Objectives
OOOO

If Expressions
OO

Functions
OOOOOOO
OOOOO

Conclusions
O

Closures

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives
●○○○

If Expressions
○○

Functions
○○○○○○○
○○○○○

Conclusions
○

Objectives

- ▶ Add conditional expressions (if then else) to your language.
- ▶ Add functions and function application to your interpreter.
- ▶ Explain the parts of a closure and why they are necessary.

Objectives
○●○○

If Expressions
○○

Functions
○○○○○○○
○○○○○

Conclusions
○

Review

- Last time: made an interpreter with arithmetic, booleans, variables, and `let`.
- This time:
  - Add `if` expressions.
  - Add functions and function calls.
- Code can be found in the `i5` directory.

Objectives
○○●○

If Expressions
○○

Functions
○○○○○○○
○○○○○

Conclusions
○

Variables and Let Expressions

```
1 eval (VarExp var) env =
2     case lookup var env of
3         Just val -> val
4         Nothing -> IntVal 0
5
6 eval (LetExp var e1 e2) env =
7     let v1 = eval e1 env
8     in eval e2 (insert var v1 env)
```

▶ **N.B.** The variable let creates disappears after the let body is evaluated!!

For Example

In HASKELL ...

```
1 Prelude> let z = 10 in z + 1
2 11
3 Prelude> z
4 <interactive>:2:1: error: Variable not in scope: z
```

In i5...

```
i5> let z = 10 in z + 1 end
IntVal 11
i5> z
IntVal 0
```

Objectives
0000

If Expressions
●○

Functions
0000000
00000

Conclusions
○

Adding If Expressions

```
1 data Exp = IfExp Exp Exp Exp
2          | ...

i5> if 5 > 2 then 10 else 20 fi
IntVal 10
i5> if 5 > 22 then 10 else 20 fi
IntVal 20
```

Objectives
0000

If Expressions
○●

Functions
0000000
00000

Conclusions
○

The Eval

```
1 eval (IfExp e1 e2 e3) env =
2   let v1 = eval e1 env
3     in case v1 of
4         BoolVal True -> eval e2 env
5         _            -> eval e3 env
```

Objectives
0000

If Expressions
00

Functions
●000000
00000

Conclusions
0

Adding Functions to Our Language

▶ Consider this function application in HASKELL.

```
1 (\x -> x + 10)   20
```

▶ We have:
  ▶ A *parameter*
  ▶ A *function body*
  ▶ An *argument*

Adding Functions: Take 1

```
1 (\x -> x + 10)   20
2 => AppExp
3     (FunExp "x" (IntOpExp "+" (VarExp "x") (IntExp 10)))
4     (IntExp 20)
```

▶ The following attempt almost works.

```
1 data Exp = FunExp String Exp
2          | AppExp Exp Exp | ...
3 data Val = FunVal String Exp | ...
4
5 eval (FunExp v body) env = FunVal v body
6 eval (AppExp e1 e2) env =
7     let (FunVal v body) = eval e1 env
8         arg = eval e2 env
9       in eval body (insert v arg env)
```

What Could Possibly Go Wrong?

▶ Consider this function definition and function call.

```
1 Main> let f =
2         \ x -> x + 10
3     in f 20
4 30
```

▶ Now we use a second let to define the increment.

```
1 Main> let f =
2         let delta = 10
3           in \ x -> x + delta
4       in f 20
5 30
```

▶ When we run f 20, is delta still in scope?

