The Arbitrariness of Variable Names

- Are the following two programs equivalent?

```
(define (f x) (+ x 1))     (define (f y) (+ y 1))
(f 10)                     (f 10)
```

yes

argument is consistently renamed

The Arbitrariness of Variable Names

- Are the following two programs equivalent?

```
(define (f x) (+ x 1))     (define (f x) (+ y 1))
(f 10)                     (f 10)
```

no

not a use of the argument anymore

The Arbitrariness of Variable Names

- Are the following two programs equivalent?

```
(define (f x) (+ x 1))     (define (f y) (+ x 1))
(f 10)                     (f 10)
```

no

not a use of the argument anymore

The Arbitrariness of Variable Names

- Are the following two programs equivalent?

```
(define (f x) (+ y 1))     (define (f z) (+ y 1))
(f 10)                     (f 10)
```

yes

argument never used, so almost any name is ok

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x) (+ y 1))     (define (f y) (+ y 1))
(f 10)                     (f 10)
```

no

now a use of the argument

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x) (+ y 1))     (define (f x) (+ z 1))
(f 10)                     (f 10)
```

no

still an undefined variable, but a different one

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x)          (define (f z)
  (let ([y 10])          (let ([y 10])
    (+ x y)))              (+ z y)))
```

yes

argument is consistently renamed

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x)          (define (f x)
  (let ([y 10])          (let ([z 10])
    (+ x y)))              (+ x z)))
```

yes

local variable is consistently renamed

The Arbitrariness of Variable Names

- Are the following two programs equivalent?

```
(define (f x)           (define (f x)
  (let ([y 10])           (let ([x 10])
    (+ x y)))               (+ x x)))
```

no

local variable now hides the argument

The Arbitrariness of Variable Names

- Are the following two programs equivalent?

```
(define (f x)           (define (f y)
  (let ([y 10])           (let ([y 10])
    (+ x y)))               (+ y y)))
```

no

local variable now hides the argument

Free and Bound Variables

- A variable for the argument of a function or the name of a local variable is a ***binding occurrence***

```
(define (f x y) (+ x y z))

(let ([a 3][c 4]) (+ a b c))
```

Free and Bound Variables

- A use of a function argument or a local variable is a ***bound occurrence***

```
(define (f x y) (+ x y z))

(let ([a 3][c 4]) (+ a b c))
```

Free and Bound Variables

- A use of a variable that is not function argument or a local variable is a ***free variable***

```
(define (f x y) (+ x y z))

(let ([a 3][c 4]) (+ a b c))
```

Evaluating Let

... (**let** ([<id>$_1$ <val>$_1$]...[<id>$_k$ <val>$_k$]) <expr>$_a$) ...

→

... <expr>$_b$ ...

where <expr>$_b$ is <expr>$_a$ with **free** <id>$_i$ replaced by <val>$_i$

```
(let ([x 10]) (let ([x 2]) x))
→
(let ([x 2]) x)
→
2
```

Evaluating Let

... (**let** ([<id>$_1$ <val>$_1$]...[<id>$_k$ <val>$_k$]) <expr>$_a$) ...

→

... <expr>$_b$ ...

where <expr>$_b$ is <expr>$_a$ with **free** <id>$_i$ replaced by <val>$_i$

```
(let ([x 10])
  (let ([x (+ x 1)]) x))
```

Evaluating Let

... (**let** ([<id>$_1$ <val>$_1$]...[<id>$_k$ <val>$_k$]) <expr>$_a$) ...

