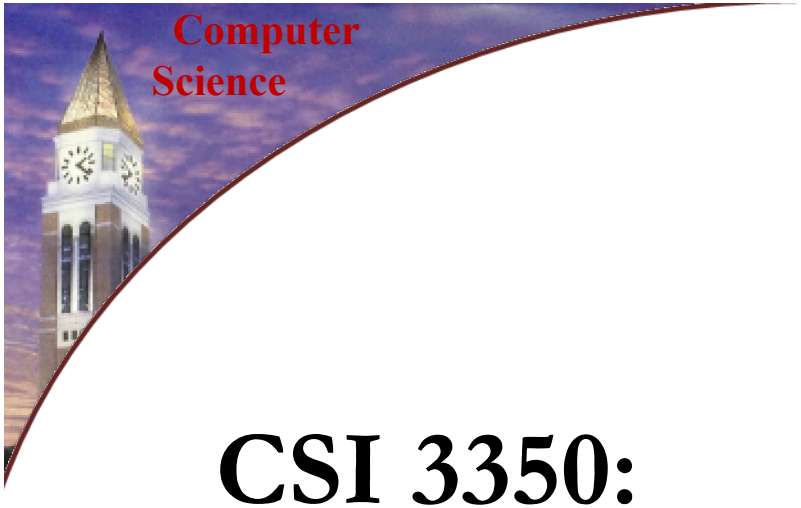


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University

Anonymous Functions

- $F(x) = (x + 1)$
- $g(x) = (x + 1)$

Lambda Expression

- $F(x) = (x + 1)$

function name

input parameter

output

anonymous function

```
(lambda (x) (+ x 1))
```

Anonymous Functions

- $F(x) = (x + 1)$
- $g(x) = (x + 1)$

A Function of Two Forms of Definition

- $F(x) = (x + 1)$

```
(define ( F x )
     (+ x 1)
)
```

```
(define F
     ( lambda (x) (+ x 1) )
)
```

- $G(x) = (x + 1)$

```
(define ( G x )
     (+ x 1)
)
```

```
(define G F)
```

Lambda Expression

- $F(x) = (x + 1)$

function name

input parameter

output

anonymous function

```
(lambda (x) (+ x 1))
```

A step-wise illustration of the higher order `foldl` function

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))
```

A step-wise illustration of the higher order `foldl` function

`(foldl - 0 '(1 2 3 4))` ; the initial default is 0, as highlighted.

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))  ; the initial default is 0, as highlighted.
        |
        |  round 1:  (- 1 0 ) = 1, so at the end of round 1, default is internally
        |  updated to 1.  next round starts with the 2nd element in the list, i.e., 2
```

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))  ; the initial default is 0, as highlighted.
        |
        |   round 1:  (- 1 0 ) = 1, so at the end of round 1, default is internally
        |   updated to 1.  next round starts with the 2nd element in the list, i.e., 2
        |   , which leads to the following:
    (foldl - 1 '(2 3 4))
```

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))  ; the initial default is 0, as highlighted.
        |
        |   round 1:  (- 1 0 ) = 1, so at the end of round 1, default is internally
        |   updated to 1.  next round starts with the 2nd element in the list, i.e., 2
        |   , which leads to the following:
    (foldl - 1 '(2 3 4))
        |
        |   round 2:  (- 2 1 ) = 1, so at the end of round 2, default is internally
        |   updated to 1.  next round starts with the 3rd element in the list, i.e., 3
```

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))  ; the initial default is 0, as highlighted.
        |
        |   round 1:  (- 1 0 ) = 1, so at the end of round 1, default is internally
        |   updated to 1.  next round starts with the 2nd element in the list, i.e., 2
        |   , which leads to the following:
    (foldl - 1 '(2 3 4))
        |
        |   round 2:  (- 2 1 ) = 1, so at the end of round 2, default is internally
        |   updated to 1.  next round starts with the 3rd element in the list, i.e., 3
        |   , which leads to the following:
    (foldl - 1 '(3 4))
```

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))  ; the initial default is 0, as highlighted.
```

| round 1: (- 1 **0**) = **1**, so at the end of round 1, **default** is internally updated to 1. next round starts with the 2nd element in the list, i.e., 2 , which leads to the following:

```
(foldl - 1 '(2 3 4))
```

| **round 2**: (- 2 **1**) = **1**, so at the end of round 2, **default** is internally updated to **1**. next round starts with the 3rd element in the list, i.e., 3 , which leads to the following:

```
(foldl - 1 '(3 4))
```

| **round 3**: (- 3 **1**) = 2, so at the end of round 3, **default** is internally updated to **2**. next round starts with the last element in the list, i.e., 4

