# Haskell intro

Assignement0

*Kai Arne S. Myklebust, Silvan Adrian*

Handed in: September 21, 2018

## Contents

**1 Design/Implementation** **1**

**2 Code Assessment** **2**

**A Code Listing** **2**

## 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 example from **Data.Either** the ‘isRight‘ function. 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.

**Edit 17. September:** We had to use eager evaluation for the single case of Negative Power Exponent since otherwise Haskell tries to be intelligent and just ignores part of the expression. That's why we use 'seq'to be sure that 'Pow' gets evaluated rightly. The rest is lazy evaluated.

## 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). As said, for example with division by zero haskell catches that error itself, so we did not find it necessary to write our own test for that.

Additionally we sometimes ended up with long lines (longer then 80 Chars) which might not seem that nice, but for the sake of having a one line solution it was a necessary evil. Sum is the worst example in this regard, but it needs so many parameters that need to be checked so it was difficult do to it shorter.

We also wrote some tests which can be run by 'stack test', which test the basic functionality to our best knowledge.

## 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
  )

```

```haskell
where

import Definitions
import Data.Either

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

```haskell

-- 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
```

```haskell
-- 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
-- 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)
    ↪
```

```haskell

showCompact :: Exp -> String
showCompact = undefined

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

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