Haskell intro

Assignement0

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 12, 2018

Contents

1 Design/Implementation

We always tried to move as much code as possible to own functions that the code doesn't get too unreadable. For Example the function ‘showExpr‘ would have had too much duplicated code which we then refactored out. Also the function ‘summ‘ is a good example for making it more readable by moving the functionality out of ‘evalFull‘ and only call the function from there. This also helps with reusability overall in the code in case one of the functions can be used many times (like ‘summ‘ in ‘evalFull‘ and ‘evalError‘).

Additionally we also didn't check for division by zero in ‘evalSimple‘ or ‘evalFull‘ since haskell takes care of those errors.

We also tried to use as much as possible out of the standard library for **Either** like ‘isRight‘ otherwise we would have had to implement it ourselves (or end up doing something totally different). For ‘fromRight‘ we decided to implement it ourselves as ‘fromRight'‘ in which we don't have to pass any default value and return an error in case it's not a ‘Right Either‘.

Overall we tried to keep it as simple as possible and declutter code wherever it was possible.

2 Code Assessment

By moving some functionality into own functions we do believe we increased the maintainability at least in some parts, especially when you don't have to change the code in many places.

We also try to handle all kind of edge cases as good as possible that the code should be able to handle errors or wrong inputs (either by error message or haskell error depending on which eval function).

Additionally we sometimes ended up with long lines (longer then 80 Chars) which might not seem that nice sometimes but for the sake of having a one line solution it was a necessary evil.

Sadly we didn't write any tests for our code, which might support our assessment even more.

But we did test it via the 'onlineta' which tests already lots of cases (see Appendix).

A Code Listing

```haskell
-- 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
import Data.Either

-- Exercise 1.1
-- Helper to make it nicer to print
```

```haskell
showExpStr :: Exp -> Exp -> String -> String
showExpStr a b s = "(" ++ showExp a ++ s ++ showExp b ++ ")"

showExp :: Exp -> String
showExp (Cst as) =
  if head(show as) == '-' then "(" ++ show as ++ ")" else show as
showExp (Add a b) = showExpStr a b " + "
showExp (Sub a b) = showExpStr a b " - "
showExp (Mul a b) = showExpStr a b " * "
showExp (Div a b) = showExpStr a b " / "
showExp (Pow a b) = showExpStr a b "^"
showExp _ = error "is not supported"

-- Exercise 1.2
evalSimple :: Exp -> Integer
evalSimple (Cst a) = a
evalSimple (Add a b) = evalSimple a + evalSimple b
evalSimple (Sub a b) = evalSimple a - evalSimple b
evalSimple (Mul a b) = evalSimple a * evalSimple b
-- div checks it self i b is zero
evalSimple (Div a b) = evalSimple a `div` evalSimple b
-- check ourselvs for negative exponent
-- and run a first with seq to se that there is nothing illegal there
evalSimple (Pow a b)
  | evalSimple b < 0 = error "Negative exponent"
  | otherwise = seq (evalSimple a) (evalSimple a ^ evalSimple b)
evalSimple _ = error "is not supported"

-- Exercise 2
extendEnv :: VName -> Integer -> Env -> Env
extendEnv v n r a = if v == a then Just n else r a

-- used to check if variable is unbound
intTest :: Maybe Integer -> Integer
intTest (Just i) = i
intTest _ = error "variable is unbound"

-- helper to calculate sum
-- takes integers instead of expressions
summ :: VName -> Integer -> Integer -> Exp -> Env -> Integer
summ v a b c r = if a > b then 0 else
  evalFull c r + summ v (a+1) b c (extendEnv v (a+1) r)

evalFull :: Exp -> Env -> Integer
evalFull (Cst a) _ = a
evalFull (Add a b) r = evalFull a r + evalFull b r
evalFull (Sub a b) r = evalFull a r - evalFull b r
evalFull (Mul a b) r = evalFull a r * evalFull b r
evalFull (Div a b) r = evalFull a r `div` evalFull b r
-- check for negative exponent
evalFull (Pow a b) r
  | evalFull b r < 0 = error "Negative exponent"
  | otherwise = seq (evalFull a r) (evalFull a r ^ evalFull b r)
-- check if a is zero
evalFull (If a b c) r =
  if evalFull a r /= 0 then evalFull b r else evalFull c r
evalFull (Var v) r = intTest(r v)
evalFull (Let a b c) r = evalFull c (extendEnv a (evalFull b r)
 ↪ r)
evalFull (Sum v a b c) r =
  summ v (evalFull a r) (evalFull b r) c (extendEnv v (evalFull a
 ↪ r) r)

-- Exercise 3
intTestErr :: Maybe Integer -> VName -> Either ArithError Integer
intTestErr (Just i) _ = Right i
intTestErr _ v = Left (EBadVar v)

evalErr :: Exp -> Env -> Either ArithError Integer
evalErr (Cst a) _ = Right a
evalErr (Add a b) r = evalEither (evalErr a r) (+) (evalErr b r)
evalErr (Sub a b) r = evalEither (evalErr a r) (-) (evalErr b r)
evalErr (Mul a b) r = evalEither (evalErr a r) (*) (evalErr b r)
-- check for division by zero
evalErr (Div a b) r = if isRight (evalErr b r)
                        then if fromRight' (evalErr b r) /= 0
                          then evalEither (evalErr a r) div
 ↪ (evalErr b r)
                          else Left EDivZero
                        else evalErr b r
-- check for negative exponent
evalErr (Pow a b) r = if isRight (evalErr b r)
                        then if fromRight' (evalErr b r) >= 0
                          then evalEither (evalErr a r) (^)
 ↪ (evalErr b r)
                          else Left ENegPower
                        else evalErr b r
```

```haskell
-- check if a is zero
evalErr (If a b c) r = if isRight (evalErr a r)
                          then if fromRight' (evalErr a r) /= 0
                            then evalErr b r
                            else evalErr c r
                        else evalErr a r
evalErr (Var v) r = intTestErr (r v) v
evalErr (Let a b c) r = if isRight (evalErr b r)
                          then evalErr c (extendEnv a (fromRight'(evalErr b r)) r)
                          else evalErr b r

evalErr (Sum v a b c) r = if isRight (evalErr a r)
                            then if isRight (evalErr b r)
                              then Right (summ v (fromRight' (evalErr a r)) (fromRight' (evalErr b r)) c (extendEnv v (fromRight'(evalErr a r)) r))
                              else evalErr b r
                            else evalErr a r

evalEither :: Either a b -> (b -> b -> b) -> Either a b -> Either a b
evalEither a b c = if isRight a
                        then if isRight c
                          then Right ( b (fromRight' a) (fromRight' c))
                          else c
                        else a

-- use own implementation of fromRight from Data.Either but not returning a
-- default value, which is not needed for the assignment
fromRight' :: Either a b -> b
fromRight' (Right c) = c
fromRight' _ = error "No value"

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

showCompact :: Exp -> String
showCompact = undefined

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

