Haskell intro

Assignement 0

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 24, 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 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.

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' evaluates one expression after the other. 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.

We do have have 4 failing Test which should fail, those test the edge cases for division by zero and negative power exponent for 'evalSimple' and 'evalFull' which throw an exception.

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,
```

```haskell
  showCompact,
  evalEager,
  evalLazy
  )

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
```

```haskell

-- 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)
```

```
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 =
119   if isRight (evalErr a r)
120     then if isRight (evalErr b r)
121       then Right (summ v (fromRight' (evalErr a r)) (fromRight'
    ↪  (evalErr b r)) c (extendEnv v (fromRight'(evalErr a r)) r))
122       else evalErr b r
123     else evalErr a r
124
125 evalEither :: Either a b -> (b -> b -> b) -> Either a b -> Either
    ↪  a b
126 evalEither a b c = if isRight a
127                         then if isRight c
128                           then Right ( b (fromRight' a)
    ↪  (fromRight' c))
129                           else c
130                         else a
131
132 -- use own implementation of fromRight from Data.Either but not
    ↪  returning a
133 -- default value, which is not needed for the assignment
```

```haskell
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

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

B Tests Listing

```haskell
import Test.Tasty
import Test.Tasty.HUnit

import Definitions
import Arithmetic

main = defaultMain tests

tests :: TestTree
tests = testGroup "test" [
    showExpTest,
    evalSimpleTest,
    extendEnvTest,
    evalFullTest,
    evalErrTest
    ]

showExpTest :: TestTree
showExpTest = testGroup "showExp"
  [ testCase "Show Mul" $ showExp (Mul (Cst 2) (Cst 3)) @?= "(2 *
  ↪  3)",
    testCase "Show Minus" $ showExp(Cst (-3)) @?= "(-3)",
    testCase "Show Add" $ showExp(Add (Cst (-3)) (Cst (-4))) @?=
  ↪  "((-3) + (-4))",
```

```
23      testCase "Show Pow" $ showExp(Pow (Cst 2) (Cst 3)) @?=
    ↪  "(2^3)",
24      testCase "Show Div" $ showExp(Div (Cst 3) (Cst 4)) @?= "(3 /
    ↪  4)",
25      testCase "Show Sub" $ showExp(Sub (Cst 2) (Cst 4)) @?= "(2 -
    ↪  4)",
26      testCase "Show Mixture" $ showExp(Pow (Div (Cst 2) (Cst 3))
    ↪  (Sub (Cst 4) (Cst 3))) @?= "((2 / 3)^(4 - 3))"
27    ]
28
29  evalSimpleTest :: TestTree
30  evalSimpleTest = testGroup "evalSimple"
31    [ testCase "Cst" $ evalSimple (Cst 3) @?= 3,
32      testCase "Add" $ evalSimple(Add (Cst 3) (Cst 5)) @?= 8,
33      testCase "Sub" $ evalSimple(Sub (Cst 3) (Cst 5)) @?= -2,
34      testCase "Mul" $ evalSimple(Mul (Cst 3) (Cst 5)) @?= 15,
35      testCase "Div" $ evalSimple(Div (Cst 12) (Cst 3)) @?= 4,
36      testCase "Pow" $ evalSimple(Pow (Cst 2) (Cst 3)) @?= 8,
37      testCase "Mixture" $ evalSimple(Pow (Div (Cst 2) (Cst 3))
    ↪  (Sub (Cst 4) (Cst 3))) @?= 0,
38      testCase "Div by Zero fails" $ evalSimple(Div (Cst 2) (Cst
    ↪  0)) @?= 0,
39      testCase "Pow negative fails" $ evalSimple(Pow (Cst 2) (Cst
    ↪  (-1))) @?= 0
40    ]
41
42  extendEnvTest :: TestTree
43  extendEnvTest = testGroup "extendEnv"
44    [
45        testCase "extendEnv Simple" $ (extendEnv "x" 5 initEnv) "x"
    ↪  @?= Just 5,
46        testCase "extendEnv Extended" $ (extendEnv "x" 5 (extendEnv
    ↪  "x" 6 initEnv)) "x" @?= Just 5
47    ]
48
49  evalFullTest :: TestTree
50  evalFullTest = testGroup "fullEval"
51    [
52      testCase "Cst" $ evalFull(Cst 3) initEnv @?= 3,
53      testCase "Add" $ evalFull(Add (Cst 3) (Cst 5)) initEnv @?= 8,
54      testCase "Sub" $ evalFull(Sub (Cst 3) (Cst 5)) initEnv @?=
    ↪  -2,
55      testCase "Mul" $ evalFull(Mul (Cst 3) (Cst 5)) initEnv @?=
    ↪  15,
```

```
56      testCase "Div" $ evalFull(Div (Cst 12) (Cst 3)) initEnv @?=
    ↪  4,
57      testCase "Pow" $ evalFull(Pow (Cst 2) (Cst 3)) initEnv @?= 8,
58      testCase "Let" $ evalFull(Let "z" (Add (Cst 2) (Cst 3)) (Var
    ↪  "z")) initEnv @?= 5,
59      testCase "Sum" $ evalFull(Sum "x" (Cst 10) (Add (Cst 5) (Cst
    ↪  5)) (Mul (Cst 3) (Var "x"))) initEnv @?= 30,
60      testCase "Div by Zero fails" $ evalFull(Div (Cst 2) (Cst 0))
    ↪  initEnv @?= 0,
61      testCase "Pow negative fails" $ evalFull(Pow (Cst 2) (Cst
    ↪  (-1))) initEnv @?= 0
62    ]
63
64  evalErrTest :: TestTree
65  evalErrTest = testGroup "errEval"
66    [
67        testCase "Cst" $ evalErr (Cst 3) initEnv @?= Right 3,
68        testCase "Add" $ evalErr(Add (Cst 3) (Cst 5)) initEnv @?=
    ↪  Right 8,
69        testCase "Sub" $ evalErr(Sub (Cst 3) (Cst 5)) initEnv @?=
    ↪  Right (-2),
70        testCase "Mul" $ evalErr(Mul (Cst 3) (Cst 5)) initEnv @?=
    ↪  Right 15,
71        testCase "Div" $ evalErr(Div (Cst 12) (Cst 3)) initEnv @?=
    ↪  Right 4,
72        testCase "Pow" $ evalErr(Pow (Cst 2) (Cst 3)) initEnv @?=
    ↪  Right 8,
73        testCase "Let" $ evalErr(Let "z" (Add (Cst 2) (Cst 3)) (Var
    ↪  "z")) initEnv @?= Right 5,
74        testCase "Sum" $ evalErr(Sum "x" (Cst 10) (Add (Cst 5) (Cst
    ↪  5)) (Mul (Cst 3) (Var "x"))) initEnv @?= Right 30,
75        testCase "Div 0" $ evalErr(Div (Cst 12) (Cst 0)) initEnv
    ↪  @?= Left EDivZero,
76        testCase "Pow Negative" $ evalErr (Pow (Cst 2) (Cst (-1)))
    ↪  initEnv @?= Left ENegPower
77    ]
```