Advanced Programming
Assignment 1 : Arithmetic Expressions
Report

AP22 Assignment 1 Group 100
KRW521, CPJ395

September 15, 2022

1 Design and Implementation

1.1 Simple Arithmetic Expressions

For Printing Expression, we complete the functions showExp and evalSimple. Basically, they are realized by recursion.

While working on showExp, when showing a constant, we first determine whether the constant is negative or not. If it is negative, we add parentheses in output. If not, we just output the constant without parentheses. While showing the arithmetic expression, we recursively call the function showExp on the left expression and right expression, and then add the symbol of the corresponding expression between them(the code can be seen in the appendix or in the "code.zip"). If the input expression is not defined in the "Definition.hs"(i.e. the input does not match the type "Exp"), the program will output an error ("You have input a wrong expression") written by us. To ensure the priority of calculation in the output String, we add parentheses output outside the **Add, Sub, Mul, Div** and **Pow** expression.

While working on evalSimple, if the input is a constant, the program just output the constant. If the input is arithmetic expression, then we recursively call the evalSimple function on the left expression and right expression and then perform the corresponding action between them. If the "Div" expression receive zero as a denominator, an error "The denominator cannot be 0" written by us will be evoked. If the **Pow** expression receive a negative number as the exponent, an error "The exponent cannot be negative" written by us will be evoked. These two kinds of errors are evoked in branches of guards. What's more, while working on evalSimple of **Pow** we choose to force the compiler to check if there are some errors in the base expression in case of the exponent expression is zero by using $(*0) + 1$: **(evalSimple expr)==0 = (evalSimple expl) *0+1**.

1.2 Extended Arithmetic Expressions

As for extending the environment, we realize the function, **extendEnv**. Basically, it is a function that receive a **String**, an **Integer** and a function, then out put a function if it receive the same **String** as input **String**, then output the **Integer** or it will call the former function and send the **String** to former function. Concretely, it can be realized in Haskell as: **extendEnv vn i formerEnv = (\name − > if name == vn then Just i else formerEnv name)**

While working on the **evalFull**, the way we realize the arithmetic expression is similar to that in **evalSimple**. So I concentrate more on the other expressions here. For **Var** expression, we use **case...of...** key word to check if the value name has been binded by the name, if binded, then output the value, or output the **error "Value not binded"**. To realize **If, Let, and Sum**, we also use recursion. For example, to realize **If**, we program in this way: **evalFull If test=exp0, yes=exp1, no=exp2 env = if (evalFull exp0 env) /= 0 then (evalFull exp1 env) else (evalFull exp2 env)**, and **Let, Sum** expressions are realized in the similar way. Particularly, in **Sum** expression, we write **evalFull exp0 env < evalFull exp1 env = evalFull exp2 (extendEnv str (evalFull exp0 env) env) + evalFull Sum var = str, from = (Add (Cst 1) exp0), to = exp1, body = exp2 env** as a branch of guards while the value of the expression in "to" is greater than that in "from", and that is also a kind of recursion. What's more, while working on evalFull of **Pow** we choose to force the compiler to check if there are some errors in the base expression in case of the exponent expression is zero by using $(*0)+1$: **(evalFull expr env)==0 = (evalFull expl env) *0+1**.

1.3 Returning Explicit Errors

For the simplest expression **Cst i** in **evalErr**, we just output **Right i**. For arithmetic expressions and **Let, Sum, If**, we rebuild functions **eAdd, eSub, eMul, eDiv, ePow, eIf, eVar, eLet, eSum** for them to judge if the subexpressions have error(s), if have, then pop up the error(s) to the upper expression, otherwise realize the corresponding function. For example, we define **eAdd :: Either ArithError Integer − > Either ArithError Integer − > Either ArithError Integer**, if the 2 parameters of **eAdd** are Right i1 and Right i2,then we make the output of eAdd become Right(i1+i2), otherwise, we make it output the Left error to pop the error to the upper expression. So we only need to call **eAdd** while realize **evalErr** function for Add just like this: **evalErr (Add expl expr) env = eAdd (evalErr expl env) (evalErr expr env)** Likewise, the **evalErr** function of remaining expressions can be completed. Particularly, we also realize a function for **Sum** expression: **eCompare :: Either ArithError Integer − > Either ArithError Integer − > Ordering** to compare the **evalErr** of the sub expression (i.e. **from** and **to**) in **Sum** expression.