```haskell

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

B Test Results Online TA

showExpr	Result
Mul (Add (Cst 2) (Cst 3)) (Cst 4)	OK
Add (Mul (Cst 2) (Cst 3)) (Cst 4)	OK
Pow (Div (Cst 2) (Cst 3)) (Sub (Cst 4) (Cst 5))	OK
Add (Sub (Cst 2) (Cst 3)) (Cst 4)	OK
Sub (Cst 2) (Add (Cst 3) (Cst 4))	OK
Div (Mul (Cst 2) (Cst 3)) (Cst 4)	OK
Mul (Cst 2) (Div (Cst 3) (Cst 4))	OK
Pow (Cst 2) (Pow (Cst 3) (Cst 4))	OK
Pow (Pow (Cst 2) (Cst 3)) (Cst 4)	OK
Cst 0	OK
Cst 3	OK
Cst (-3)	OK
Add (Cst (-3)) (Cst (-4))	OK
Mul (Cst (-3)) (Cst (-4))	OK
Pow (Cst (-3)) (Cst (-4))	OK

evalSimple	Result
Cst 3	OK
Cst 12345678901234567890	OK
Add (Cst 3) (Cst 5)	OK
Sub (Cst 3) (Cst 5)	OK
Mul (Cst 3) (Cst 5)	OK
Mul (Cst 1234567890) (Cst 1234567890)	OK
Div (Cst 12) (Cst 3)	OK
Div (Cst (-12)) (Cst 3)	OK
Div (Cst 10) (Cst 3)	OK
Div (Cst (-10)) (Cst 3)	OK
Div (Cst 10) (Cst (-3))	OK
Div (Cst (-10)) (Cst (-3))	OK
Pow (Cst 3) (Cst 5)	OK

