Advanced Programming

Assignment 0: Arithmetic Expressions

pcr902

September 9, 2020

Contents

1 Design and Implementation Choices

1.1 showExp

It reports error for unsupported expressions

1.2 evalSimple

It uses the expression using lazy evaluation. But in the case of Pow e1 e2 it uses eager evaluation by evaluating e1 first in case of any errors. In case of any errors like unsupported operations, negative power, and divide by zero, it aborts with meaningful messages.

1.3 extendEnv

It returns a new environment with an integer bound to a variable. It creates the binding in the environment as the functional values.

1.4 evalFull

It uses the lazy evaluation, the default way in which Haskell functions. In *Let var aux body* operation, if an error occurs in auxiliary expression aux, it will not be signaled if the bound variable var is not used in the body. Otherwise, It reports error well.

1.5 evalErr

It uses the eager and left to right evaluation, i.e., test expressions from left to right before for errors and finally evaluate. In *Let var aux body* operation, if an error occurs in auxiliary expression aux, it will be signaled irrespective of whether the bound variable var is used in the body or not. It reports all the possible errors and avoids any run-time errors.

1.6 showCompact

Not Implemented

1.7 evalEager

Not Implemented

1.8 evalLazy

Not Implemented

2 Code Assessment

The code is reasonably readable, clean, and appropriately commented. It uses a consistent indentation style and avoids lines of over 80 characters as much as possible. It crosses 80 characters limit while defining evalErr sum. It reports well-defined error messages for unsupported expressions in showExp, evalSimple, and evalFull.

We avoided repetition of code by abstracting out common snippets into auxiliary functions evalErrHelp, evalErrHelp1 in evalErr. Though, We think there is still scope of improvement. We could also have done the naming of the variables used throughout the code in a better way.

It is relatively robust. The code has passed all the tests on the onlineTA. And We have also tested it for corner cases of all the functions. However, we were not able to test the error handling in showExp, evalsimple, and evalFull functions through our automated unit testing. But We tested it manually, and it works perfectly fine. We don't get any warnings on compiling the code with ghc(i) -W. On OnlineTa, we get some suggestions for the redundant bracket used in the code. But we have used that for our readability of the code.

We have used a test-driven development process for this assignment. We created the test cases beforehand and then developed the code and tested it thoroughly.

3 Appendix A - Warmup.hs

```
1
2  ----------------------------------------
3  -- move
4  ----------------------------------------
5  move :: Direction -> Pos -> Pos
6  move North (x,y) = (x, y+1)
7  move West  (x,y) = (x-1, y)
8  move East  (x,y) = (x+1, y)
9  move South (x,y) = (x, y-1)
10
11
12 ----------------------------------------
13 -- moves
14 ----------------------------------------
15 moves :: [Direction] -> Pos -> Pos
16 moves [] pos = pos
17 moves (d:ds) pos = moves ds . move d $ pos
18
19
20 data Nat = Zero | Succ Nat
21   deriving (Eq, Show, Read, Ord)
22
23 ----------------------------------------
24 -- add
25 ----------------------------------------
26 add :: Nat -> Nat -> Nat
27 add Zero x = x
28 add x Zero = x
29 add x (Succ y) = add (Succ x) y
30
31 ----------------------------------------
32 -- mult
33 ----------------------------------------
34 mult :: Nat -> Nat -> Nat
35 mult Zero _ = Zero
36 mult _ Zero = Zero
37 mult (Succ Zero) x = x
38 mult x (Succ Zero) = x
39 mult x (Succ y) = add x . mult x $ y
40
41 ----------------------------------------
42 -- nat2int
43 ----------------------------------------
44 nat2int :: Nat -> Int
45 nat2int Zero = 0
46 nat2int (Succ x) = 1 + nat2int x
47
48 ----------------------------------------
49 -- int2nat
50 ----------------------------------------
51 int2nat :: Int -> Nat
52 int2nat x
```

```
53    | x==0 = Zero
54    | x>0 = Succ( int2nat (x-1))
55    | otherwise = Zero
56
57
58  data Tree = Leaf | Node Int Tree Tree
59    deriving (Eq, Show, Read, Ord)
60
61  ---------------------------------------
62  -- insert
63  ---------------------------------------
64  insert :: Int -> Tree -> Tree
65  insert n Leaf = Node n Leaf Leaf
66  insert n t@(Node k left right)
67    | n==k = t
68    | n<k = Node k (insert n left) right
69    | otherwise = Node k left (insert n right)
```