A step-wise illustration of the higher order `foldl` function

```
(foldl - 0 '(1 2 3 4))  ; the initial default is 0, as highlighted.
        |
        |  round 1:  (- 1 0 ) = 1, so at the end of round 1, default is internally
        |  updated to 1.  next round starts with the 2nd element in the list, i.e., 2
        |  , which leads to the following:
    (foldl - 1 '(2 3 4))
        |
        |  round 2:  (- 2 1 ) = 1, so at the end of round 2, default is internally
        |  updated to 1.  next round starts with the 3rd element in the list, i.e., 3
        |  , which leads to the following:
    (foldl - 1 '(3 4))
        |
        |  round 3:  (- 3 1 ) = 2, so at the end of round 3, default is internally
        |  updated to 2.  next round starts with the last element in the list, i.e., 4
        |  , which leads to the following:
    (foldl - 2 '(4))
        |  (see next slide)
```

A step-wise illustration of the higher order `foldl` function

```
(foldl - 2 '(4))
      |
      |
```

round 4: (- 4 **2**) = **2**, so at the end of round 4, default is internally updated to **2**.

A step-wise illustration of the higher order `foldl` function

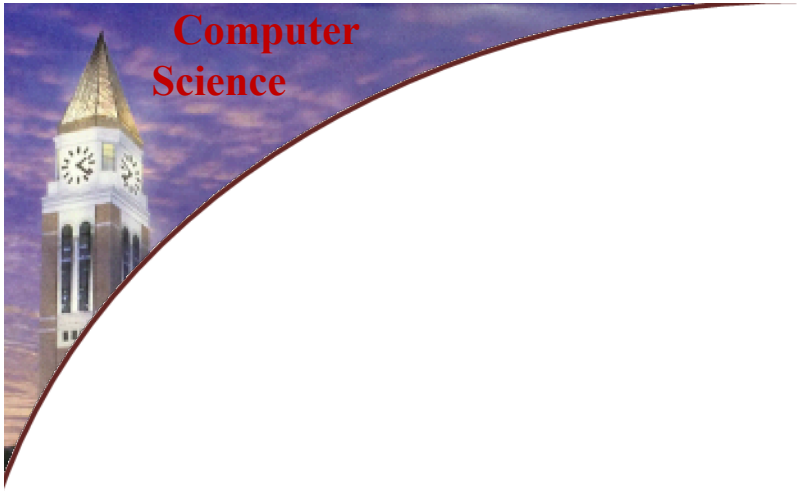
```
(foldl - 2 '(4))
```

|

| **round 4**: (- 4 **2**) = **2**, so at the end of round 4, default is internally updated to **2**. no more element in the list left, the default holds the final value, i.e.,**2**

A step-wise illustration of the higher order `foldl` function

```
(foldl - 2 '(4))
      |
      |   round 4:  (- 4 2 ) = 2, so at the end of round 4, default is internally
      |   updated to 2.  no more element in the list left, the default holds the
      |  final value, i.e.,2
     2
```




```
(foldl string-append "" '( "C" "S" "I" "3" "3" "5" "0") )
```

= ?

How to translate the following Java code to its equivalent Scheme code ?

How to translate the following Java code to its equivalent Scheme code ?

Java

```
int sum = 0;
for (int i = 0; i < n; i++)
  {
     sum += i;
  }
return sum;
```

How to translate the following Java code to its equivalent Scheme code ?

