# CS790/657–002 Functional Programming – Spring 2020 final (100 points)

May 5, 2020

## 1 Interpreter (100 points)

Implement an interpreter for a language with the following Abstract Syntax Tree definition.

```haskell
-- function/variable declaration
data Decl = Fun String String Exp -- fun f x = e;
          | Val String Exp        -- val x = e;

newtype DeclList = Decls [Decl]

-- expressions
data Exp = Lt Exp Exp      -- e1 < e2
         | Gt Exp Exp      -- e1 > e2
         | Eq Exp Exp      -- e1 = e2
         | Plus Exp Exp    -- e1 + e2
         | Minus Exp Exp   -- e1 - e2
         | Times Exp Exp   -- e1 * e2
         | Div Exp Exp     -- e1 div e2
         | Var String      -- x
         | If Exp Exp Exp -- if e0 then e1 else e2
         | Fn String Exp   -- fn x => e
         | Let DeclList Exp -- let val x = e0; fun f = e1; in e2 end
         | App Exp Exp     -- e1 e2
         | Const Integer   -- n
```

The interpeter mainly includes two functions: `eval`, which evaluates an expression, and `evalD`, which evaluates a list of declarations. Both functions return `Eval` types, where `Eval` is an alias of `ReaderT Context (Either EvalError)`.

```haskell
type Context = [(String, Val)]

data Val = IntVal Integer
         | BoolVal Bool
         | FVal (Maybe String, String, Exp) Context

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | DivByZero
               | NotAFun Val deriving (Show)

-- reader+either monad for interpreter functions
--   'reader' for remembering val/fun declarations in contexts
--   'either' for throwing evaluation errors
type Eval = ReaderT Context (Either EvalError)
```

```
-- evaluate a list of declarations
evalD :: DeclList -> Eval Context
-- TODO

-- evaluate expression
eval :: Exp -> Eval Val
-- TODO
```

The following test functions are provided to run `Eval` monads.

- `runD`: interprete a declaration list with `evalD d`.
- `runE`: interprete an expression with `eval e`.

```
-- run a list of declarations and print the resulting context
runD :: DeclList -> String
runD d = y
  where Right y = runReaderT x []
        x = do
              a <- evalD d
              return $ "answers:\n" ++ toString a ++ "\n"
            `catchError` (\e -> return $ show e ++ "\n")
        toString a = unlines $ map (\(x,v)-> "\t val " ++ x ++ " = " ++ show v) $ reverse a

-- run an expression and print the results
runE :: Exp -> String
runE e = y
  where Right y = runReaderT x []
        x = do
              a <- eval e
              return $ "answers: " ++ show a ++ "\n"
            `catchError` (\e -> return $ show e ++ "\n")
```

## 2 Pretty Printer (bonus question - 60 points)

The type `PrettyPrint` is an alias for `ReaderT String (Writer String) ()`. Implement two functions:

- `ppd` to generate a PrettyPrint from a declaration
- `ppe` to generate a PrettyPrint from an expression.

The test function `pp` is provided to run the PrettyPrint of a declaration list.

```
-- reader+writer monad for pretty printing
--   reader for remembering indentation
--   writer for generating string
type PrettyPrint = ReaderT String (Writer String) ()

-- run the PrettyPrint monad (of a declaration list) to return a string
pp :: DeclList -> String
pp lst = snd $ runWriter $ runReaderT (ppl lst) ""

-- pretty-print a list of declarations
ppl :: DeclList -> PrettyPrint
ppl (Decls decls) = mapM_ (\d -> ppd d >> tell "\n") decls

-- pretty-print a declaration
ppd :: Decl -> PrettyPrint
-- TODO

-- pretty-print an expression
ppe :: Exp -> PrettyPrint
-- TODO
```

## 3 Test Code

Use the following code to test your implementation. The parser 'prog' is provided in the file Parser.hs. If you do not implement the pretty-printer, you should comment out line 19.

```
 1 import AST
 2 import Eval
 3 import Parser
 4 import Data.Either
 5 
 6 main :: IO ()
 7 main = do
 8          let f = "test.txt"
 9 
10          -- read a list of declarations 'd' from file 'f'
11          d <- readFile f
12 
13          -- run 'prog' parser to parse 'd' to obtain an AST
14          let ast = runP prog d
15 
16          case ast of Left e  -> putStrLn $ show e   -- print parse error
17                      Right x -> do
18                                   putStrLn $ show x -- print the AST
19                                   putStrLn $ pp x   -- pretty-print the AST
20                                   putStrLn $ runD x -- eval x and print the context or eval error
```

The test file test.txt has the following content, where gcd and gcd' are two implementation of the greatest common divisor function.

```
fun fact x = if (x < 1) then 1 else x * fact (x - 1)

fun gcd x = fn y =>
   if x = y then x
   else if x < y then gcd x (y-x) else gcd (x-y) y

fun gcd' x = fn y =>
   let
     fun mod x = fn y => if x > y then x - (x/y) * y else x
   in
     if y = 0 then x else gcd' y (mod x y)
   end

val a = let val y = 10 in fact y end

val b = gcd 117 369

val c = gcd' 117 369
```

The following is the expected output of the test file, where

- the first part is the output of show x (new lines are added to fit in the page margin),
- the second part is the output of pp x, and
- the last part is the output of runD x.

```
fun fact x = if (x < 1) then 1 else (x * (fact (x - 1)))
fun gcd x = (fn y => if (x = y) then x
                     else if (x < y) then ((gcd x) (y - x)) else ((gcd (x - y)) y))
fun gcd' x = (fn y => (let [fun mod x = (fn y => if (x > y) then (x - ((x / y) * y)) else x)]
                       in if (y = 0) then x else ((gcd' y) ((mod x) y)) end))
```

```
val a = (let [val y = 10] in (fact y) end)
val b = ((gcd 117) 369)
val c = ((gcd' 117) 369)

fun fact x = if (x < 1)
             then 1
             else (x * (fact (x - 1)))
fun gcd x = fn y => if (x = y)
                    then x
                    else if (x < y)
                         then ((gcd x) (y - x))
                         else ((gcd (x - y)) y)
fun gcd' x = fn y => let
                        fun mod x = fn y => if (x > y)
                                            then (x - ((x / y) * y))
                                            else x
                     in
                        if (y = 0)
                        then x
                        else ((gcd' y) ((mod x) y))
                     end
val a = let
           val y = 10
        in
           (fact y)
        end
val b = ((gcd 117) 369)
val c = ((gcd' 117) 369)

answers:
         val fact = fn
         val gcd = fn
         val gcd' = fn
         val a = 3628800
         val b = 9
         val c = 9
```

## 4 Files to submit

Four files are provided: `Parser.hs`, `AST.hs`, `Eval.hs`, and `Main.hs`. You should implement functions in `Eval.hs` (interpreter functions) and `AST.hs` (pretty-print questions). Please submit all files to the dropbox.