

# Haskell intro

## Assignement0

*Kai Arne S. Myklebust, Silvan Adrian*

Handed in: September 12, 2018



## Contents

<b>1</b>	<b>Design/Implementation</b>	<b>1</b>
<b>2</b>	<b>Code Assessment</b>	<b>2</b>
2.1	Test Results Online TA . . . . .	2
<b>A</b>	<b>Code Listing</b>	<b>3</b>

## 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').

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).

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.

### 2.1 Test Results Online TA

<b>showExpr</b>	<b>Result</b>
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

<b>evalSimple</b>	<b>Result</b>
-------------------	---------------

## 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
   ↪ there
47 evalSimple (Pow a b)
48   | evalSimple b < 0 = error "Negative exponent"
49   | otherwise = seq (evalSimple a) (evalSimple a ^ evalSimple b)
50 evalSimple _ = error "is not supported"
51
52 -- Exercise 2
53 extendEnv :: VName -> Integer -> Env -> Env
54 extendEnv v n r a = if v == a then Just n else r a
55
56 -- used to check if variable is unbound
57 intTest :: Maybe Integer -> Integer
58 intTest (Just i) = i
59 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)
    ↪ r)
82 evalFull (Sum v a b c) r =
83     summ v (evalFull a r) (evalFull b r) c (extendEnv v (evalFull a
    ↪ r) r)
84
85 -- Exercise 3
86 intTestErr :: Maybe Integer -> VName -> Either ArithError Integer
87 intTestErr (Just i) _ = Right i
88 intTestErr _ v = Left (EBadVar v)
89
90 evalErr :: Exp -> Env -> Either ArithError Integer
91 evalErr (Cst a) _ = Right a
92 evalErr (Add a b) r = evalEither (evalErr a r) (+) (evalErr b r)
93 evalErr (Sub a b) r = evalEither (evalErr a r) (-) (evalErr b r)
94 evalErr (Mul a b) r = evalEither (evalErr a r) (*) (evalErr b r)
95 -- check for division by zero
96 evalErr (Div a b) r = if isRight (evalErr b r)
97     then if fromRight' (evalErr b r) /= 0
98     then evalEither (evalErr a r) div
    ↪ (evalErr b r)
99     else Left EDivZero
100    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) (^)
    ↪ (evalErr b 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
    ↪   (fromRight' (evalErr b r)) r)
116         else evalErr b r
117
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'
    ↪   (evalErr a r)) (fromRight' (evalErr b r)) c (extendEnv v
    ↪   (fromRight' (evalErr a r)) r))
121         else evalErr b r
122         else evalErr a r
123
124 evalEither :: Either a b -> (b -> b -> b) -> Either a b -> Either
    ↪   a b
125 evalEither a b c = if isRight a
126         then if isRight c
127         then Right (b (fromRight' a)
    ↪   (fromRight' c))
128         else c
129         else a
130
131 -- use own implementation of fromRight from Data.Either but not
    ↪   returning a
132 -- default value, which is not needed for the assignment
133 fromRight' :: Either a b -> b
134 fromRight' (Right c) = c
135 fromRight' _ = error "No value"
136
137 -- optional parts (if not attempted, leave them unmodified)
    ↪
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

```

---