→

... <expr>$_b$ ...

where <expr>$_b$ is <expr>$_a$ with **free** <id>$_i$ replaced by <val>$_i$

```
(let ([x 10])
  (let ([x (+ x 1)]) x))
→
(let ([x (+ 10 1)]) x)
→
(let ([x 11]) x)  →  11
```

Evaluating Function Calls, Revised

... (**define** (<id>$_0$ <id>$_1$...<id>$_k$) <expr>$_a$) **...**
... (<id>$_0$ <val>$_1$...<val>$_k$) **...**

→

... (**define** (<id>$_0$ <id>$_1$...<id>$_k$) <expr>$_a$) **...**
... <expr>$_b$ **...**

where <expr>$_b$ is <expr>$_a$ with **free** <id>$_i$ replaced by <val>$_i$

Local Functions

Recall that

(**define** <id>$_0$ (**lambda** (<id>$_1$...<id>$_k$) <expr>))

is shorthand for

(**define** (<id>$_0$ <id>$_1$...<id>$_k$) <expr>)

New rule: **lambda** is allowed in **let** bindings to define local functions:

(**let** ([**f** (**lambda** (**x**) (+ **x** 1))])
  (**f** 10))

Evaluation of Local Functions

(**let** ([**f** (**lambda** (**x**) (+ **x** 1))])
  (**f** 10))
→
(**define f**$_{1073}$ (**lambda** (**x**) (+ **x** 1)))
(**f**$_{1073}$ 10)
→
(**define f**$_{1073}$ (**lambda** (**x**) (+ **x** 1)))
(+ 10 1)
→
11

Evaluation of Local Functions

...
... (**let** ([<id> (**lambda** (<id>$_1$...<id>$_k$) <expr>)]) <expr>$_a$) **...**
→
... (**define** (<id>$_x$ <id>$_1$...<id>$_k$) <expr>)
... <expr>$_b$ **...**

where <expr>$_b$ is <expr>$_a$ with free <id> replaced by <id>$_x$ and $_x$ is a subscript that has never been used before, and never will be used again

Lexical Scope

```
(define (f x)
  (let ([g (lambda (y) (+ y x))])
    (let ([x 2])
      (g 3))))
(f 7)
```

Will x be 7 or 2 ?

7, due to ***lexical scope***: the value of a bound occurrence comes from its binding

Need a complete definition of ***free*** and ***bound***...

Free and Bound Variables in Scheme

For simplicity, we consider a variant of Scheme that is more restricted than usual:

```
<expr>  ::=  <num>
        ::=  <id>
        ::=  (+ <expr> <expr>)
        ::=  (let ([<id> <expr>]) <expr>)
        ::=  (let ([<id> (lambda (<id>) <expr>)]) <expr>)
        ::=  (<id> <expr>)
```

Free Variables in Scheme

- <num> has no free variables
- <id> has one free variable: <id>
- (+ <expr>$_1$ <expr>$_2$) has all the free variables of <expr>$_1$ and <expr>$_2$ combined
- (**let** ([<id>$_a$ <expr>$_b$]) <expr>$_a$) has all the free variables of <expr>$_a$, but without <id>$_a$, plus all the free variables of <expr>$_b$
- (**let** ([<id>$_a$ (**lambda** (<id>$_b$) <expr>$_b$)]) <expr>$_a$) has all the free variables of <expr>$_a$, but without <id>$_a$, plus all the free variables of <expr>$_b$, but without <id>$_b$
- (<id> <expr>) has all the free variable <id> plus all the free variables of <expr>

Free Variables in Scheme

See implementation in Scheme

Reviews **`define-datatype`** motivation and use

Bound Variables in Scheme

- <num> has no bound variables
- <id> has no bound variables
- (+ <expr>$_1$ <expr>$_2$) has all the bound variables of <expr>$_1$ and <expr>$_2$ combined
- (**let** ([<id>$_a$ <expr>$_b$]) <expr>$_a$) has the bound variable <id>$_a$ if it is free in <expr>$_a$, plus all the bound variables of <expr>$_a$ and <expr>$_b$
- (**let** ([<id>$_a$ (**lambda** (<id>$_b$) <expr>$_b$)]) <expr>$_a$) has the bound variable <id>$_a$ if it is free in <expr>$_a$, plus the bound variable <id>$_b$ if it is free in <expr>$_b$, plus all the bound variables of <expr>$_a$ and <expr>$_b$
- (<id> <expr>) has all the bound variables of <expr>

let*

let$*$ is a shorthand for nested **let**s

(**let**$*$ ([<id>$_1$ <expr>$_1$]...[<id>$_k$ <expr>$_k$]) <expr>)

=

(**let** ([<id>$_1$ <expr>$_1$]) ... (**let** ([<id>$_k$ <expr>$_k$]) <expr>)...)

