# Binding, Free Variables, and All That

CS4450/7450

# Syntax for Basic Scheme

```
<expr> ::= <ident>
<expr> ::= ( lambda ( <ident>* ) <expr> )
<expr> ::= ( <expr>* )
```

```
data Expr = Ident String
          | Lambda [String] Expr
          | Funcall [Expr]
```

# Binding rule for Scheme

```
( lambda ( <ident>) <expr> )
```

- The binder "`lambda (<ident>)`" binds all occurrences of `<ident>` within `<expr>` **unless**
  - there is an intervening declaration of the `<ident>`

Ex: **`(lambda (x) (+ x ((lambda (x) x) 4)))`**

# Free & bound occurrences

- A variable **`x`** *occurs free* in expression E iff there is some use of **`x`** not bound by a declaration in E
- A variable **`x`** *occurs bound* in E iff there is some occurrence of **`x`** bound in E
  - Which variable occurrences are free/bound in:
    - `((lambda (x) x) y)`
    - `(lambda (y) ((lambda (x) x) y))`
    - `((lambda (x) x) x)`

# Calculating the free variables

```
data Lam = Ident String
         | Lambda [String] Lam
         | Apply [Lam]

(* (lambda (x) (x y)) *)
e = Lambda ["x”] (Apply [Ident "x”,Ident "y"])
```

```
free (Ident x)          = …
free (Lambda args e)    = …
free (Apply es)         = …
```

Q: how do we tell whether `x` is free or bound?

# Calculating the free variables

```
lkup x []       = False
lkup x (y:ys) = x==y || lkup x ys

free seen (Ident x)          = …
free seen (Lambda args e)    = …
free seen (Apply es)         = …
```

A: include a list of variables known to be bound

# Calculating the free variables

```
lkup x []     = False
lkup x (y:ys) = x==y || lkup x ys

free seen (Ident x)       = if lkup x seen then [] else [x]
free seen (Lambda args e) = free (seen ++ args) e
free seen (Apply es)      = foldr (++) [] (map (free seen) es)
```

# Abstract & Concrete Syntax of Imp

```
type Name = String
type FunDefn = (Name,[Name],[Stmt])
type Prog = ([FunDefn],[Stmt])
data Stmt =
    Assign Name Exp
| If BExp [Stmt] [Stmt]
       …
| Return Exp

data Exp =
    Add Exp Exp …
  | FunCall Name [Exp]
data BExp =
    IsEq Exp Exp
          …
  | LitBool Bool
```

```
function double(n) {
  return n+n;
}

y := double(5);
```

# Variable Occurrences

```
function iseven(n) {
   if n==0  {return 1;} else {return isodd(n-1);}
}

function isodd(n) {
    if n==0 {return 0;} else {return iseven (n-1);}
}

x := iseven (m+n);
```

# Variable Binders

```
function iseven(n) {
   if n==0  {return 1;} else {return isodd(n-1);}
}

function isodd(n) {
    if n==0 {return 0;} else {return iseven (n-1);}
}

x := iseven (m+n);
```

```
let sum := 0 in { ... }
for i := e1,e2 { ... }
```

"Binders" are language constructs that define a name or variable.

# Scope of Variable Binders

```
function iseven(n) {
   if n==0  {return 1;} else {return isodd(n-1);}
}

function isodd(n) {
    if n==0 {return 0;} else {return iseven (n-1);}
}

x := iseven (m+n);
```

```
let sum := 0 in { ... }
```

```
for i := e1,e2 { ... }
```

Scopes of **isodd**, **sum**, and **i** binders in blue

# Scope of Variable Binders

```
function iseven(n) {
   if n==0  {return 1;} else {return isodd(n-1);}
}

function isodd(n) {
    if n==0 {return 0;} else {return iseven (n-1);}
}

x := iseven (m+n);
```

# Free Variable Occurrences

```
function iseven(n) {
   if n==0  {return 1;} else {return isodd(n-1);}
}

function isodd(n) {
    if n==0 {return 0;} else {return iseven (n-1);}
}

x := iseven (m+n);
```

Free variable occurrences

Not free variable occurrences