2 Assessment of The Code

2.1 Completeness

All functions are complete except for the three optional extensions, and the completions of all functions are as follows:

Question No.	Function Name	Completion
1.1 Printing expressions	showExp	○
1.2 Evaluating expressions	evalSimple	○
2 Extended arithmetic expressions	extendEnv	○
2 Extended arithmetic expressions	evalFull	○
3 Returning explicit errors	evalErr	○
4.2 Printing with minimal parentheses	showCompact	×
4.3 Explicitly eager/lazy semantics	evalEager	×
4.3 Explicitly eager/lazy semantics	evalLazy	×

2.2 Correctness

All functions have been judged by **OnlineTA**, and the feedback by OnlineTA is all "**OK**", In addition, through our tests, all functions of the project perform well, and our evaluation is as follows:

Question No.	Function Name	Test Result
1.1 Printing expressions	showExp	OK
1.2 Evaluating expressions	evalSimple	OK
2 Extended arithmetic expressions	extendEnv	OK
2 Extended arithmetic expressions	evalFull	OK
3 Returning explicit errors	evalErr	OK
4.2 Printing with minimal parentheses	showCompact	FAIL
4.3 Explicitly eager/lazy semantics	evalEager	FAIL
4.3 Explicitly eager/lazy semantics	evalLazy	FAIL

2.3 Efficiency

All functions run as expected in our tests. The running time of functions such as **Add Sub Mul Div Pow** is basically the same as the functions in the Haskell basic function library, and no abnormality was observed under the stress test. We also designed brute force tests such as $(1^1)^1 + (2^2)^2 + (3^3)^3 + (4^4)^4 + (5^5)^5 + (6^6)^6$ for extremely large numerical calculations, and its running time was in line with expectations. And We also designed a multi-level nested **Sum** statements to test whether **evalErr** function can correctly feedback deep errors The evaluation of function efficiency is as follows:

Question No.	Function Name	Run Time
1.1 Printing expressions	showExp	Normal
1.2 Evaluating expressions	evalSimple	Normal
2 Extended arithmetic expressions	extendEnv	Normal
2 Extended arithmetic expressions	evalFull	Normal
3 Returning explicit errors	evalErr	Normal
4.2 Printing with minimal parentheses	showCompact	N/A
4.3 Explicitly eager/lazy semantics	evalEager	N/A
4.3 Explicitly eager/lazy semantics	evalLazy	N/A

2.4 Maintainability

All functions in the code are commented, and the code is concise and neat. Since the **evalErr** functions do not use Monad, the code is somewhat verbose. Besides, due to the refactoring of the basic operation function, there are some code duplication in the form of copy-pasted segments with minor changes while realizing the **evalErr** function. But overall it is more readable and easy to maintain. Our evaluations of code maintainability are as follows:

Question No.	Function Name	Maintain.
1.1 Printing expressions	showExp	Perfect
1.2 Evaluating expressions	evalSimple	Perfect
2 Extended arithmetic expressions	extendEnv	Perfect
2 Extended arithmetic expressions	evalFull	Good
3 Returning explicit errors	evalErr	Not Bad
4.2 Printing with minimal parentheses	showCompact	N/A
4.3 Explicitly eager/lazy semantics	evalEager	N/A
4.3 Explicitly eager/lazy semantics	evalLazy	N/A

