Closures

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

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.

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.

Variables and Let Expressions

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

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

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

For Example

In HASKELL ...

```
Prelude> let z = 10 in z + 1
11
Prelude> z
<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
```

Adding If Expressions

```
data Exp = IfExp Exp Exp Exp
         | ...
```

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

The Eval

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

Adding Functions to Our Language

- Consider this function application in HASKELL.

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

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

Adding Functions: Take 1

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

- The following attempt almost works.

```
data Exp = FunExp String Exp
         | AppExp Exp Exp | ...
data Val = FunVal String Exp | ...

eval (FunExp v body) env = FunVal v body
eval (AppExp e1 e2) env =
    let (FunVal v body) = eval e1 env
        arg = eval e2 env
     in eval body (insert v arg env)
```

What Could Possibly Go Wrong?

- Consider this function definition and function call.

```
Main> let f =
          \ x -> x + 10
      in f 20
30
```

- Now we use a second `let` to define the increment.

```
Main> let f =
        let delta = 10
          in \ x -> x + delta
        in f 20
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?

```
Main> let f =
          let delta = 10 in \ x -> x + delta
        in
          let delta = 20 in f 20
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*.

```
data Exp = FunExp String Exp
         | AppExp Exp Exp | ...
data Val = Closure String Exp Env | ...

eval (FunExp v body) env = Closure v body env
eval (AppExp e1 e2) env =
    let (Closure v body clenv) = eval e1 env
        arg = eval e2 env
     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
```

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"))))
     []
```

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)]]
```

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.

```
eval (AppExp e1 e2) env =
    let (Closure v body clenv) = eval e1 env
        arg = eval e2 env
     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.