Pow (Cst (-3)) (Cst 5)	OK
Pow (Cst (-3)) (Cst 0)	OK
Pow (Cst 0) (Cst 0)	OK
*Div (Cst 4) (Cst 0)	OK
*Pow (Cst 4) (Cst (-1))	OK
Mul (Add (Cst 2) (Cst 3)) (Cst 4)	OK
Pow (Cst 2) (Mul (Cst 3) (Cst 4))	OK
*Mul (Cst 0) (Div (Cst 0) (Cst 0))	OK
*Pow (Pow (Cst 2) (Cst (-1))) (Cst 0)	OK

extendEnv	Result
(extendEnv ”x” 5 initEnv) ”x”	OK
(extendEnv ”x” 5 initEnv) ”y”	OK
(extendEnv ”x” 5 (extendEnv ”y” 6 initEnv)) ”x”	OK
(extendEnv ”x” 5 (extendEnv ”y” 6 initEnv)) ”y”	OK
(extendEnv ”x” 5 (extendEnv ”y” 6 initEnv)) ”z”	OK
(extendEnv ”x” 5 (extendEnv ”x” 6 initEnv)) ”x”	OK

evalFull	Result
Cst 3	OK
Cst 12345678901234567890	OK
Add (Cst 3) (Cst 5)	OK
Sub (Cst 3) (Cst 5)	OK
Mul (Cst 3) (Cst 5)	OK
Mul (Cst 1234567890) (Cst 1234567890)	OK
Div (Cst 12) (Cst 3)	OK
Div (Cst (-12)) (Cst 3)	OK
Div (Cst 10) (Cst 3)	OK
Div (Cst (-10)) (Cst 3)	OK
Div (Cst 10) (Cst (-3))	OK
Div (Cst (-10)) (Cst (-3))	OK
Pow (Cst 3) (Cst 5)	OK
Pow (Cst (-3)) (Cst 5)	OK
Pow (Cst (-3)) (Cst 0)	OK
Pow (Cst 0) (Cst 0)	OK
*Div (Cst 4) (Cst 0)	OK
*Pow (Cst 4) (Cst (-1))	OK
Mul (Add (Cst 2) (Cst 3)) (Cst 4)	OK

Pow (Cst 2) (Mul (Cst 3) (Cst 4))	OK
*Mul (Cst 0) (Div (Cst 0) (Cst 0))	OK
*Pow (Pow (Cst 2) (Cst (-1))) (Cst 0)	OK
If (Cst 1) (Cst 4) (Cst 5)	OK
If (Cst (-3)) (Cst 4) (Cst 5)	OK
If (Sub (Cst 3) (Cst 3)) (Cst 4) (Cst 5)	OK
If (Cst 2) (Cst 5) (Div (Cst 7) (Cst 0))	OK
If (Cst 0) (Div (Cst 7) (Cst 0)) (Cst 5)	OK
Var ”x”	OK
Var ”y”	OK
*Var ”z”	OK
Let ”z” (Add (Cst 2) (Cst 3)) (Var ”z”)	OK
Let ”z” (Add (Cst 2) (Cst 3)) (Pow (Var ”z”) (Var ”z”))	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Var ”x”)	OK
Let ”x” (Add (Cst 3) (Var ”x”)) (Var ”x”)	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Var ”y”)	OK
Mul (Var ”x”) (Let ”x” (Cst 10) (Var ”x”))	OK
Mul (Let ”x” (Cst 10) (Var ”x”)) (Var ”x”)	OK
*Mul (Let ”z” (Cst 10) (Var ”z”)) (Var ”z”)	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Let ”y” (Mul (Var ”x”) (Cst 2)) (Var ”x”))	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Let ”y” (Mul (Var ”x”) (Cst 2)) (Var ”y”))	OK
Let ”x” (Let ”y” (Cst 3) (Sub (Var ”x”) (Var ”y”))) (Mul (Var ”x”) (Var ”y”))	OK
Sum ”x” (Sub (Cst 3) (Cst 2)) (Add (Cst 3) (Cst 2)) (Var ”x”)	OK
Sum ”x” (Cst 1) (Cst 5) (Pow (Var ”x”) (Cst 2))	OK
Sum ”x” (Cst 10) (Add (Cst 5) (Cst 5)) (Mul (Cst 3) (Var ”x”))	OK
Sum ”x” (Cst 11) (Add (Cst 5) (Cst 5)) (Var ”x”)	OK
Sum ”x” (Cst 12) (Add (Cst 5) (Cst 5)) (Div (Var ”x”) (Cst 0))	OK
Sum ”x” (Cst 123456789012345) (Cst 0) (Cst 1)	OK
Sum ”x” (Cst 1) (Var ”x”) (Let ”x” (Add (Var ”x”) (Cst 1)) (Var ”x”))	OK
Sum ”x” (Cst 1) (Var ”x”) (Sum ”x” (Var ”x”) (Cst 10) (Var ”x”))	OK
*Add (Var ”b1”) (Var ”b2”)	OK
*If (Var ”b1”) (Var ”b2”) (Var ”b3”)	OK
*Sum ”x” (Var ”b1”) (Var ”b2”) (Var ”b3”)	OK
*Mul (Div (Cst 3) (Cst 0)) (Pow (Cst 4) (Cst (-1)))	OK