A Appendix: Arithmetic.hs

```
-- This is a skeleton file for you to edit

module Arithmetic
        (
        showExp,
        evalSimple,
        extendEnv,
        evalFull,
        evalErr,
        showCompact,
        evalEager,
        evalLazy
        )

where

import Definitions

--showExp function is basically realized by recursion
showExp :: Exp -> String
showExp (Cst i)
        | i>=0 = show i
        | i<0 = "("++  show i ++ ")"
showExp (Add expl expr) = "("++ showExp expl ++ "+"
      ++ showExp expr ++ ")"
showExp (Sub expl expr) = "("++ showExp expl ++ "-"
      ++ showExp expr ++ ")"
showExp (Mul expl expr) = "("++ showExp expl ++ "*"
      ++ showExp expr ++ ")"
showExp (Div expl expr) = "("++ showExp expl ++ "`div`"
   ++ showExp expr ++ ")"
showExp (Pow expl expr) = "("++ showExp expl ++ "^"
      ++ showExp expr ++ ")"
showExp _ = error "You have input a wrong expression"

--showSimple function is basically realized by
   recursion
evalSimple :: Exp -> Integer
evalSimple (Cst i) =  (i)
evalSimple (Add expl expr) = evalSimple expl +
   evalSimple expr
evalSimple (Sub expl expr) = evalSimple expl −
   evalSimple expr
evalSimple (Mul expl expr) = evalSimple expl ∗
   evalSimple expr
evalSimple (Div expl expr)
         | (evalSimple expr)==0 = error "The denominator
             cannot be 0"
         | otherwise = (evalSimple expl `div` evalSimple
             expr)
evalSimple (Pow expl expr)
         | (evalSimple expr)<0 = error "The exponent
            cannot be negative"
         | (evalSimple expr)==0 = (evalSimple expl) ∗0+1
         | otherwise = (evalSimple expl) ^ (evalSimple
            expr)


{−The output  of extendEnv is the function that check
   if the input name is Vname.
If is, then out put the Integer;
if not, then find if the former environment has the
   input name binded with an Integer −}
extendEnv :: VName −> Integer −> Env −> Env
extendEnv vn i formerEnv = (\name −> if name == vn then
    Just i else formerEnv name)


−−showFull function is basically realized by recursion
evalFull :: Exp −> Env −> Integer
evalFull (Cst i) env  =  (i)
evalFull (Add expl expr) env  = (evalFull expl env)
       + (evalFull expr env)
evalFull (Sub expl expr) env  = (evalFull expl env)
       − (evalFull expr env)
evalFull (Mul expl expr) env  = (evalFull expl env)
       ∗ (evalFull expr env)
evalFull (Div expl expr) env  = (evalFull expl env)
   `div` (evalFull expr env)
```

```
evalFull (Pow expl expr) env
        | (evalFull expr env)  < 0 = error "The
           exponent cannot be negative"
        | (evalFull expr env) == 0 = (evalFull expl env
           ) *0+1
        | otherwise = (evalFull expl env) ^ (evalFull
           expr env)
evalFull If {test=exp0, yes=exp1, no=exp2} env = if (
   evalFull exp0 env) /= 0 then (evalFull exp1 env)
   else (evalFull exp2 env)
evalFull (Var str) env =
        case env str of
                Just n -> n
                Nothing -> error "Value not binded"
evalFull Let {var = str, def= exp0, body=exp1} env =
   evalFull exp1 (extendEnv str (evalFull exp0 env) env
   )
evalFull Sum {var = str, from = exp0, to = exp1, body =
    exp2} env
        | evalFull exp0 env > evalFull exp1 env = 0
        | evalFull exp0 env == evalFull exp1 env =
           evalFull exp2 (extendEnv str (evalFull exp0
           env) env)
        | evalFull exp0 env < evalFull exp1 env =
           evalFull exp2 (extendEnv str (evalFull exp0
           env) env) + evalFull Sum {var = str, from =
           (Add (Cst 1) exp0), to = exp1, body = exp2}
           env

{-While realizing the evalErr function of these
   expressions,just call the functions
eAdd, eSub, eMul, eDiv, ePow, eIf, eVar, eLet and eSum
   -}
evalErr :: Exp -> Env -> Either ArithError Integer
evalErr (Cst i) env  = Right i
evalErr (Add expl expr) env = eAdd (evalErr expl env)
   (evalErr expr env)
evalErr (Sub expl expr) env = eSub (evalErr expl env)
   (evalErr expr env)
evalErr (Mul expl expr) env = eMul (evalErr expl env)