4 Appendix B - Arithmetic.hs

```haskell

----------------------------------------
-- error constants
----------------------------------------
invalidExpressionError = "EOther : Not a valid expression"
divZeroError = "EDivZero"
negPowerError = "ENegPower"
badVarError = "EBadVar"

----------------------------------------
-- showExp
----------------------------------------
showExp :: Exp -> String
showExp (Cst x)
  | x < 0 = "("++ show x ++ ")"
  | otherwise = show x
showExp (Add x y) = "("++ showExp x ++ "+" ++ showExp y ++ ")"
showExp (Sub x y) = "("++ showExp x ++ "-" ++ showExp y ++ ")"
showExp (Mul x y) = "("++ showExp x ++ "*" ++ showExp y ++ ")"
showExp (Div x y) = "("++ showExp x ++ "/" ++ showExp y ++ ")"
showExp (Pow x y) = "("++ showExp x ++ "^" ++ showExp y ++ ")"
showExp _ = error invalidExpressionError

----------------------------------------
-- evalSimple
----------------------------------------
evalSimple :: Exp -> Integer
evalSimple (Cst x) = x
evalSimple (Add x y) = evalSimple x + evalSimple y
evalSimple (Sub x y) = evalSimple x - evalSimple y
evalSimple (Mul x y) = evalSimple x * evalSimple y
evalSimple (Div x y)
  | (evalSimple y) == 0 = error divZeroError
  | otherwise = div (evalSimple x) (evalSimple y)
evalSimple (Pow x y)
  | (evalSimple y) < 0 = error negPowerError
  ----------------------------------------
  -- evaluate x for errors
  ----------------------------------------
  | (evalSimple x) == 0 = 0 ^ (evalSimple y)
  | otherwise = (evalSimple x) ^ (evalSimple y)
evalSimple _ = error invalidExpressionError

----------------------------------------
-- extendEnv
----------------------------------------
extendEnv :: VName -> Integer -> Env -> Env
extendEnv name num env e
 |name == e = Just num
 |otherwise = env e

----------------------------------------
```

```haskell
-- evalFull
-----------------------------------------
evalFull :: Exp -> Env -> Integer
evalFull (Cst x) _ = x
evalFull (Var x) env = case (env x) of
  Nothing -> error badVarError
  Just v -> v
evalFull (Add x y) env = evalFull x env + evalFull y env
evalFull (Sub x y) env = evalFull x env - evalFull y env
evalFull (Mul x y) env = evalFull x env * evalFull y env
evalFull (Div x y) env
  | (evalFull y env) == 0 = error divZeroError
  | otherwise = div (evalFull x env) (evalFull y env)
evalFull (Pow x y) env
  | (evalFull y env) < 0 = error negPowerError
  | (evalFull x env) == 0 = 0 ^ (evalFull y env)
  | otherwise = (evalFull x env) ^ (evalFull y env)
evalFull (If test yes no) env
  | r == 0 = evalFull no env
  | otherwise = evalFull yes env
  where r = (evalFull test env)
evalFull (Let var aux body) env = evalFull body (extendEnv var x env)
  where x = evalFull aux env
evalFull (Sum var from to body) env
  | n1 > n2 = 0
  | otherwise = (evalFull body env1) +
        evalFull (Sum var (Add (Cst n1) (Cst 1) ) (Cst n2) body) env
  where n1 = evalFull from env
        n2 = evalFull to env
        env1 = extendEnv var n1 env

-----------------------------------------------------------------
-- evalErrHelp
-- helper function for evalErr which evaluates an expression
-- and then apply its value to the partial function passed
-- as parameter if it doesn't retutn ArithError
-----------------------------------------------------------------
evalErrHelp :: Exp -> Env -> (Integer -> Either ArithError Integer) ->
  Either ArithError Integer
evalErrHelp ex env f = case evalErr ex env of
  Left n -> Left n
  Right n -> f n

-----------------------------------------------------------------
-- evalErrHelp1
-- helper function for evalErr which evaluates expressions
-- ex and ex1 and then apply their values to the partial
-- function passed as parameter if they donot return
-- ArithError
-----------------------------------------------------------------
evalErrHelp1 :: Exp -> Exp -> Env -> Env ->
  ((Integer, Integer) -> Either ArithError Integer) ->
  Either ArithError Integer
evalErrHelp1 ex ex1 env env1 f = case evalErr ex env of
  Left n1 -> Left n1
```

```
  Right n1 -> case evalErr ex1 env1 of
    Left n2 -> Left n2
    Right n2 -> f (n1,n2)

----------------------------------------
-- evalErr
----------------------------------------
evalErr :: Exp -> Env -> Either ArithError Integer
evalErr (Cst x) _ = Right x
evalErr (Var x) env = case (env x) of
  Nothing -> Left (EBadVar x)
  Just v -> Right v
evalErr (Add x y) env = evalErrHelp1 x y env env f1
  where f1 = \(n1, n2) ->  Right (n1 + n2)
evalErr (Sub x y) env = evalErrHelp1 x y env env f1
  where f1 = \(n1, n2) ->  Right (n1 - n2)
evalErr (Mul x y) env = evalErrHelp1 x y env env f1
  where f1 = \(n1, n2) ->  Right (n1 * n2)
evalErr (Div x y) env = evalErrHelp1 x y env env f1
  where f1 = \(n1, n2) ->  if n2==0 then Left EDivZero else Right (div
    n1 n2)
evalErr (Pow x y) env = evalErrHelp1 x y env env f1
  where f1 = \(n1, n2) ->  if n2<0 then Left ENegPower else Right (n1 ^
    n2)
evalErr (If test yes no) env = evalErrHelp test env f1
  where f1 = \n1 -> if n1==0 then evalErr no env else evalErr yes env
evalErr (Let var aux body) env = evalErrHelp aux env f1
  where f1 = \n1 -> evalErr body (extendEnv var n1 env)
----------------------------------------------
-- evaluate from and to values and
-- then while from <= to bind updated
-- from value in env and compute the
-- sum recursively
----------------------------------------------
evalErr (Sum var from to body) env = evalErrHelp1 from to env env f1
  where f1 = \(n1, n2) -> if n1>n2 then Right 0 else
        evalErrHelp1 body (Sum var (Add (Cst n1) (Cst 1)) (Cst n2) body)
            (extendEnv var n1 env) env f2
        f2 = \(n3,n4) -> Right (n3 + n4)

-- optional parts (if not attempted, leave them unmodified)
----------------------------------------
-- showCompact
----------------------------------------
showCompact :: Exp -> String
showCompact = undefined

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

----------------------------------------
-- evalLazy
```

