

Opening Thought

Why must functions always have a name?

Anonymous Functions

From now on, functions can be anonymous

- Old code

```
(define (eval-rands rands env fenv)
  (let ([eval-one (lambda (rand)
                    (eval-expression rand env fenv))])
    (map eval-one rands)))
```

- New code

```
(define (eval-rands rands env fenv)
  (map (lambda (rand) (eval-expression rand env fenv))
       rands))
```

Lambda as an Expression

To support anonymous functions, we must first

- allow **(lambda (<id>*) <expr>)** as an expression
- change the application grammar to **(<expr> <expr>*)**

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

```
<val> ::= <num>
        ::= (lambda (<id>*) <expr>)
```

Evaluation with Lambda Expressions

Now we need only one kind of **let** form

```
(let ([identity (lambda (x) x)])
  (identity 5))
```

→

```
((lambda (x) x) 5)      usual substitution with values
```

→

```
5      new procedure application rule...
```

New Application Rule

$$\dots ((\text{lambda } (\langle \text{id} \rangle_1 \dots \langle \text{id} \rangle_k) \langle \text{expr} \rangle_a) \langle \text{val} \rangle_1 \dots \langle \text{val} \rangle_k) \dots$$
$$\rightarrow$$
$$\dots \langle \text{expr} \rangle_b \dots$$

where $\langle \text{expr} \rangle_b$ is $\langle \text{expr} \rangle_a$ with free $\langle \text{id} \rangle_i$ replaced by $\langle \text{val} \rangle_i$

$$((\text{lambda } (x) x) 5) \rightarrow 5$$

New Application Rule

$$\dots ((\text{lambda } (\langle \text{id} \rangle_1 \dots \langle \text{id} \rangle_k) \langle \text{expr} \rangle_a) \langle \text{val} \rangle_1 \dots \langle \text{val} \rangle_k) \dots$$
$$\rightarrow$$
$$\dots \langle \text{expr} \rangle_b \dots$$

where $\langle \text{expr} \rangle_b$ is $\langle \text{expr} \rangle_a$ with free $\langle \text{id} \rangle_i$ replaced by $\langle \text{val} \rangle_i$

$$((\text{lambda } (x y) (+ x y)) 2 3) \rightarrow (+ 2 3) \rightarrow 5$$

Using Anonymous Functions

Using anonymous functions, we can easily feed a list of fish:

```
;; feed all fish 1 lb of food:  
(map (lambda (x) (+ x 1)) '(4 5 8))  
= '(5 6 9)
```

```
;; feed all fish 2 lbs of food:  
(map (lambda (x) (+ x 2)) '(5 6 9))  
= '(7 8 11)
```

Using Anonymous Functions

Using anonymous functions, we can easily feed a list of fish:

```
;; feed all fish 1 lb of food:  
(map (lambda (x) (+ x 1)) '(4 5 8))  
= '(5 6 9)
```

```
;; feed all fish 2 lbs of food:  
(map (lambda (x) (+ x 2)) '(5 6 9))  
= '(7 8 11)
```

Avoid cut-and-paste of the **lambda** expression?

Functions that Return Functions

```
;; make-feeder : <num> → (<num> → <num>)
(define (make-feeder amt)
  (lambda (x) (+ x amt)))

;; feed all fish 1 lb of food:
(map (make-feeder 1) '(4 5 8))
= '(5 6 9)

;; feed all fish 2 lbs of food:
(map (make-feeder 2) '(5 6 9))
= '(7 8 11)
```

Another Example with Procedures as Values

```
(let ([mk-add (lambda (x) (lambda (y) (+ x y)))]
      [add5 (mk-add 5)])
  (add5 7))
→
(let ([add5 ((lambda (x) (lambda (y) (+ x y))) 5)])
  (add5 7))
→
(let ([add5 (lambda (y) (+ 5 y))])
  (add5 7))
→
((lambda (y) (+ 5 y)) 7)
→
(+ 5 7) → 12
```

Terminology: First-Order and Higher-Order

- The procedures supported by top-level definitions are **first-order** procedures
 - A procedure cannot consume or produce a procedure
 - Methods in Java and procedures in Fortran are first-order
 - Functions C are first-order, but function pointers are values

Terminology: First-Order and Higher-Order

- The procedures supported by **lambda** are **higher-order** procedures
 - A procedure can return a procedure that returns a procedure that consumes a procedure that returns a procedure...
 - Procedures in Scheme are higher-order

Procedure Expressions in the Book Language

Concrete extensions:

```
<prog> ::= <expr>
<expr> ::= proc (<id>*(.)) <expr>
        ::= (<expr> <expr>*)
```

```
let identity = proc(x) x
in (identity 5)
→→ 5
```

Procedure Expressions in the Book Language

Concrete extensions:

```
<prog> ::= <expr>
<expr> ::= proc (<id>*(.)) <expr>
        ::= (<expr> <expr>*)
```

```
let sum = proc(x, y, z) +(x, +(y, z))
in (sum 10 20 30)
→→ 60
```