Java

```
int sum = 0;
for (int i = 0; i < n; i++)
  {
     sum += i;
  }
return sum;
```

Scheme

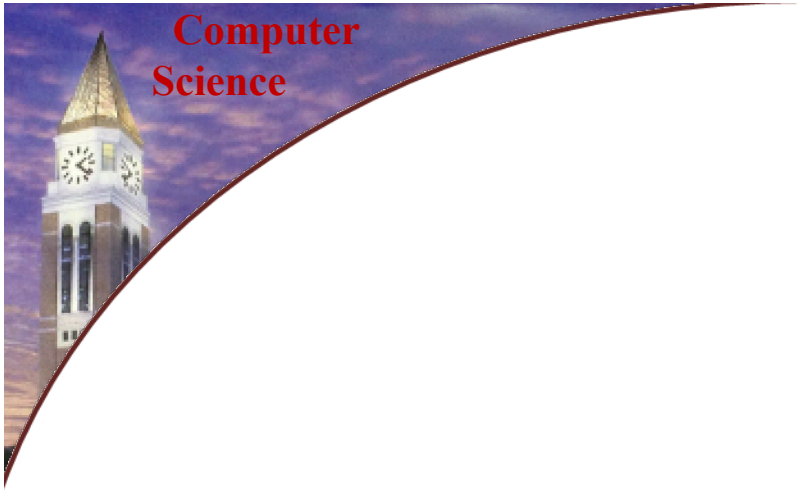
How to translate the following Java code to its equivalent Scheme code ?

Java

```
int sum = 0;
for (int i = 0; i < n; i++)
  {
    sum += i;
  }
return sum;
```

Scheme

```
(foldl + 0 (range n))
```



HW03

Already out

Local Variable Binding

Local Variable Binding using `let`

Local Variable Binding using `let`

```
(let
    ((a  2)
     (b  3))
    (+ a b)
 )
```

Local Variable Binding using `let`

```
(let
    ((a  2)
     (b  3))
    (+ a b)
 )
```

Local Variable Binding using `let`

```
(let
    ((a  2)
     (b  3))
    (+ a b)
 )
```

the value of any **`let`** expression is the last subexpression in it!

Local Variable Binding using `let`

binding for var **a**

```
(let
     ((a   2)
      (b   3))
     (+ a b)
 )
```

Local Variable Binding using `let`

```
(let
     ((a   2)
      (b   3))
     (+ a b)
 )
```

binding for var **a**

binding for var **b**

Local Variable Binding using `let`

```
(let
     ((a   2)
      (b   3))
     (+ a b)
 )
```

binding for var **a**

binding list

binding for var **b**

Local Variable Binding using `let`

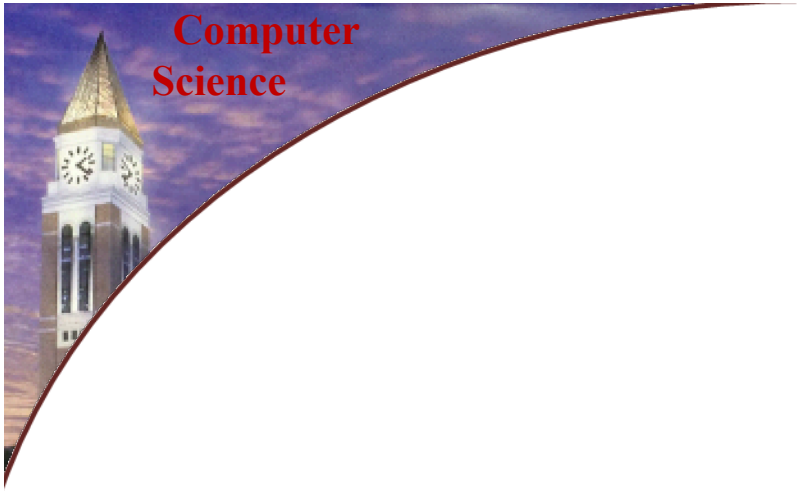
```
(let
     ((a   2)
      (b   3))
     (+ a b)
 )
```

binding for var **a**

binding list

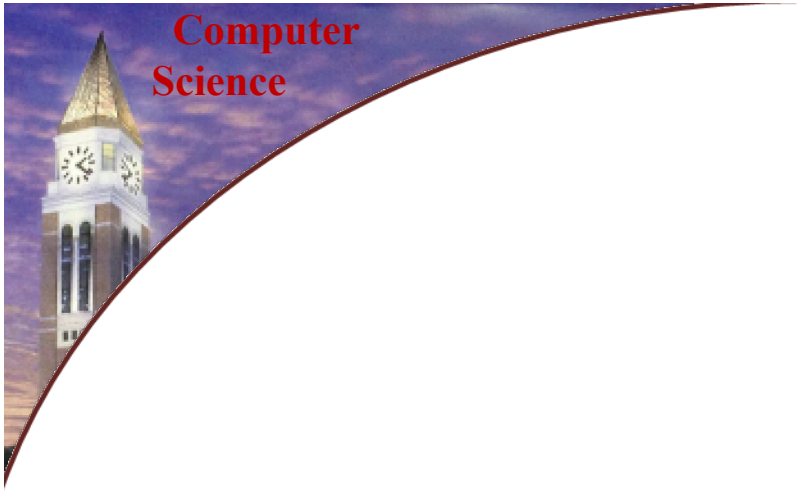
binding for var **b**

```
(let
     ((a   +)
      (b   3))
     (a 2 b)
 )
```




```
(define (f a)
    (+ a 3))

(f 4)
```




```
(define (f a)
    (+ a 3))

(f 4)
```