   (evalErr expr env)
```

```
evalErr (Div expl expr) env
        | (evalErr expr env) == Right 0 = Left EDivZero
        | otherwise = eDiv (evalErr expl env)  (evalErr
            expr env)
evalErr (Pow expl expr) env
        | eCompare (evalErr expr env) (Right 0) == LT =
            Left ENegPower
        | otherwise = ePow (evalErr expl env) (evalErr
           expr env)
evalErr If {test=exp0, yes=exp1, no=exp2} env = eIf (
   evalErr exp0 env)  (evalErr exp1 env) (evalErr exp2
   env)
evalErr (Var str) env = eVar str env
evalErr Let {var = str, def= exp0, body=exp1} env =
   eLet (evalErr exp0 env) (evalErr exp1 (extendEnv str
    ((\(Right i)->i) (evalErr exp0 env)) env))
evalErr Sum {var = str, from = exp0, to = exp1, body =
   exp2} env
        | eCompare (evalErr exp0 env)   (evalErr exp1
           env) == GT = Right 0
        | eCompare (evalErr exp0 env)   (evalErr exp1
           env) == EQ = eSum (evalErr exp0 env) (
           evalErr exp1 env) (evalErr exp2 (extendEnv
           str ((\(Right i)->i) (evalErr exp0 env))
           env))
        | eCompare (evalErr exp0 env)   (evalErr exp1
           env) == LT = eAdd (eSum (evalErr exp0 env) (
           evalErr exp1 env) (evalErr exp2 (extendEnv
           str ((\(Right i)->i) (evalErr exp0 env))
           env))) (evalErr Sum {var = str, from = (Add
            (Cst 1) exp0), to = exp1, body = exp2} env)


{-Rebuild functions eAdd, eSub, eMul, eDiv, ePow, eIf,
   eVar, eLet and eSum
for the evalErr functions of expressions to judge if
   the sub-expressions have error(s),
if have, then pop up the error(s) to the upper
   expression,
otherwise realize the corresponding function. -}
```

```
eAdd :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer
eAdd (Right i1) (Right i2) = Right (i1+i2)
eAdd (Left a1) _ = Left a1
eAdd _ (Left a2) = Left a2

eSub :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer
eSub (Right i1) (Right i2) = Right (i1-i2)
eSub (Left a1) _ = Left a1
eSub _ (Left a2) = Left a2

eMul :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer
eMul (Right i1) (Right i2) = Right (i1*i2)
eMul (Left a1) _ = Left a1
eMul _ (Left a2) = Left a2

eDiv :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer
eDiv (Right i1) (Right i2) = Right (i1 `div` i2)
eDiv (Left a1) _ = Left a1
eDiv _ (Left a2) = Left a2

ePow :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer
ePow (Right i1) (Right i2) = Right (i1^i2)
ePow (Left a1) _ = Left a1
ePow _ (Left a2) = Left a2

eIf :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer -> Either
   ArithError Integer
eIf (Right i1) (Right i2) (Left a3)  = if i1 /= 0 then
   (Right i2) else (Right i2)
eIf (Right i1) (Left a2)  (Right i3) = if i1 == 0 then
   (Right i3) else (Right i3)
eIf (Right i1) (Right i2) (Right i3) = if i1 == 0 then
   (Right i3) else (Right i2)
eIf (Left a1) _ _ = Left a1
eIf _ (Left a2) _ = Left a2
```

```haskell
eIf _ _ (Left a3) = Left a3

eVar :: VName -> Env -> Either ArithError Integer
eVar n env
        | env n == Nothing = Left (EBadVar n)
        | otherwise        = Right ((\(Just a)->a) (env
            n))

eLet ::  Either ArithError Integer -> Either ArithError
    Integer -> Either ArithError Integer
eLet  (Right i1) (Right i2) =  Right i2
eLet  (Left a1) _ =  Left a1
eLet  _ (Left a2) =  Left a2

eSum :: Either ArithError Integer -> Either ArithError
   Integer -> Either ArithError Integer -> Either
   ArithError Integer
eSum (Right i1) (Right i2) (Right i3) = Right i3
eSum (Left a1) _ _ =  Left a1
eSum _ (Left a2) _ =  Left a2
eSum _ _ (Left a3) =  Left a3


{- Realize a function for evalErr of Sum expression,
to compare the evalErr of the sub expression (i.e. from
    and to) in Sum expression -}
eCompare :: Either ArithError Integer -> Either
   ArithError Integer -> Ordering
eCompare (Right i1) (Right i2)
        | i1>i2 = GT
        | i1<i2 = LT
        | i1==i2 =EQ
eCompare _ _ = EQ



-- 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 Appendix: Test.hs