evalErr	Result
Cst 3	OK
Cst 12345678901234567890	OK
Add (Cst 3) (Cst 5)	OK
Sub (Cst 3) (Cst 5)	OK
Mul (Cst 3) (Cst 5)	OK
Mul (Cst 1234567890) (Cst 1234567890)	OK
Div (Cst 12) (Cst 3)	OK
Div (Cst (-12)) (Cst 3)	OK
Div (Cst 10) (Cst 3)	OK
Div (Cst (-10)) (Cst 3)	OK
Div (Cst 10) (Cst (-3))	OK
Div (Cst (-10)) (Cst (-3))	OK
Pow (Cst 3) (Cst 5)	OK
Pow (Cst (-3)) (Cst 5)	OK
Pow (Cst (-3)) (Cst 0)	OK
Pow (Cst 0) (Cst 0)	OK
Div (Cst 4) (Cst 0)	OK
Pow (Cst 4) (Cst (-1))	OK
Mul (Add (Cst 2) (Cst 3)) (Cst 4)	OK
Pow (Cst 2) (Mul (Cst 3) (Cst 4))	OK
Mul (Cst 0) (Div (Cst 0) (Cst 0))	OK
Pow (Pow (Cst 2) (Cst (-1))) (Cst 0)	OK
If (Cst 1) (Cst 4) (Cst 5)	OK
If (Cst (-3)) (Cst 4) (Cst 5)	OK
If (Sub (Cst 3) (Cst 3)) (Cst 4) (Cst 5)	OK
If (Cst 2) (Cst 5) (Div (Cst 7) (Cst 0))	OK
If (Cst 0) (Div (Cst 7) (Cst 0)) (Cst 5)	OK
Var ”x”	OK
Var ”y”	OK
Var ”z”	OK
Let ”z” (Add (Cst 2) (Cst 3)) (Var ”z”)	OK
Let ”z” (Add (Cst 2) (Cst 3)) (Pow (Var ”z”) (Var ”z”))	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Var ”x”)	OK
Let ”x” (Add (Cst 3) (Var ”x”)) (Var ”x”)	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Var ”y”)	OK
Mul (Var ”x”) (Let ”x” (Cst 10) (Var ”x”))	OK
Mul (Let ”x” (Cst 10) (Var ”x”)) (Var ”x”)	OK
Mul (Let ”z” (Cst 10) (Var ”z”)) (Var ”z”)	OK

Let ”x” (Add (Cst 3) (Var ”y”)) (Let ”y” (Mul (Var ”x”) (Cst 2)) (Var ”x”))	OK
Let ”x” (Add (Cst 3) (Var ”y”)) (Let ”y” (Mul (Var ”x”) (Cst 2)) (Var ”y”))	OK
Let ”x” (Let ”y” (Cst 3) (Sub (Var ”x”) (Var ”y”))) (Mul (Var ”x”) (Var ”y”))	OK
Sum ”x” (Sub (Cst 3) (Cst 2)) (Add (Cst 3) (Cst 2)) (Var ”x”)	OK
Sum ”x” (Cst 1) (Cst 5) (Pow (Var ”x”) (Cst 2))	OK
Sum ”x” (Cst 10) (Add (Cst 5) (Cst 5)) (Mul (Cst 3) (Var ”x”))	OK
Sum ”x” (Cst 11) (Add (Cst 5) (Cst 5)) (Var ”x”)	OK
Sum ”x” (Cst 12) (Add (Cst 5) (Cst 5)) (Div (Var ”x”) (Cst 0))	OK
Sum ”x” (Cst 123456789012345) (Cst 0) (Cst 1)	OK
Sum ”x” (Cst 1) (Var ”x”) (Let ”x” (Add (Var ”x”) (Cst 1)) (Var ”x”))	OK
Sum ”x” (Cst 1) (Var ”x”) (Sum ”x” (Var ”x”) (Cst 10) (Var ”x”))	OK
Add (Var ”b1”) (Var ”b2”)	OK
If (Var ”b1”) (Var ”b2”) (Var ”b3”)	OK
Sum ”x” (Var ”b1”) (Var ”b2”) (Var ”b3”)	OK
Mul (Div (Cst 3) (Cst 0)) (Pow (Cst 4) (Cst (-1)))	OK