What is the **equivalent** `let` expression ?

```
(let
  (
    (a 4)
   )
(+ a 3))
```

```
(define (f a)
    (+ a 3))

(f 4)
```

What is the **equivalent** `let` expression ?

```
(let
  (
    (a 4)
   )
(+ a 3))
```

```
((lambda(a) (+ a 3)) 4)
```

What is the **equivalent** `let` expression ?

```
(let
  (
    (a 4)
  )
(+ a 3))
```

```
(define (f a)
    (+ a 3))

(f 4)
```

```
((lambda(a) (+ a 3)) 4)
```

What is the **equivalent** `let` expression ?

```
(let
  (
    (a 4)
    (b 5)
   )
(+ a b 3))
```

What is the **equivalent function definition and function call for this `let`** expression ?

```
(let
  (
    (a 4)
    (b 5)
   )
(+ a b 3))
```

```
(define (f a b)
    (+ a b 3))

(f 4 5)
```

What is the **equivalent function definition and function call for this `let`** expression ?

```
(let
  (
    (a 4)
    (b 5)
   )
(+ a b 3))
```

```
(define (f a b)
    (+ a b 3))

(f 4 5)
```

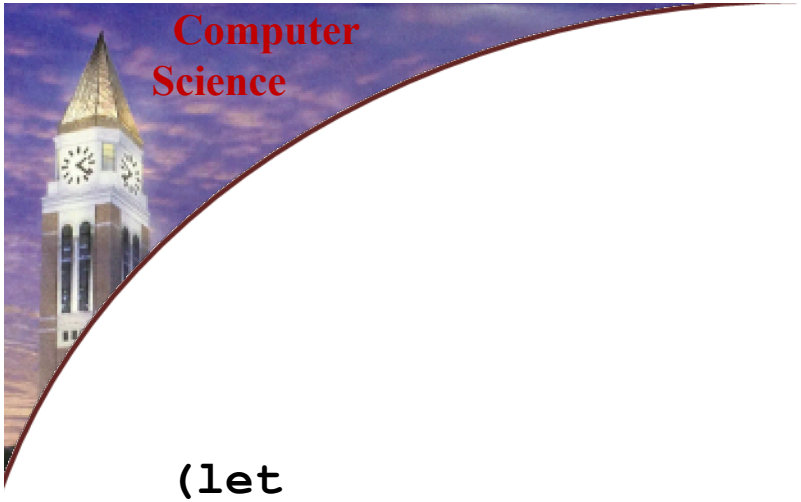
let is simply another way to define and call **function.**

```
(let
  (
    (a 4)
    (b 5)
   )
(+ a b 3))
```

```
(define (f a b)
    (+ a b 3))

(f 4 5)
```

let is called a **syntactic sugar** for function definition & function call.




```
(let
  (
    (a 4)
  )
(+ a 3))
```

```
(define (f a)
    (+ a 3))

(f 4)
```

```
((lambda(a) (+ a 3)) 4)
```

What is the **equivalent** `let` expression ?

```
(let (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

```
(let (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

```
(let (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

```
(let (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

```
(letrec (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

let vs letrec

```
(let (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

```
(letrec (
      (a 1)
      (b (+ a 2))
      )
  (+ a b))
```

Syntactic Sugar – Define New Syntax

- Define Your Own Syntax
- `define-syntax-rule`

```
(let
  (
    (a 4)
  )
(+ a 3))
```

```
(define (f a)
    (+ a 3))

(f 4)
```

```
((lambda(a) (+ a 3)) 4)
```

