodified)

showCompact :: Exp -> String
showCompact = undefined
```

```
evalEager :: Exp -> Env -> Either ArithError Integer
evalEager = undefined

evalLazy :: Exp -> Env -> Either ArithError Integer
evalLazy = undefined
```

Test.hs

```
-- Very rudimentary test of Arithmetic. Feel free to replace completely

import Definitions
import Arithmetic

import Data.List (intercalate)
import System.Exit (exitSuccess, exitFailure)  -- for when running stand-alone


--1.1
tests :: [(String, Bool)]
tests = [test1, test2, test3,test4,test5,test6,test7,test8,test9,test10,test11,test12,test13,
    test14,test15,test16,test17,test18,test19,test20,test21,test22,test23,test24,test25] where
  --1.1
  test1 = ("test1",showExp(Div (Cst 0) (Cst 1))=="(0/1)" )
  test2 = ("test2",showExp(Add (Mul (Cst 2) (Cst 3)) (Cst 4)) =="((2*3)+4)" )
  test3 = ("test3",showExp(Pow (Div (Cst 2) (Cst 3)) (Sub (Cst 4) (Cst 5)))=="((2/3)^(4-5))" )
  test4 = ("test4",showExp(Add (Sub (Cst 2) (Cst 3)) (Cst 4)) =="((2-3)+4)")
  test5 = ("test5",showExp(Sub (Cst 2) (Add (Cst 3) (Cst 4))) =="(2-(3+4))")
  --1.2
  test6 = ("test6",evalSimple(Mul (Add (Cst 2) (Cst 3)) (Cst 4)) ==20)
  test7 = ("test7",evalSimple(Div (Mul (Cst 2) (Cst 3)) (Cst 4)) ==1)
  test8 = ("test8",evalSimple(Div (Cst 0) (Cst 1))==0 )
  test9 = ("test9",evalSimple(Pow (Cst 2) (Pow (Cst 3) (Cst 4)))==2417851639229258349412352
      )
  test10 = ("test10",evalSimple(Pow (Pow (Cst 2) (Cst 3)) (Cst 4))==4096 )
  --2.1
  test11 = ("test11",(extendEnv "x" 5 initEnv) "x"==Just 5 )
  test12 = ("test12",(extendEnv "x" 5 initEnv) "y"==Nothing )
  test13 = ("test13",(extendEnv "x" 5 (extendEnv "y" 6 initEnv) "x")==Just 5 )
  test14 = ("test14",(extendEnv "x" 5 (extendEnv "y" 6 initEnv) "y")==Just 6 )
  test15 = ("test15",(extendEnv "x" 5 (extendEnv "x" 6 initEnv) "x")==Just 5 )
  --2.2

  test16 = ("test16", evalFull (Div (Cst 4) (Cst 1)) initEnv==4 )
  test17 = ("test17", evalFull(Pow (Div (Cst 4) (Cst 0)) (Cst 0))initEnv== 1 )
  test18 = ("test18", evalFull(Mul (Cst 1) (Div (Cst 0) (Cst 1)))initEnv== 0 )
  test19 = ("test19", evalFull(If (Sub (Cst 3) (Cst 3)) (Cst 4) (Cst 5))initEnv== 5)
  test20 =  ("test20", evalFull(Var "x")(extendEnv "x" 5 initEnv)==5)

  --3.1

  test21 = ("test21",evalErr (Sum "x" (Sub (Cst 3) (Cst 2)) (Add (Cst 3) (Cst 2)) (Var "x"))
      initEnv==Right 15 )
  test22 = ("test22",evalErr ( Let "x" (Let "y" (Cst 3) (Sub (Var "x") (Var "y"))) (Mul (Var "x
      ") (Var "y")))initEnv==Left (EBadVar "x") )
  test23 = ("test23",evalErr ( Let "z" (Add (Cst 2) (Cst 3)) (Var "z"))initEnv== Right 5)
  test24 = ("test24",evalErr (Let "x" (Add (Cst 3) (Var "y")) (Var "y"))initEnv==Left (EBadVar
      "y") )
  test25 = ("test25",evalErr (Let "x" (Add (Cst 3) (Var "y")) (Let "y" (Mul (Var "x") (Cst 2)) (
      Var "x")))initEnv==Left (EBadVar "y") )



main :: IO ()
main =
  let failed = [name | (name, ok) <- tests, not ok]
```

```
in case failed of
       []  −> do putStrLn "All tests passed!"
                 exitSuccess
       _ −> do putStrLn $ "Failed tests: " ++ intercalate ", " failed
                exitFailure
```