(**let** ([**x** 1][**y** **x**][**z** **y**]) **z**) →→ **undefined variable x**

(**let**$*$ ([**x** 1][**y** **x**][**z** **y**]) **z**) →→ 1

letrec

letrec binds its identifiers in local function bodies, as well as the main body

...

... (**letrec** ([<id> (**lambda** (<id>$_1$...<id>$_k$) <expr>$_c$)]) <expr>$_a$) **...**

→

... (**define** (<id>$_x$ <id>$_1$...<id>$_k$) <expr>$_d$)

... <expr>$_b$ **...**

where <expr>$_b$ is <expr>$_a$ with free <id> replaced by <id>$_x$, <expr>$_d$ is <expr>$_c$ with free <id> replaced by <id>$_x$ and $_x$ is a subscript that has never been used before, and never will be used again

Free Variables with letrec

- (**letrec** ([<id>$_a$ (**lambda** (<id>$_b$) <expr>$_b$)]) <expr>$_a$) has all the free variables of <expr>$_a$, but without <id>$_a$, plus all the free variables of <expr>$_b$, but without <id>$_a$ and <id>$_b$

Bound Variables with letrec

- (**let** ([<id>$_a$ (**lambda** (<id>$_b$) <expr>$_b$)]) <expr>$_a$) has the bound variable <id>$_a$ if it is free in <expr>$_a$ or <expr>$_b$, plus the bound variable <id>$_b$ if it is free in <expr>$_b$, plus all the bound variables of <expr>$_a$ and <expr>$_b$

Language EoPL 3.4

<expr> ::= <num>

::= <id>

::= <prim> (<expr>$^{*(,)}$)

::= **if** <expr> **then** <expr> **else** <expr>

::= **let** { <id> = <expr> }* **in** <expr>

Language EoPL 3.4

```
(define-datatype expression expression?
  (lit-exp
   (datum number?))
  (var-exp
   (id symbol?))
  (primapp-exp
   (rator primitive?)
   (rands (list-of expression?)))
  (if-exp
   (test-exp expression?)
   (then-exp expression?)
   (else-exp expression?))
  (let-exp
   (ids (list-of symbol?))
   (rands (list-of expression?))
   (body expression?)))
```

Free Variables in EoPL 3.4

- (**lit–exp** <num>) has no free variables

- (**var–exp** <symbol>) has one free variable: <symbol>

- (**primapp-exp** <prim> (**list** <expr>$_1$ **...** <expr>$_n$)) has all the free variables of <expr>$_1$ through <expr>$_n$ combined

- (**if–exp** <expr>$_1$ <expr>$_2$ <expr>$_3$) has all the free variables of <expr>$_1$ through <expr>$_3$ combined

- (**let–exp** (**list** <symbol>$_1$ **...** <symbol>$_n$)
  (**list** <expr>$_1$ **...** <expr>$_n$)
  <expr>$_0$) has all the free variables of <expr>$_0$, but without <symbol>$_1$ through <symbol>$_n$, plus all the free variables of <expr>$_1$ through <expr>$_n$

Bound Variables in EoPL 3.4

- (**lit–exp** <num>) has no bound variables
- (**var–exp** <symbol>) has no bound variables
- (**primapp-exp** <prim> (**list** $<expr>_1$ **...** $<expr>_n$)) has all the bound variables of $<expr>_1$ through $<expr>_n$ combined
- (**if–exp** $<expr>_1$ $<expr>_2$ $<expr>_3$) has all the bound variables of $<expr>_1$ through $<expr>_3$ combined
- (**let–exp** (**list** $<symbol>_1$ **...** $<symbol>_n$)
  (**list** $<expr>_1$ **...** $<expr>_n$)
  $<expr>_0$) has all the bound variables of $<expr>_0$ through $<expr>_n$, plus any of $<symbol>_1$ through $<symbol>_n$ that are free variables of $<expr>_0$