Interpreter Activity 2

Mattox Beckman

Here is part of the code for the `i4.hs` interpreter.

```haskell
-- The Types

data Val = IntVal Integer
   deriving (Show,Eq)

data Exp = IntExp Integer
         | IntOpExp String Exp Exp
         | VarExp String
         | LetExp String Val Exp
   deriving (Show,Eq)

type Env = [(String,Val)]

-- Evaluator

intOps = [ ("+",(+))
         , ("-",(-))
         , ("*",(*))
         , ("/",div)]

liftIntOp f (IntVal i1) (IntVal i2) = IntVal (f i1 i2)
liftIntOp f _           _           = IntVal 0

eval :: Exp -> Env -> Val
eval (IntExp i) _ = IntVal i

eval (IntOpExp op e1 e2) env =
  let v1 = eval e1 env
      v2 = eval e2 env
      Just f = lookup op intOps
  in liftIntOp f v1 v2

eval (VarExp v) env =
  case lookup v env of
    Just vv -> v
    Nothing -> IntVal 0

eval (LetExp var e1 e2) env =
  let v1 = eval e1 env
   in eval e2 (var,v1):env
```

1. With a partner, code review this. Two lines have errors, and they are different ones than from last time! Find them and correct them.

2. Add anonymous functions to the language. A function expression should have a string for the parameter and an expression for the body of the function.

   A function value is called a *closure*. It consists of a string for the parameter, an expression for the body, and the environment as it existed when the function was created.

   Why do you think we need to save the environment? (The interpreter one, not the literal one, though we should be trying to save it too.)

   Consider the following example as a hint:

```
1  let inc =
2     let delta = 1
3      in \ x -> x + delta
```

3. Now add function application. Assume a function takes just one argument for now, but if you find this too easy then by all means try to add support for multiple argument functions.

   Use call by value.