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

5 Appendix C - Test.hs

```
1
2  env :: Env
3  env = (extendEnv "a" 3 (extendEnv "b" 4 initEnv))
4
5  tests :: [(String, Bool)]
6  tests = [testShowExp1, testShowExp2, testShowExp3,
7           testevalSimple1, testevalSimple2, testevalSimple3,
8           testevalSimple4, testevalSimple5, testevalSimple6,
9           testextendEnv1, testextendEnv2, testextendEnv3, testextendEnv4
       ,
10          testevalFull1, testevalFull2, testevalFull3, testevalFull4,
11          testevalFull5, testevalFull6, testevalFull7, testevalFull8,
12          testevalFull9, testevalFull10, testevalFull11, testevalFull12,
13          testevalFull13, testevalFull14,
14          testevalErr1, testevalErr2, testevalErr3, testevalErr4,
15          testevalErr5, testevalErr6, testevalErr7, testevalErr8,
16          testevalErr9, testevalErr10, testevalErr11, testevalErr12,
17          testevalErr13, testevalErr14, testevalErr15, testevalErr16,
18          testevalErr17
19        ] where
20
21  ----------------------------------------
22  -- tests for showExp
23  ----------------------------------------
24    testShowExp1 = ("testShowExp1", showExp (Cst (-4)) == "(-4)")
25    testShowExp2 = ("testShowExp2", showExp (Cst (5)) == "5")
26    testShowExp3 = ("testShowExp3", showExp (Div (Cst (-2)) (Sub (Cst 3) (
       Cst 4))) == "((-2)/(3-4))")
27
28  ----------------------------------------
29  -- tests for evalSimple
30  ----------------------------------------
31    testevalSimple1 = ("testevalSimple1", evalSimple (Cst (-7)) == (-7))
32    testevalSimple2 = ("testevalSimple2", evalSimple (Add (Cst 12) (Cst
       24)) == 36)
33    testevalSimple3 = ("testevalSimple3",
34      evalSimple (Sub (Add (Cst 2) (Cst 3)) (Cst 5)) == 0)
35    testevalSimple4 = ("testevalSimple4",
36      evalSimple (Pow (Cst 2) (Add (Cst (-1))(Cst 1))) == 1)
37    testevalSimple5 = ("testevalSimple5", evalSimple (Pow (Cst 0) (Cst 0))
       == 1)
38    testevalSimple6 = ("testevalSimple6",
39      evalSimple (Div (Cst 1) (Mul (Cst 2)(Cst 1))) == 0)
40
41  ----------------------------------------
42  -- tests for extendEnv
43  ----------------------------------------
44    testextendEnv1 = ("testextendEnv1", (extendEnv "a" 9 initEnv) "a" ==
       Just 9)
45    testextendEnv2 = ("testextendEnv2", (extendEnv "a" 4 initEnv) "z" ==
       Nothing)
46    testextendEnv3 = ("testextendEnv3", env "a" == Just 3)
```

```
47    testextendEnv4 = ("testextendEnv4", env "z" == Nothing)
48
49  ----------------------------------------
50  -- tests for evalFull
51  ----------------------------------------
52    testevalFull1 = ("testevalFull1",
53      evalFull (Let "a" (Cst 42) (Var "a")) initEnv == 42)
54    testevalFull2 = ("testevalFull2", evalFull (Cst (-7)) env == (-7))
55    testevalFull3 = ("testevalFull3", evalFull (Add (Cst 12) (Cst 24)) env
        == 36)
56    testevalFull4 = ("testevalFull4",
57      evalFull (Sub (Add (Cst 2) (Cst 3)) (Cst 5)) env == 0)
