

Haskell intro

Assignement0

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 18, 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

```
1  -- This is a skeleton file for you to edit
2
3  {-# OPTIONS_GHC -W #-}  -- Just in case you forgot...
4
5  module Arithmetic
6  (
7      showExp,
8      evalSimple,
9      extendEnv,
10     evalFull,
11     evalErr,
12     showCompact,
13     evalEager,
14     evalLazy
15 )
16
```

```

17 where
18
19 import Definitions
20 import Data.Either
21
22 -- Exercise 1.1
23 -- Helper to make it nicer to print
24 showExpStr :: Exp -> Exp -> String -> String
25 showExpStr a b s = "(" ++ showExp a ++ s ++ showExp b ++ ")"
26
27 showExp :: Exp -> String
28 showExp (Cst as) =
29     if head(show as) == '-' then "(" ++ show as ++ ")" else show as
30 showExp (Add a b) = showExpStr a b " + "
31 showExp (Sub a b) = showExpStr a b " - "
32 showExp (Mul a b) = showExpStr a b " * "
33 showExp (Div a b) = showExpStr a b " / "
34 showExp (Pow a b) = showExpStr a b "^"
35 showExp _ = error "is not supported"
36
37 -- Exercise 1.2
38 evalSimple :: Exp -> Integer
39 evalSimple (Cst a) = a
40 evalSimple (Add a b) = evalSimple a + evalSimple b
41 evalSimple (Sub a b) = evalSimple a - evalSimple b
42 evalSimple (Mul a b) = evalSimple a * evalSimple b
43 -- div checks it self i b is zero
44 evalSimple (Div a b) = evalSimple a `div` evalSimple b
45 -- check ourselves for negative exponent
46 -- and run a first with seq to se that there is nothing illegal
47   => there
48 evalSimple (Pow a b)
49   | evalSimple b < 0 = error "Negative exponent"
50   | otherwise = seq (evalSimple a) (evalSimple a ^ evalSimple b)
51 evalSimple _ = error "is not supported"
52
53 -- Exercise 2
54 extendEnv :: VName -> Integer -> Env -> Env
55 extendEnv v n r a = if v == a then Just n else r a
56
57 -- used to check if variable is unbound
58 intTest :: Maybe Integer -> Integer
59 intTest (Just i) = i
60 intTest _ = error "variable is unbound"

```

```

60
61 -- helper to calculate sum
62 -- takes integers instead of expressions
63 summ :: VName -> Integer -> Integer -> Exp -> Env -> Integer
64 summ v a b c r = if a > b then 0 else
65   evalFull c r + summ v (a+1) b c (extendEnv v (a+1) r)
66
67 evalFull :: Exp -> Env -> Integer
68 evalFull (Cst a) _ = a
69 evalFull (Add a b) r = evalFull a r + evalFull b r
70 evalFull (Sub a b) r = evalFull a r - evalFull b r
71 evalFull (Mul a b) r = evalFull a r * evalFull b r
72 evalFull (Div a b) r = evalFull a r `div` evalFull b r
73 -- check for negative exponent
74 evalFull (Pow a b) r
75   | evalFull b r < 0 = error "Negative exponent"
76   | otherwise = seq (evalFull a r) (evalFull a r ^ evalFull b r)
77 -- check if a is zero
78 evalFull (If a b c) r =
79   if evalFull a r /= 0 then evalFull b r else evalFull c r
80 evalFull (Var v) r = intTest(r v)
81 evalFull (Let a b c) r = evalFull c (extendEnv a (evalFull b r)
82   ↪ r)
83 evalFull (Sum v a b c) r =
84   summ v (evalFull a r) (evalFull b r) c (extendEnv v (evalFull a
85   ↪ r) r)
86
87 -- Exercise 3
88 intTestErr :: Maybe Integer -> VName -> Either ArithError Integer
89 intTestErr (Just i) _ = Right i
90 intTestErr _ v = Left (EBadVar v)
91
92 evalErr :: Exp -> Env -> Either ArithError Integer
93 evalErr (Cst a) _ = Right a
94 evalErr (Add a b) r = evalEither (evalErr a r) (+) (evalErr b r)
95 evalErr (Sub a b) r = evalEither (evalErr a r) (-) (evalErr b r)
96 evalErr (Mul a b) r = evalEither (evalErr a r) (*) (evalErr b r)
97 -- check for division by zero
98 evalErr (Div a b) r = if isRight (evalErr b r)
99   then if fromRight' (evalErr b r) /= 0
100     then evalEither (evalErr a r) div
101     ↪ (evalErr b r)
102   else Left EDivZero
103   else evalErr b r

```

```

101 -- check for negative exponent
102 evalErr (Pow a b) r = if isRight (evalErr b r)
103                      then if fromRight' (evalErr b r) >= 0
104                      then evalEither (evalErr a r) (^)
105                      else Left ENegPower
106                      else evalErr b r
107 -- check if a is zero
108 evalErr (If a b c) r = if isRight (evalErr a r)
109                      then if fromRight' (evalErr a r) /= 0
110                      then evalErr b r
111                      else evalErr c r
112                      else evalErr a r
113 evalErr (Var v) r = intTestErr (r v) v
114 evalErr (Let a b c) r = if isRight (evalErr b r)
115                      then evalErr c (extendEnv a
116                      → (fromRight' (evalErr b r)) r)
117                      else evalErr b r
118 evalErr (Sum v a b c) r = if isRight (evalErr a r)
119                      then if isRight (evalErr b r)
120                      then Right (summ v (fromRight'
121                      → (evalErr a r)) (fromRight' (evalErr b r)) c (extendEnv v
122                      → (fromRight' (evalErr a r)) r))
123                      else evalErr b r
124                      else evalErr a r
125 evalEither :: Either a b -> (b -> b -> b) -> Either a b -> Either
126           → a b
127 evalEither a b c = if isRight a
128                  then if isRight c
129                  then Right ( b (fromRight' a)
130                  → (fromRight' c))
131                  else c
132                  else a
133 -- use own implementation of fromRight from Data.Either but not
134   → returning a
135 -- default value, which is not needed for the assignment
136 fromRight' :: Either a b -> b
137 fromRight' (Right c) = c
138 fromRight' _ = error "No value"
139 -- optional parts (if not attempted, leave them unmodified)
140 →

```

```
138
139 showCompact :: Exp -> String
140 showCompact = undefined
141
142 evalEager :: Exp -> Env -> Either ArithError Integer
143 evalEager = undefined
144
145 evalLazy :: Exp -> Env -> Either ArithError Integer
146 evalLazy = undefined
```
