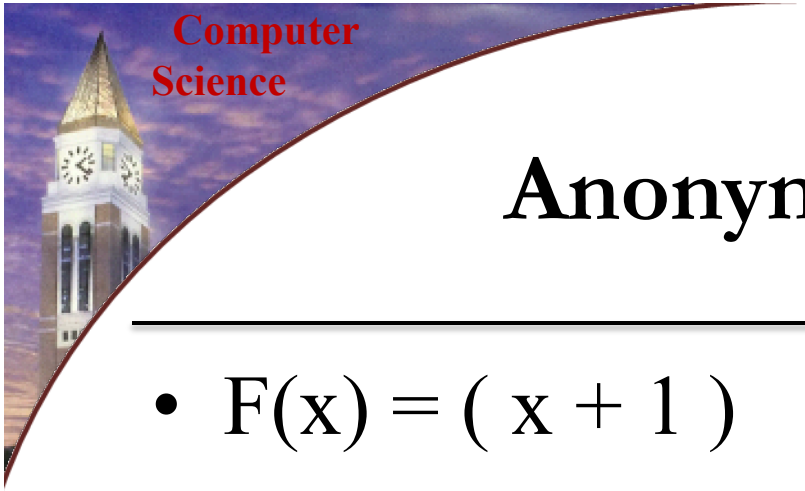
CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University



Anonymous Functions

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

Lambda Expression

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

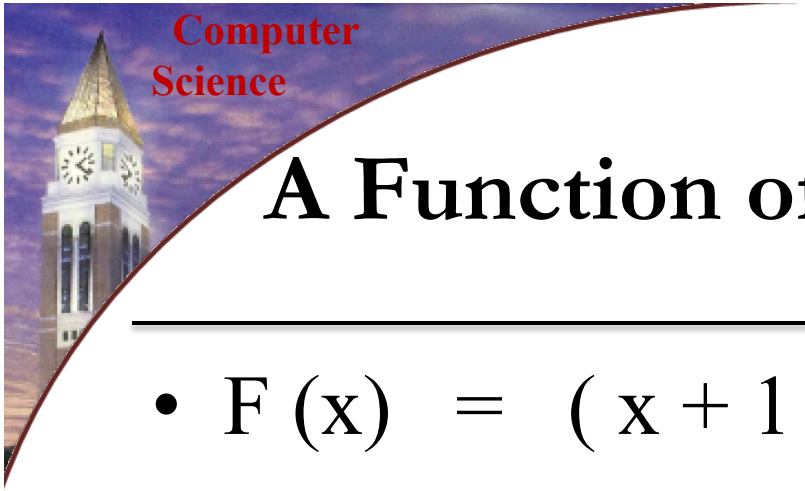
function name

input parameter

output

anonymous function

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



A Function of Two Forms of Definition

- F (x) = (x + 1)

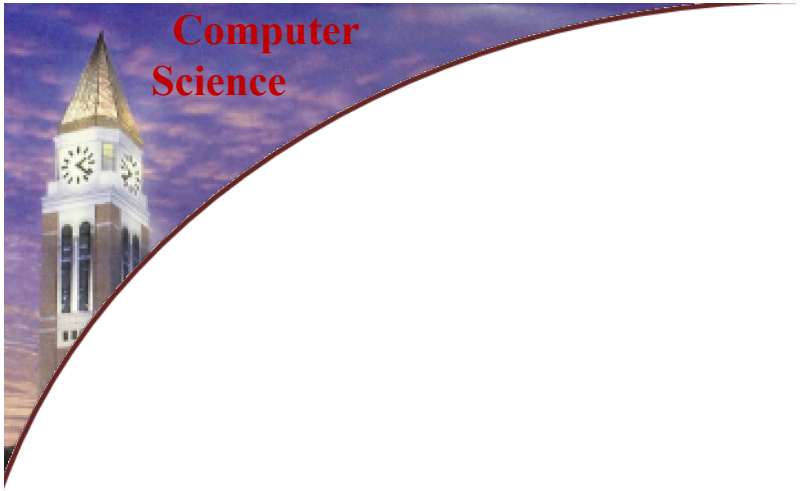
```
(define ( F x )
     (+ x 1)
)
```

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

- G (x) = (x + 1)