Scheme / Racket

What is the **equivalent** `let` expression ?

```
(define-syntax-rule (let-335 (var exp) body)
     ((lambda (var) body) exp)
)
```

function definition

function call

Syntax Abstraction

- What's the difference between
  - define:
    - (define a b)
  - define-syntax-rule & define-syntax
    - (define-syntax-rule a b)

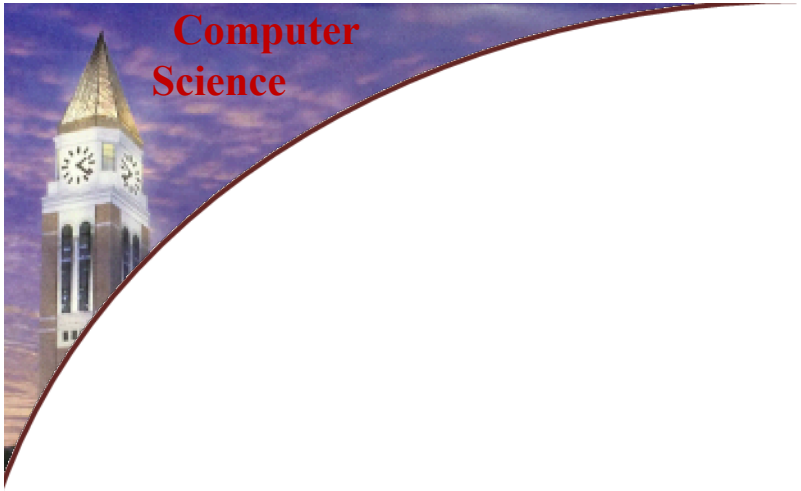
```
(define-syntax-rule (let-335 (var exp) body)
     ((lambda (var) body) exp)
)
```

a

b

```
(if-335 (> 3 1) then "right!" else "wrong!")
```

```
"right!"
```



Add `while` loop to the Scheme language!

Can You Implement `or`

```
(or335)   => #f

(or335 #t)   => #t

(or335 #f #t)   => #t

(or335 #f #f)   => #f
```

```
(define-syntax-rule (or335)
   (
       . . .
    )
)


(define-syntax-rule (or335 a)
   (
       . . .
    )
)


(define-syntax-rule (or335 a b)
   (
       . . .
    )
)
```

Relevant Syntax Pattern: . . .

- . . . repeats the immediate previous symbol zero or more times

define-syntax

```
(define-syntax id
  (syntax-rules ()
    [ pattern  def  ]
     ...
    [ pattern  def  ]
  )
)
```

list of correlated syntax rules

new syntax

legal Scheme code to evaluate the new syntax

Syntax Abstraction

```
(or335 #f  #f  #f)
```

```
(or335 #f  #f  #f)
```

```
(define-syntax or335
   (syntax-rules ()
     [(or335)  #f ]
     [(or335 a b ...) (if a #t (or335 b ...))]
   )
)
```

at least a space between **b** and . . .

match: **a** #f
b ... #f #f

substitution:

```
(if #f #t (or335 #f #f))
```

evaluate to:

```
(or335 #f #f)
```

Syntax Abstraction

```
(or335 #f  #f)

(or335 #f  #f)
```

```
(define-syntax or335
   (syntax-rules ()
     [(or335)  #f ]
     [(or335 a b ...) (if a #t (or335 b ...))]
   )
)
```

at least a space between **b** and . . .

match: **a** #f
b ... #f

substitution:

```
(if #f #t (or335 #f ))
```

evaluate to:

```
(or335 #f)
```

Syntax Abstraction

```
(or335 #f)
```

```
(or335 #f)
```

```
(define-syntax or335
   (syntax-rules ()
     [(or335)  #f ]
     [(or335 a b ...) (if a #t (or335 b ...))]
   )
)
```

at least a space between **b** and . . .

match: **a** #f
b … (empty)

substitution:

```
(if #f #t (or335 ))
```

evaluate to:

```
(or335 )
```

Syntax Abstraction

```
(or335)

(or335)
```

```
(define-syntax or335
   (syntax-rules ()
     [(or335)  #f ]
     [(or335 a b ...) (if a #t (or335 b ...))]
   )
)
```

at least a space between `b` and . . .

match: **the first syntax rule**

substitution:

nothing (ready to evaluate!)

evaluate to:

#f