The Arbitrariness of Variable Names

- Are the following two programs equavalent?

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

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

yes

argument is consistently renamed

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

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

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

no

not a use of the argument anymore

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

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

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

no

not a use of the argument anymore

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

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

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

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

```
(define (f x) (+ z 1))
(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)
  (let ([y 10])
    (+ x y)))
(f 0)
```

```
(define (f z)
  (let ([y 10])
    (+ z y)))
(f 0)
```

yes

argument is consistently renamed

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x)
  (let ([y 10])
    (+ x y)))
(f 0)
```

```
(define (f x)
  (let ([z 10])
    (+ x z)))
(f 0)
```

yes

local variable is consistently renamed

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x)              (define (f x)
  (let ([y 10])              (let ([x 10])
    (+ x y)))                  (+ x x)))
(f 0)                      (f 0)
```

no

local variable now hides the argument

The Arbitrariness of Variable Names

- Are the following two programs equavalent?

```
(define (f x)              (define (f y)
  (let ([y 10])              (let ([y 10])
    (+ x y)))                  (+ y y)))
(f 0)                      (f 0)
```

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>$_a$) **...**

$\rightarrow$

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

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

(**let** ([**x** 10]) (**let** ([**x** 2]) **x**))
$\rightarrow$
(**let** ([**x** 2]) **x**)
$\rightarrow$
2

Evaluating Let

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

$\rightarrow$

... <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>$_a$) **...**

$\rightarrow$

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

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

(**let** ([**x** 10])
  (**let** ([**x** (+ **x** 1)]) **x**))
$\rightarrow$
(**let** ([**x** (+ 10 1)]) **x**)
$\rightarrow$
(**let** ([**x** 11]) **x**) $\rightarrow$ 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 f1073 (lambda (x) (+ x 1)))
(f1073 10)
```
→
```
(define f1073 (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>
- (+ $\langle expr \rangle_1$ $\langle expr \rangle_2$) has all the free variables of $\langle expr \rangle_1$ and $\langle expr \rangle_2$ combined
- (**let** ([$\langle id \rangle_a$ $\langle expr \rangle_b$]) $\langle expr \rangle_a$) has all the free variables of $\langle expr \rangle_a$ minus $\langle id \rangle_a$, plus all the free variables of $\langle expr \rangle_b$
- (**let** ([$\langle id \rangle_a$ (**lambda** ($\langle id \rangle_b$) $\langle expr \rangle_b$)]) $\langle expr \rangle_a$) has all the free variables of $\langle expr \rangle_a$ minus $\langle id \rangle_a$, plus all the free variables of $\langle expr \rangle_b$ minus $\langle id \rangle_b$
- (<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 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 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**) $\rightarrow\rightarrow$ **undefined variable x**

(**let*** ([**x** 1][**y x**][**z y**]) **z**) $\rightarrow\rightarrow$ 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>$_a$) **...**
$\rightarrow$
... (**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>$_a$ minus <id>$_a$, plus all the free variables of <expr>$_b$ minus <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>$_b$