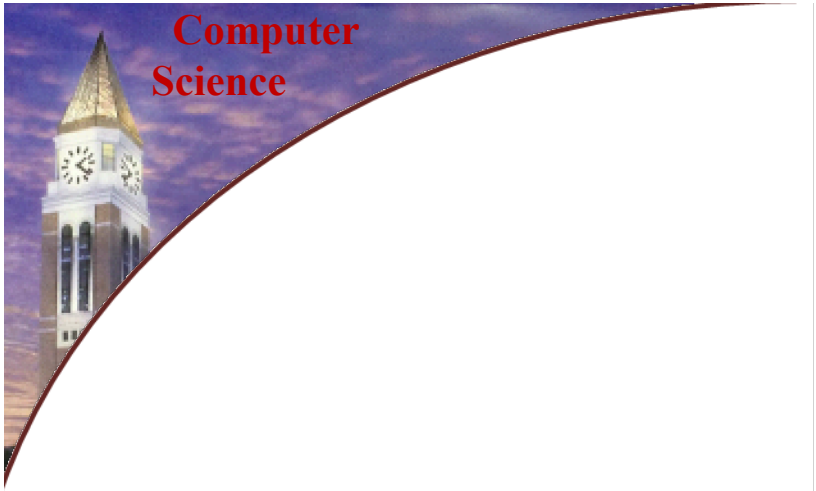
```
(define ( G x )
     (+ x 1)
)
```

```
(define G F)
```



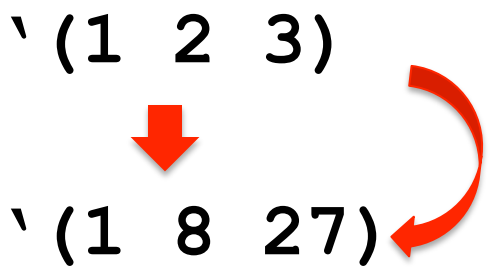
HW03

Out tonight



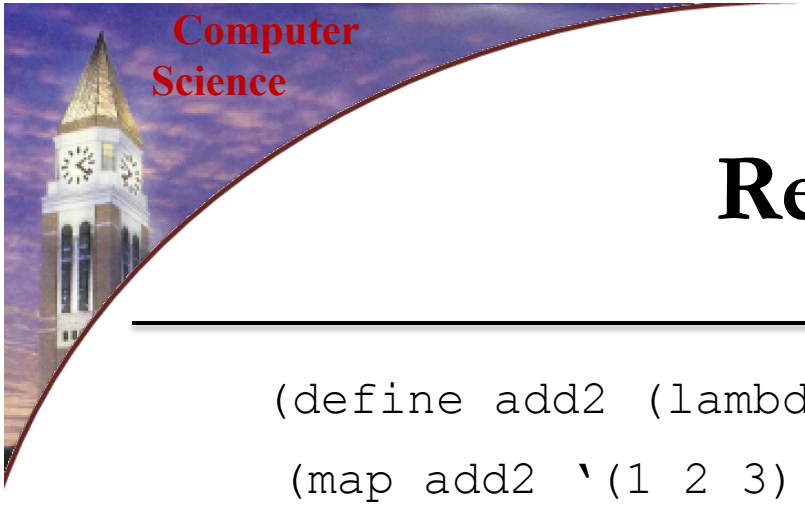
HW 02
(p5 - parenthesis balancing)

Map Function



cubic each number in a list

```
(map (lambda (x) (* x x x)) '(1 2 3) )
```



Revisit map

```
(define add2 (lambda (x) (+ x 2) )  )
(map add2 '(1 2 3) )
(map add2 '(1 2 3) )  =>  '(3 4 5)

(map + '(1 2 3) )  =>  ?

(map + '(1 2 3) '( 1 2 3) )  =>  ?

(map list '(1 2 3) '( 1 2 3) )  =>  ?
```

zip function

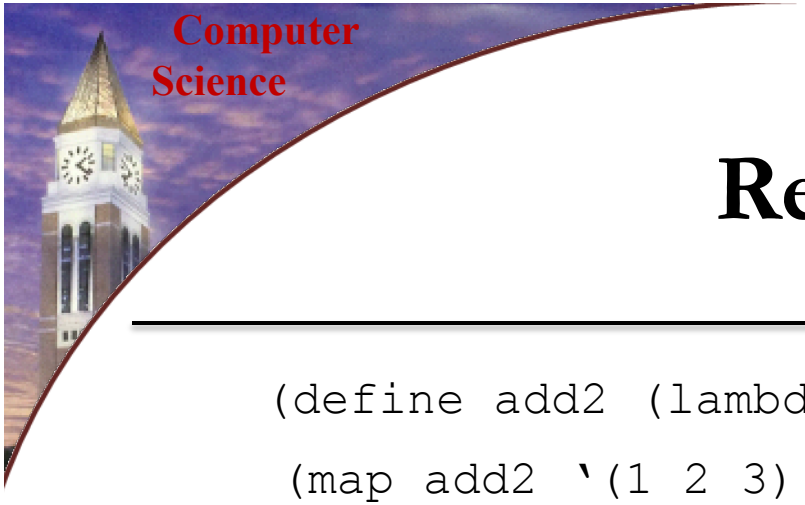
```
(zip '(3 4 2) '(5 9 7) )    ==> '((3 5) (4 9) (2 7))

(zip '(4 2) '(9 7) )        ==>  '((4 9) (2 7))

(zip '(2 3 1) '(9 2) )      ==>  '( (2 9) (3 2) )

(zip '() '(3 1 4 1 5 9) )   ==>  '()
```