The Need for Closures

- ► Now consider this one. We have *two* variables called `delta`!
- ► How does the function know which one to use?

```
1 Main> let f =
2         let delta = 10 in \ x -> x + delta
3       in
4         let delta = 20 in f 20
5 30  --- Why not 40??
```

Closures

- The "function value" needs to remember the values of free variables in its function body.
- The resulting data structure is called a *closure*.

```
1 data Exp = FunExp String Exp
2          | AppExp Exp Exp | ...
3 data Val = Closure String Exp Env | ...
4
5 eval (FunExp v body) env = Closure v body env
6 eval (AppExp e1 e2) env =
7    let (Closure v body clenv) = eval e1 env
8        arg = eval e2 env
9    in eval body (insert v arg clenv)
```

An Example Evaluation

▶ Let's evaluate this expression:

```
let d = 10 in \ x -> x + d
```

▶ Initial call to eval:

```
eval (LetExp "d" (IntExp 10)
    (FunExp "x" (IntOpExp "+"
                    (VarExp "x") (VarExp "d"))))
    []
```

▶ Step 1: *eval* will be called on the IntExp 10 to get the value of d.

```
eval (IntExp 10) [] => IntVal 10
```

Objectives
0000

If Expressions
00

Functions
000000●
00000

Conclusions
○

Example, Continued

▶ Now d is part of the environment when we evaluate the body of the let.

```
eval (FunExp "x" (IntOpExp "+"
                     (VarExp "x")
                     (VarExp "d")))
    [("d",IntVal 10)]
=> Closure "x" (IntOpExp "+"
                   (VarExp "x")
                   (VarExp "d"))
    [("d",IntVal 10)]
```

Now Let's Call the Function!

```
let f =
    let d = 10 in \ x -> x + d
 in let y = 20 in f y

eval (LetExp "f"
        (LetExp "d" (IntExp 10)
           (FunExp "x"
              (IntOpExp "+"
                 (VarExp "x") (VarExp "d"))))
        (LetExp "y" (IntExp 20)
           (AppExp (VarExp "f") (VarExp "y"))))
     []
```

Objectives
0000

If Expressions
00

Functions
0000000
0●000

Conclusions
0

Now Let's Call the Function! Pt 2

- After the function has been evaluated into a closure ...

```
eval (LetExp "y" (IntExp 20)
          (AppExp (VarExp "f") (VarExp "y")))
      [("f", Closure "x"
              (IntOpExp "+"
                 (VarExp "x") (VarExp "d"))
              [("d",IntVal 10)]]
```

Objectives
0000

If Expressions
OO

Functions
0000000
00●00

Conclusions
O

Now Let's Call the Function! Pt 3

- ▶ After the function has been evaluated into a closure …
- ▶ And y has been defined …

```
eval (AppExp (VarExp "f") (VarExp "y"))
     [("y",IntVal 20)
     ,("f", Closure "x"
              (IntOpExp "+"
                 (VarExp "x") (VarExp "d"))
              [("d",IntVal 10)]]
```

Reminder of the Code

```
eval (AppExp (VarExp "f") (VarExp "y"))
     [("y",IntVal 20)
     ,("f", Closure "x"
                (IntOpExp "+"
                    (VarExp "x") (VarExp "d"))
                [("d",IntVal 10)]]
```

▶ Remember what eval says to do with function calls.

```
1 eval (AppExp e1 e2) env =
2     let (Closure v body clenv) = eval e1 env
3         arg = eval e2 env
4      in eval body (insert v arg clenv)
```

Now Let's Call the Function! Pt 4

```
eval (AppExp (VarExp "f") (VarExp "y"))
     [("y",IntVal 20)
     ,("f", Closure "x"
               (IntOpExp "+"
                   (VarExp "x") (VarExp "d"))
               [("d",IntVal 10)]]
```

► We unfold the f and y values …

```
eval (IntOpExp "+" (VarExp "x") (VarExp "d"))
     [("x", eval (VarExp "y") [("y",IntVal 20)), ...]
     ,("d", IntVal 10)]
```

Conclusions

- ▶ Some history
  - ▶ The first language to use closures (and call them that) was Peter Landin's SECD machine.
  - ▶ The first widespread use of closures was in SCHEME, a dialect of LISP.
  - ▶ Today they are very common!
- ▶ Things to try
  - ▶ What if you wanted C-style ifs?
  - ▶ Try some other examples of function calls.
  - ▶ Try making multi-parameter functions.