Instructions

Homework 5

For this assignment, you will create an interpreter for a minimal imperative *WHILE* language in Haskell.

First, copy the following definitions into a new file called **hw5.hs**.

```
-- Necessary imports
import Control.Applicative ((<$>),liftA,liftA2)
import Data.Map
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.Language (emptyDef)
import Text.Parsec.String (Parser)
import qualified Text.Parsec.Token as P


--------- AST Nodes ---------

-- Variables are identified by their name as string
type Variable = String

-- Values are either integers or booleans
data Value = IntVal Int       -- Integer value
         | BoolVal Bool     -- Boolean value

-- Expressions are variables, literal values, unary and binary operations
data Expression = Var Variable                    -- e.g. x
              | Val Value                     -- e.g. 2
              | BinOp Op Expression Expression  -- e.g. x + 3
              | Assignment Variable Expression  -- e.g. x = 3

-- Statements are expressions, conditionals, while loops and sequences
data Statement = Expr Expression                  -- e.g. x = 23
             | If Expression Statement Statement -- if e then s1 else s2 end
             | While Expression Statement        -- while e do s end
             | Sequence Statement Statement      -- s1; s2
             | Skip                              -- no-op

-- All binary operations
data Op = Plus         --  +  :: Int -> Int -> Int
       | Minus        --  -  :: Int -> Int -> Int
       | Times        --  *  :: Int -> Int -> Int
       | GreaterThan  --  >  :: Int -> Int -> Bool
       | Equals       --  == :: Int -> Int -> Bool
       | LessThan     --  <  :: Int -> Int -> Bool

-- The `Store` is an associative map from `Variable` to `Value` representing the memory
type Store = Map Variable Value
```

```
--------- Parser ---------

-- The Lexer

lexer = P.makeTokenParser (emptyDef {
  P.identStart = letter,
  P.identLetter = alphaNum,
  P.reservedOpNames = ["+", "-", "*", "!", ">", "=", "==", "<"],
  P.reservedNames = ["true", "false", "if", "in", "then", "else", "while", "end", "to", "do",
"for"]
})

-- The Parser

-- Number literals
numberParser :: Parser Value
numberParser = (IntVal . fromIntegral) <$> P.natural lexer

-- Boolean literals
boolParser :: Parser Value
boolParser =  (P.reserved lexer "true" >> return (BoolVal True))
          <|> (P.reserved lexer "false" >> return (BoolVal False))

-- Literals and Variables
valueParser :: Parser Expression
valueParser =  Val <$> (numberParser <|> boolParser)
           <|> Var <$> P.identifier lexer

-- -- Expressions
exprParser :: Parser Expression
exprParser = liftA2 Assignment
                  (try (P.identifier lexer >>= (\v ->
                        P.reservedOp lexer "=" >> return v)))
                  exprParser
          <|> buildExpressionParser table valueParser
    where table = [[Infix (op "*" (BinOp Times)) AssocLeft]
                  ,[Infix (op "+" (BinOp Plus)) AssocLeft]
                  ,[Infix (op "-" (BinOp Minus)) AssocLeft]
                  ,[Infix (op ">" (BinOp GreaterThan)) AssocLeft]
                  ,[Infix (op "==" (BinOp Equals)) AssocLeft]
                  ,[Infix (op "<" (BinOp LessThan)) AssocLeft]]
          op name node = (P.reservedOp lexer name) >> return node

-- Sequence of statements
stmtParser :: Parser Statement
stmtParser = stmtParser1 `chainl1` (P.semi lexer >> return Sequence)

-- Single statements
stmtParser1 :: Parser Statement
stmtParser1 = (Expr <$> exprParser)
          <|> do
              P.reserved lexer "if"
              cond <- exprParser
```

```
            P.reserved lexer "then"
            the <- stmtParser
            P.reserved lexer "else"
            els <- stmtParser
            P.reserved lexer "end"
            return (If cond the els)
         <|> do
            P.reserved lexer "while"
            cond <- exprParser
            P.reserved lexer "do"
            body <- stmtParser
            P.reserved lexer "end"
            return (While cond body)

-------- Helper functions --------

-- Lift primitive operations on IntVal and BoolVal values
liftIII :: (Int -> Int -> Int) -> Value -> Value -> Value
liftIII f (IntVal x) (IntVal y) = IntVal $ f x y
liftIIB :: (Int -> Int -> Bool) -> Value -> Value -> Value
liftIIB f (IntVal x) (IntVal y) = BoolVal $ f x y

-- Apply the correct primitive operator for the given Op value
applyOp :: Op -> Value -> Value -> Value
applyOp Plus        = liftIII (+)
applyOp Minus       = liftIII (-)
applyOp Times       = liftIII (*)
applyOp GreaterThan = liftIIB (>)
applyOp Equals      = liftIIB (==)
applyOp LessThan    = liftIIB (<)

-- Parse and print (pp) the given WHILE programs
pp :: String -> IO ()
pp input = case (parse stmtParser "" input) of
    Left err -> print err
    Right x  -> print x

-- Parse and run the given WHILE programs
run :: (Show v) => (Parser n) -> String -> (n -> Store -> v) -> IO ()
run parser input eval = case (parse parser "" input) of
    Left err -> print err
    Right x  -> print (eval x empty)

{-  Uncomment the following function for question #5 and #6

-- Parse and run the given WHILE programs using monads
runMonad :: String -> Maybe Store
runMonad input = proc (parse stmtParser "" input)
    where proc (Right x) = snd `fmap` runImperative (evalS_monad x) empty
          proc _         = Nothing

-}
```

Using these definitions, we are going to build up our evaluator in several iterations.