Procedure Expressions in the Book Language

Concrete extensions:

```
<prog> ::= <expr>
<expr> ::= proc (<id>*(.)) <expr>
        ::= (<expr> <expr>*)
```

```
(proc(x) x 5)
→→ 5
```

Procedure Expressions in the Book Language

Concrete extensions:

```
<prog> ::= <expr>
<expr> ::= proc (<id>*(.)) <expr>
        ::= (<expr> <expr>*)
```

```
let mkadd = proc(x) proc(y) +(x, y)
in let add5 = (mkadd 5)
in let x = 10
in (add5 6)
→→ 11
```

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = +(2, 3) in x	{}

- This trace shows the expression and environment arguments to `eval-expression`

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = +(2, 3) in x	{}
→+(2, 3)	{}

- Arrows show nested recursive calls

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = +(2, 3) in x	{}
→5	{}

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = +(2, 3) in x	{}
→5	{}

- Eventually a value is reached for each recursive call
- To continue with **let**, extend the environment and evaluate the body

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
x	{ x = 5 }

- Drop the context for the recursive body evaluation, since it isn't needed

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
5	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 5	
in let x = 6	{ }
in x	

- Another example: nested **let**

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 5	
in let x = 6	{ }
in x	
→5	{ }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 5	
in let x = 6	{ }
in x	
→5	{ }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 6	
in x	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 6	
in x	{ x = 5 }
→6	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 6	
in x	{ x = 5 }
→6	{ x = 5 }

- New value for **x** replaces the old one for the body

Evaluation with Environments

Expr Env
x { **x = 6** }

Evaluation with Environments

Expr Env
6 { **x = 6** }

Evaluation with Environments

Expr Env
let x = 5
in let y = let x = 6 in x {}
in x

- Another example: **let** nested in a different way

Evaluation with Environments

Expr Env
let x = 5
in let y = let x = 6 in x {}
in x
→5 {}

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let x = 5 in let y = let x = 6 in x in x	{ }
→5	{ }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→ let x = 6 in x	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→ let x = 6 in x	{ x = 5 }
→6	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→ let x = 6 in x	{ x = 5 }
→ 6	{ x = 5 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→ x	{ x = 6 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→ 6	{ x = 6 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→ 6	{ x = 6 }

- What environment is extended with **y = 6** ?

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
let y = let x = 6 in x in x	{ x = 5 }
→6	{ x = 6 }

- Answer: the original one for the **let** of **y**

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
x	{ x = 5, y = 6 }

Evaluation with Environments

<i>Expr</i>	<i>Env</i>
5	{ x = 5, y = 6 }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y) in let add5 = (mkadd 5) in (add5 6)	{ }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y) in let add5 = (mkadd 5) in (add5 6)	{ }
→proc(x) proc(y) +(x, y)	{ }

- Is a **proc** expression a value?
- A **lambda** was a value in Scheme... so let's say it's ok

this choice will turn out to be slightly wrong

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y) in let add5 = (mkadd 5) in (add5 6)	{ }
→proc(x) proc(y) +(x, y)	{ }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→(mkadd 5)	{ mkadd = proc(x) proc(y) +(x, y) }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→(mkadd 5)	{ mkadd = proc(x) proc(y) +(x, y) }
→mkadd	{ mkadd = proc(x) proc(y) +(x, y) }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→(mkadd 5)	{ mkadd = proc(x) proc(y) +(x, y) }
→proc(x) proc(y) +(x, y)	{ mkadd = proc(x) proc(y) +(x, y) }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→(mkadd 5)	{ mkadd = proc(x) proc(y) +(x, y) }
→proc(x) proc(y) +(x, y)	{ mkadd = proc(x) proc(y) +(x, y) }
→5	{ mkadd = proc(x) proc(y) +(x, y) }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→(mkadd 5)	{ mkadd = proc(x) proc(y) +(x, y) }
→proc(x) proc(y) +(x, y)	{ mkadd = proc(x) proc(y) +(x, y) }
→5	{ mkadd = proc(x) proc(y) +(x, y) }

- To evaluate an application, extend the application's environment with a binding for the argument

this isn't quite right, either

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→proc (y) +(x, y)	{ mkadd = proc(x) proc(y) +(x, y) x = 5 }

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = proc(x) proc(y) +(x, y) }
→proc (y) +(x, y)	{ mkadd = proc(x) proc(y) +(x, y) x = 5 }

- So the value for **add5** is also a procedure
- Extend the original environment for the **let**

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = proc(x) proc(y) +(x, y) add5 = proc (y) +(x, y) }

- We can see where this is going... **x** has no value
- What went wrong?