58    testevalFull5 = ("testevalFull5",
59      evalFull (Pow (Cst 2) (Add (Cst (-1))(Cst 1))) env == 1)
60    testevalFull6 = ("testevalFull6", evalFull (Pow (Cst 0) (Cst 0)) env
       == 1)
61    testevalFull7 = ("testevalFull7",
62      evalFull (Div (Cst 12) (Mul (Cst (-2))(Cst 1))) env == (-6))
63    testevalFull8 = ("testevalFull8",
64      evalFull (Let "x" (Pow (Cst 3) (Cst (-1))) (Var "b")) env  == 4)
65    testevalFull9 = ("testevalFull9",
66      evalFull (Let "x" (Div (Cst 3) (Cst 3)) (Add (Var "x")(Var "b")))
       env  == 5)
67    testevalFull10 = ("testevalFull10", evalFull (Var "a") env == 3)
68    testevalFull11 = ("testevalFull11",
69      evalFull (If (Cst 0) (Div (Cst 1) (Cst 0)) (Cst 2)) env == 2)
70    testevalFull12 = ("testevalFull12",
71      evalFull (If (Cst (1)) (Cst 1) (Div (Cst 1) (Cst 0))) env == 1)
72    testevalFull13 = ("testevalFull13",
73      evalFull (Sum "a" (Cst 1) (Cst 3) (Cst 5)) env == 15)
74    testevalFull14 = ("testevalFull14",
75      evalFull (Sum "x" (Cst 0) (Add (Cst 2) (Cst 2)) (Mul (Var "x") (Var
       "x"))) env == 30)
76
77  ----------------------------------------
78  -- tests for evalErr
79  ----------------------------------------
80    testevalErr1 = ("testevalErr1",
81      evalErr (Let "a" (Cst 42) (Var "a")) initEnv == Right 42)
82    testevalErr2 = ("testevalErr2", evalErr (Cst (-7)) env == Right (-7))
83    testevalErr3 = ("testevalErr3", evalErr (Add (Cst 12) (Cst 24)) env ==
        Right 36)
84    testevalErr4 = ("testevalErr4",
85      evalErr (Sub (Add (Cst 2) (Cst 3)) (Cst 5)) env == Right 0)
86    testevalErr5 = ("testevalErr5",
87      evalErr (Pow (Cst 2) (Add (Cst (-1))(Cst 1))) env == Right 1)
88    testevalErr6 = ("testevalErr6", evalErr (Pow (Cst 0) (Cst 0)) env ==
       Right 1)
89    testevalErr7 = ("testevalErr7",
90      evalErr (Div (Cst 12) (Mul (Cst (-2))(Cst 1))) env == Right (-6))
91    testevalErr8 = ("testevalErr8",
92      evalErr (Let "x" (Pow (Cst 3) (Cst (-1))) (Var "b")) env  == Left
       ENegPower)
93    testevalErr9 = ("testevalErr9",
94      evalErr (Let "x" (Div (Cst 3) (Cst 3)) (Add (Var "x")(Var "b"))) env
```

```
    == Right 5)
testevalErr10 = ("testevalErr10", evalErr (Var "a") env == Right 3)
testevalErr11 = ("testevalErr11",
  evalErr (If (Cst 0) (Div (Cst 1) (Cst 0)) (Cst 2)) env == Right 2)
testevalErr12 = ("testevalErr12",
  evalErr (If (Cst (1)) (Cst 1) (Div (Cst 1) (Cst 0))) env == Right 1)
testevalErr13 = ("testevalErr13",
  evalErr (Sum "a" (Cst 1) (Cst 3) (Cst 5)) env == Right 15)
testevalErr14 = ("testevalErr14",
  evalErr (Sum "x" (Cst 0) (Add (Cst 2) (Cst 2)) (Mul (Var "x") (Var "
 x"))) env == Right 30)
testevalErr15 = ("testevalErr15",
  evalErr (Div (Cst 12) (Cst 0)) env  == Left EDivZero)
testevalErr16 = ("testevalErr16", evalErr (Var "c") env == Left (
 EBadVar "c"))
testevalErr17 = ("testevalErr17",
  evalErr (Sum "z" (Cst (0)) (Div (Cst 3) (Cst 0) ) (Var "z") ) env ==
  Left EDivZero)
```