Important: When trying to compile the definitions on the lab computers, it will complain about a missing Text.Parsec library. This can be solved by running the following two lines in a shell:

```
$> cabal update
$> cabal install parsec
```

1. First, make all abstract syntax tree (AST) node types instances of the Show type class. You will have to implement the show function such that showing statements and expressions yields code that would be accepted by the parser.
2. instance Show Value where
3.   ...
4.
5. instance Show Op where
6.   ...
7.
8. instance Show Expression where
9.   ...
10.
11. instance Show Statement where
    ...

    Examples:

    ```
    > pp "1+1"
    1 + 1

    > pp "23*x<42"
    23 * x < 42

    > pp "if false then x=2 else x = 3 end ; x = x + 2"
    if false then x = 2 else x = 3 end ;x = x + 2

    > pp "x = 1; while x < 5 do x = x + 1 end"
    x = 1;while x < 5 do x = x + 1 end
    ```

    (10 points)

12. Write a function

    ```
    evalE :: Expression -> Store -> (Value, Store)
    ```

    that takes as input an expression and a store and returns a value. If a variable is not found (e.g. because it is not initialized) throw an error with the error function.

    The following case is given to you:

    ```
    evalE (BinOp o a b) s = (applyOp o a' b', s'')
        where (a', s')  = evalE a s
              (b', s'') = evalE b s'
    ```

You still have to write the following:

```
evalE (Var x) s = ...
evalE (Val v) s = ...
evalE (Assignment x e) s = ...
```

Examples:

```
> evalE (Val (BoolVal True)) Data.Map.empty
(true,fromList [])

> run exprParser "1+1" evalE
(2,fromList [])

> run exprParser "13*2 < 27" evalE
(true,fromList [])

> evalE (Var "x") (fromList [("x",IntVal 23)])
(23,fromList [("x",23)])

> run exprParser "x = 23" evalE
(23,fromList [("x",23)])

> run exprParser "x = y = 2 + 3" evalE
(5,fromList [("x",5),("y",5)])
```

Hint: Use `Data.Map.lookup symbol map` to lookup a variable in the map and `Data.Map.insert key value map` to insert a variable with the provided key into the map.

(15 points)

13. Next, write a function

```
evalS :: Statement -> Store -> Store
```

that takes as input a statement and a store and returns a possibly modified store.

The following case is given to you:

```
evalS w@(While e s1) s = case (evalE e s) of
                    (BoolVal True,s')  -> let s'' = evalS s1 s' in evalS w s''
                    (BoolVal False,s') -> s'
                    _                  -> error "Condition must be a BoolVal"
```

You still have to write the following

```
evalS Skip s             = ...
evalS (Expr e) s         = ...
evalS (Sequence s1 s2) s = ...
evalS (If e s1 s2) s     = ...
```

In the `If` case, if `e` evaluates to a non-boolean value, throw an error using the `error` function.

Examples:

```
> run stmtParser "x=1+1" evalS
fromList [("x",2)]

> run stmtParser "x = 2; x = x + 3" evalS
fromList [("x",5)]

> run stmtParser "if true then x = 1 else x = 2 end" evalS
fromList [("x",1)]

> run stmtParser "x=2; y=x + 3; if y < 4 then z = true else z = false end" evalS
fromList [("x",2),("y",5),("z",false)]

> run stmtParser "x = 1; while x < 3 do x = x + 1 end" evalS
fromList [("x",3)]

> run stmtParser "x = 1 ; y = 1; while x < 5 do x = x + 1 ; y = y * x end" evalS
fromList [("x",5),("y",120)]
```

(15 points)

14. We use the Maybe type to deal with cases like uninitialized variables or non-boolean tests instead of throwing a runtime error.

    Write the following function that takes an expression and store and returns a Maybe (Value,Store) and never throws a runtime error.