```
-- Very rudimentary test of Arithmetic. Feel free to replace completely

import Definitions
import Arithmetic

import Data.List (intercalate)
import System.Exit (exitSuccess, exitFailure)  -- for when running
    stand-alone

tests :: [(String, Bool)]
tests = [test1, test2, test3,test4,test5,test6,test7,test8,test9,test10,
    test11,test12,test13,test14,test15,test16,test17,test18,test19,
    test20,test21,test22,test23,test24,test25,test26,test27,test28] where
  -- showExp tests
  test1 = ("test1", showExp (Add (Cst 1) (Sub (Cst 2) (Mul (Cst 3) (Div
      (Cst 4) (Pow (Cst 5) (Cst 6)))))) == "(1+(2-(3*(4`div`(5^6)))))")
  test2 = ("test2", showExp (Pow (Cst 1) (Pow (Cst 1) (Pow (Cst 1)
          (Pow (Cst 1) (Pow (Cst 1) (Pow (Cst 1) (Pow (Cst 1) (Pow (Cst
      1) (Cst 1)))))))) == "(1^(1^(1^(1^(1^(1^(1^(1^1))))))))")
  test3 = ("test3", showExp (Add (Cst (-3)) (Cst (-4))) == "((-3)
      +(-4))")
  test4 = ("test4", showExp (Cst (-1)) == "(-1)")
  test5 = ("test5", showExp (Sub (Cst (-2)) (Add (Cst (-3))
      (Cst (-4)))) == "((-2)-((-3)+(-4)))")
  -- evalSimple tests
  test6 = ("test6", evalSimple (Mul (Cst 1234567890) (Cst 1234567890))
      == 1234567890*1234567890)
  test7 = ("test7", evalSimple (Cst (-1)) == -1)
  test8 = ("test8", evalSimple (Div (Cst 0) (Cst 1)) == 0)
  test9 = ("test9", evalSimple (Add (Cst 1) (Sub (Cst 2) (Mul (Cst 3)
        (Div (Cst 4) (Pow (Cst 5) (Cst 6)))))) ==
    1+(2−(3∗(4`div`(5ˆ6)))))
test10 = (”test10”, evalSimple (Pow (Cst 2) (Pow (Cst 2) (Pow (Cst 2)
    (Cst 2)))) == 2ˆ(2ˆ(2ˆ2)))
−− extendEnv tests
test11 = (”test11”, (extendEnv ”x” 5 initEnv) ”x” == Just 5)
test12 = (”test12”, (extendEnv ”x” 5 (extendEnv ”y” 6 initEnv)) ”z”
    == Nothing)
test13 = (”test13”, (extendEnv ”x” 5 (extendEnv ”y” 6 initEnv)) ”x”
    == Just 5)
test14 = (”test14”, (extendEnv ”x” 5 (extendEnv ”y” 6 initEnv)) ”y”
    == Just 6)
test15 = (”test15”, (extendEnv ”x” 5 (extendEnv ”x” 6 initEnv)) ”x”
    == Just 5)
−− evalFull tests
test16 = (”test16”, evalFull (Cst (−1234567890987654321)) initEnv ==
     −1234567890987654321)
test17 = (”test17”, evalFull (Add (Cst 1) (Sub (Cst 2) (Mul (Cst 3)
        (Div (Cst 4) (Pow (Cst 5) (Cst 6)))))) initEnv ==
    1+(2−(3∗(4`div`(5ˆ6)))))
test18 = (”test18”, evalFull (Let ”x” (Add (Cst 3)
                                (Cst (−10000000000000003))) (Var ”x”))
    initEnv == −10000000000000000)
test19 = (”test19”, evalFull (Sum ”x” (Let ”x” (Sub (Cst 2) (Cst 1))
                        (Var ”x”)) (If (Sub (Cst 1) (Cst 1)) (Cst 1) (Cst
    100)) (Var ”x”)) initEnv == ((1+100)∗100)`div`2)
test20 = (”test20”, evalFull (Var ”x”) (extendEnv ”x” (−1) initEnv)
    == −1)
−− evalErr tests
test21 = (”test21”, evalErr (Add (Cst 1) (Sub (Cst 2) (Mul (Cst 3)
        (Div (Cst 4) (Pow (Cst 5) (Cst 6)))))) initEnv == Right
    (1+(2−(3∗(4`div`(5ˆ6))))))
test22 = (”test22”, evalErr (Let ”x” (Add (Cst 3) (Var ”x”)) (Var ”x”))
     initEnv == Left (EBadVar ”x”))
test23 = (”test23”, evalErr (Cst (−1234567890987654321)) initEnv ==
    Right (−1234567890987654321))
test24 = (”test24”, evalErr (Sum ”x” (Cst 10000) (Cst 0) (Cst 1))
    initEnv == Right 0)
test25 = (”test25”, evalErr (Sum ”x” (Cst 1) (Pow (Cst 1) (Sub (Cst 1)
     (Cst 2))) (Sum ”x” (Cst 1) (Cst 10) (Var ”x”))) initEnv == Left
    ENegPower)
```