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = proc(x) proc(y) +(x, y) add5 = proc (y) +(x, y) }

- In Scheme, procedures as values worked because they had eager substitutions

Evaluation with Procedures and Environments

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = proc(x) proc(y) +(x, y) add5 = proc (y) +(x, y) }

- With lazy substitutions: combine a **proc** and an environment to get a value
- The combination is called a **closure**

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y) in let add5 = (mkadd 5) in (add5 6)	{ }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y) in let add5 = (mkadd 5) in (add5 6)	{ }
→proc(x) proc(y) +(x, y)	{ }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y) in let add5 = (mkadd 5) in (add5 6)	{ }
→<proc(x) proc(y) +(x, y), { }>	{ }

- Create a closure with the current environment to get a value

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y)	
in let add5 = (mkadd 5)	{ }
in (add5 6)	
→ <(x), proc(y) +(x, y), { }>	{ }

- Alternate form: arguments, body, and environment

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let mkadd = proc(x) proc(y) +(x, y)	
in let add5 = (mkadd 5)	{ }
in (add5 6)	
→ <(x), proc(y) +(x, y), { }>	{ }

- A closure is a value

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5)	
in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5)	
in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→ (mkadd 5)	{ mkadd = <(x), proc(y) +(x, y), { }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→(mkadd 5)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→mkadd	{ mkadd = <(x), proc(y) +(x, y), { }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→(mkadd 5)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→<(x), proc(y) +(x, y), { }>	{ mkadd = <(x), proc(y) +(x, y), { }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→(mkadd 5)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→<(x), proc(y) +(x, y), { }>	{ mkadd = <(x), proc(y) +(x, y), { }> }
→5	{ mkadd = <(x), proc(y) +(x, y), { }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→(mkadd 5)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→<(x), proc(y) +(x, y), { }>	{ mkadd = <(x), proc(y) +(x, y), { }> }
→5	{ mkadd = <(x), proc(y) +(x, y), { }> }

- To evaluate an application, extend the *closure's* environment with a binding for the argument

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→proc (y) +(x, y)	{ x = 5 }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→<(y), +(x, y), { x = 5 }>	{ x = 5 }

- Again, create a closure
- Note that the **x** binding is saved in the closure

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
let add5 = (mkadd 5) in (add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> }
→<(y), +(x, y), { x = 5 }>	{ x = 5 }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }
→add5	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }
→<(y), +(x, y), { x = 5 }>	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }
→<(y), +(x, y), { x = 5 }>	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }
→6	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }

Evaluation with Closures

<i>Expr</i>	<i>Env</i>
(add5 6)	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }
→<(y), +(x, y), { x = 5 }>	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }
→6	{ mkadd = <(x), proc(y) +(x, y), { }> add5 = <(y), +(x, y), { x = 5 }> }

- Extend the closure's environment { x = 5 } with a binding for y

Evaluation with Closures

Expr Env

$+(x, y)$ $\{ x = 5, y = 6 \}$

- This is clearly going to work

Environments in Picture Form

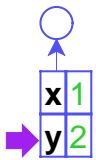


top purple arrow points to the current environment

purple in bottom area hilites the current expression

let x = 1 y = 2
in +(x, y)

Environments in Picture Form



top purple arrow points to the current environment

purple in bottom area hilites the current expression

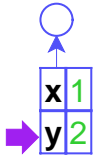
let x = 1 y = 2
in +(x, y)

Environments in Picture Form



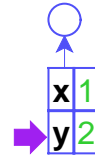
let x = 1 y = 2
in let f = proc (z) +(z, y)
in (f y)

Environments in Picture Form



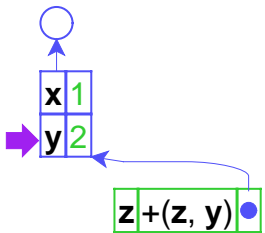
```
let x = 1 y = 2
  in let f = proc (z) +(z, y)
     in (f y)
```

Environments in Picture Form



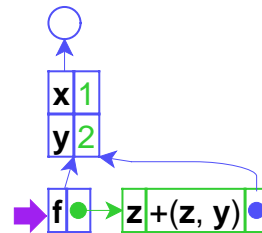
```
let x = 1 y = 2
  in let f = proc (z) +(z, y)
     in (f y)
```

Environments in Picture Form



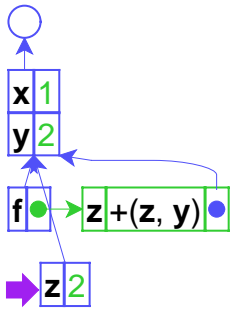
```
let x = 1 y = 2
  in let f = proc (z) +(z, y)
     in (f y)
```

Environments in Picture Form



```
let x = 1 y = 2
  in let f = proc (z) +(z, y)
     in (f y)
```

Environments in Picture Form



```
let x = 1 y = 2
in let f = proc (z) +(z, y)
in (f y)
```

Procedure Expressions in the Book Language

Abstract extensions:

```
<prog> ::= (a-program <expr>)
<expr> ::= (proc-exp (list <id>*) <expr>)
          ::= (app-exp <expr> (list <expr>*))
<val>   ::= <num>
          ::= <proc>
<proc> ::= (closure (list <id>*) <expr> <env>)
```