    The following case is given to you:

```
evalE_maybe (BinOp o a b) s = do (a',s') <- evalE_maybe a s
                                 (b',s'') <- evalE_maybe b s'
                                 return (applyOp o a' b', s'')
```

    You still have to write:

```
evalE_maybe :: Expression -> Store -> Maybe (Value, Store)
evalE_maybe (Var x) s          = ..
evalE_maybe (Val v) s          = ...
evalE_maybe (Assignment x e) s = ...
```

    Similarly for statements

```
evalS_maybe :: Statement -> Store -> Maybe Store
evalS_maybe (While e s1) s      = ...
evalS_maybe Skip s              = ...
evalS_maybe (Sequence s1 s2) s = ...
evalS_maybe (Expr e) s          = ...
evalS_maybe (If e s1 s2) s      = ...
```

    Examples:

```
> run exprParser "1+1" evalE_maybe
Just (2,fromList [])

> run exprParser "10 < x + 1" evalE_maybe
Nothing

> run exprParser "10 == 4 * 2" evalE_maybe
Just (false,fromList [])

> run stmtParser "x = 2; y = z" evalS_maybe
Nothing

> run stmtParser "x = true; if x then y = 1 else y = 2 end" evalS_maybe
Just (fromList [("x",true),("y",1)])

> run stmtParser "x = 1; if x then y = 1 else y = 2 end" evalS_maybe
Nothing
```

(15 points)

15. Here is a nice helpful monad that combines Maybe with a pending computation which requires a store to start processing.

```
newtype Imperative a = Imperative {
   runImperative :: Store -> Maybe (a, Store)
}

instance Monad Imperative where
   return a = Imperative (\s -> Just (a,s))
   b >>= f = Imperative (\s -> do (v1,s1) <- (runImperative b) s
                                  runImperative (f v1) s1)
   fail _ = Imperative (\s -> Nothing)
```

Rewrite the evaluator in this monad:

```
evalE_monad :: Expression -> Imperative Value
evalS_monad :: Statement -> Imperative ()
```

The following case is given to you:

```
evalE_monad (BinOp o a b) = do a' <- evalE_monad a
                               b' <- evalE_monad b
                               return (applyOp o a' b')
```

You still have to write:

```
evalE_monad :: Expression -> Imperative Value
evalE_monad (Var x) = ..
evalE_monad (Val v) = ...
evalE_monad (Assignment x e) = ...
```

Similarly for statements

```
evalS_monad :: Statement -> Imperative ()
evalS_monad (While e s1)     = ...
evalS_monad Skip             = ...
evalS_monad (Sequence s1 s2) = ...
evalS_monad (Expr e)         = ...
evalS_monad (If e s1 s2)     = ...
```

For the assignment and variable references, you need to return a monad that actually accesses the store as a map. It might help to define two function `getVar` and `setVar` first that do this kind of "dirty work".

```
getVar :: Variable -> Imperative Value
getVar var = Imperative (\store -> ((Data.Map.lookup var store) >>= (\v -> Just
(v,store))))

setVar :: Variable -> Value -> Imperative Value
setVar var val = Imperative (\store -> Just (val, Data.Map.insert var val store))
```

By using these methods, the rest of your code should consist of clean and easy to understand "do" blocks. The following example creates a monad called "miniprog" that basically does "x = 2; y = 3; return 2+3;":

```
miniprog :: Imperative Value
miniprog = do
        setVar "x" (IntVal 2)
        setVar "y" (IntVal 3)
        a <- getVar "x"
        b <- getVar "y"
        return (applyOp Plus a b)
```

You can run this monad with an empty initial store with

```
> runImperative miniprog Data.Map.empty
Just (5,fromList [("x",2),("y",3)])
```

Uncomment the function `runMonad` in the provided code to run the following examples:

```
> runMonad "x = 1"
Just (fromList [("x", 1)])

> runMonad "x = 1; if x == 1 then y = 1 else y = 2 end"
Just (fromList [("x", 1),("y", 1)])

> runMonad "x = 1; if x == z then y = 1 else y = 2 end"
Nothing

> runMonad "while 23 x = x + 1 end"
Nothing
```

(25 points)

25. Extend the `Statement` data type and the parser by adding a `for` loop to the language. The syntax is

```
for var in e1 to e2 do s end
```

where var is a variable name, e1 and e2 are expressions and s is a statement.

The semantics of this for loop is that the variable with the name var will be set to an initial value given by whatever the expression e1 returns and will execute the body s repeatedly, increasing the loop variable after each iteration, until the variable var is greater than the value returned by expression e2.

In addition to parsing, you also have to
implement `show`, `evalS`, `evalS_maybe` and `evalS_monad` for your new loop.

Examples:

```
> (For "a" (Val (IntVal 1)) (Val (IntVal 2)) (Expr (Val (IntVal 3))))
for a in 1 to 2 do 3 end

> parse stmtParser "" "for x in 1 to 4 do y = x end"
Right for x in 1 to 4 do y = x end

> run stmtParser "for x in 1 to 4 do y = x end" evalS
fromList [("x",5),("y",4)]

> run stmtParser "for x in 1 to 4 do y = x end" evalS_maybe
Just (fromList [("x",5),("y",4)])

> run stmtParser "for x in 1 to 4 do y = z end" evalS_maybe
Nothing

> runMonad "for x in 1 to 4 do y = x end"
Just (fromList [("x",5),("y",4)])

> runMonad "for x in 1 to 4 do y = z end"
Nothing
```

(20 points)