```
  -- pressure test
  test26 = ("test26", evalFull (Sum "x" (Let "x" (Cst 1) (Var "x"))
      (Sum "x" (Cst 1) (Cst 3) (Var "x")) (Pow (Var "x") (Pow (Var "x")
       (Var "x")))) initEnv == (1^(1^1)+2^(2^2)+3^(3^3)+4^(4^4)
      +5^(5^5)+6^(6^6)))
  test27 = ("test27", evalErr (Sum "x" (Cst 1) (Sum "x" (Cst 1) (Sum "
      x" (Cst 1)          (Sum "x" (Cst 1) (Sum "x" (Cst 1) (Sum "x" (Cst
      1) (Div (Sum "x" (Cst 1) (Pow (Cst 1) (Cst (-1))) (Var "x")) (Cst
      0)) (Var "x")) (Var "x")) (Var "x")) (Var "x")) (Var "x"))
      (Var "x")) initEnv == Left EDivZero)
  test28 = ("test28", evalErr (Sum "x" (Cst 1) (Sum "x" (Cst 1) (Sum
          "x" (Cst 1) (Sum "x" (Cst 1) (Sum "x" (Cst 1) (Sum "x" (Cst
      1) (Div (Sum "x" (Let "x" (Var "y") (Var "x")) (Pow (Cst 1) (Cst
      1)) (Var "x")) (Cst 1)) (Var "x")) (Var "x")) (Var "x")) (Var "x"))
        (Var "x")) (Var "x")) initEnv == Left (EBadVar "y"))

main :: IO ()
main =
  let failed = [name | (name, ok) <- tests, not ok]
  in case failed of
       [] -> do putStrLn "All tests passed!"
                exitSuccess
       _ -> do putStrLn $ "Failed tests: " ++ intercalate ", " failed
               exitFailure
```