Interpreters, Part 2

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Define the Types – Types.hs

```haskell
data Exp = IntExp Integer
         | IntOpExp String Exp Exp
         | RelOpExp String Exp Exp
         | BoolOpExp String Exp Exp
         | BoolExp Bool
   deriving (Show, Eq)

data Val = IntVal Integer
         | BoolVal Bool
   deriving (Show, Eq)
```

Eval – Bools, and, or

```haskell
boolOps = [ ("&&",(&&))
          , ("||",(||))]

liftBoolOp f (BoolVal i1) (BoolVal i2) = BoolVal (f i1 i2)
liftBoolOp f _            _            = BoolVal False

eval (BoolExp b) _ = BoolVal b

eval (BoolOpExp op e1 e2) env =
  let v1 = eval e1 env
      v2 = eval e2 env
      Just f = lookup op boolOps
   in liftBoolOp f v1 v2
```

Adding Comparisons

```haskell
relOps = [ ("<", (<)) , ("<=", (<=)) , (">", (>))
         , (">=", (<=)) , ("==", (<=)) , ("/=", (/=)) ]

liftRelOp f (IntVal i1) (IntVal i2) = BoolVal (f i1 i2)
liftRelOp f _           _           = BoolVal False

eval (RelOpExp op e1 e2) env =
  let v1 = eval e1 env
      v2 = eval e2 env
      Just f = lookup op relOps
   in liftRelOp f v1 v2
```

A Simple Let Expression

- We want to define local variables:

```
i4> 3 + let x = 2 + 3 in x * x end
IntVal 28
```

- Need two new `Exp` constructors.

```
data Exp = VarExp String
         | LetExp String Exp Exp
         | ...
```

Coding Eval for Variables

- For variables, we look them up in the environment.

```
eval (VarExp var) env =
  case lookup var env of
     Just val -> val
     Nothing -> IntVal 0
```

Coding Eval for Let

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

- The `insert var v1 env` call acts like pushing a value onto a stack!

Next Time

- You now have some interesting things for your interpreter.
- The reference implementation is in `i4`.
- We've also added a `IfExp` to the types if you want to try adding this.