Sep 18 lecture slide 48

zip function

```
(define (zip lst1 lst2)
  (if (or (null? lst1)
          (null? lst2))
      '()
      (cons (list (car lst1) (car lst2))
            (zip (cdr lst1) (cdr lst2)))
      )
  )
```

Sep 18 lecture slide 49



Revisit map

```
(define add2 (lambda (x) (+ x 2) )  )
(map add2 '(1 2 3) )
(map add2 '(1 2 3) )  =>  '(3 4 5)

(map + '(1 2 3) )  =>  ?

(map + '(1 2 3) '( 1 2 3) )  =>  ?

(map list '(1 2 3) '( 1 2 3) )  =>  ?
```

compare this with the `zip` function we coded (Sep 18 lecture notes).

Carpet

(carpet 4) =
step-1: for-each list in `(carpet 3)` expand it by adding `'%` to the beginning and end of it

```
(carpet 3)
'(  (+ + + + + + +)
    (+ % % % % % +)
    (+ % + + + % +)
    (+ % + % + % +)
    (+ % + + + % +)
    (+ % % % % % +)
    (+ + + + + + +))
```

```
'(  (% + + + + + + + %)
    (% + % % % % % + %)
    (% + % + + + % + %)
    (% + % + % + % + %)
    (% + % + + + % + %)
    (% + % % % % % + %)
    (% + + + + + + + %))
```

```
(carpet 4)
'(  (% % % % % % % % %)
    (% + + + + + + + %)
    (% + % % % % % + %)
    (% + % + + + % + %)
    (% + % + % + % + %)
    (% + % + + + % + %)
    (% + % % % % % + %)
    (% + + + + + + + %)
    (% % % % % % % % %))
```

step-2: add `'(%%%%%%%%%)` to the beginning and the end of the result returned by step-1

Carpet

(carpet 3)

```
`(  (+ + + + + + +)
    (+ % % % % % +)
    (+ % + + + % +)
    (+ % + % + % +)
    (+ % + + + % +)
    (+ % % % % % +)
    (+ + + + + + +)  )
```

(carpet 4)

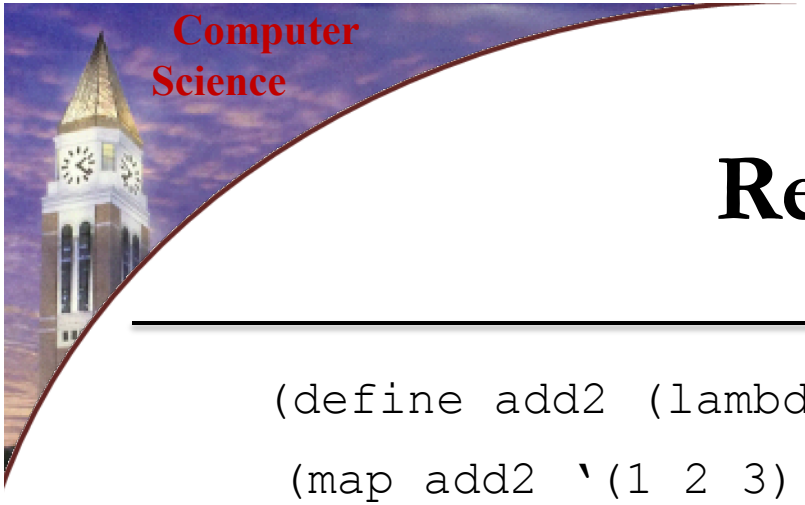
```
`(  (% % % % % % % % %)
    (% + + + + + + + %)
    (% + % % % % % + %)
    (% + % + + + % + %)
    (% + % + % + % + %)
    (% + % + + + % + %)
    (% + % % % % % + %)
    (% + + + + + + + %)
    (% % % % % % % % %))
```

(carpet 4) =
step-1: for-each list in `(carpet 3)` expand it by adding `'%` to the beginning and end of it

```
`(  (% + + + + + + + %)
    (% + % % % % % + %)
    (% + % + + + % + %)
    (% + % + % + % + %)
    (% + % + + + % + %)
    (% + % % % % % + %)
    (% + + + + + + + %))
```

can you use **map** to generate the top and bottom ?

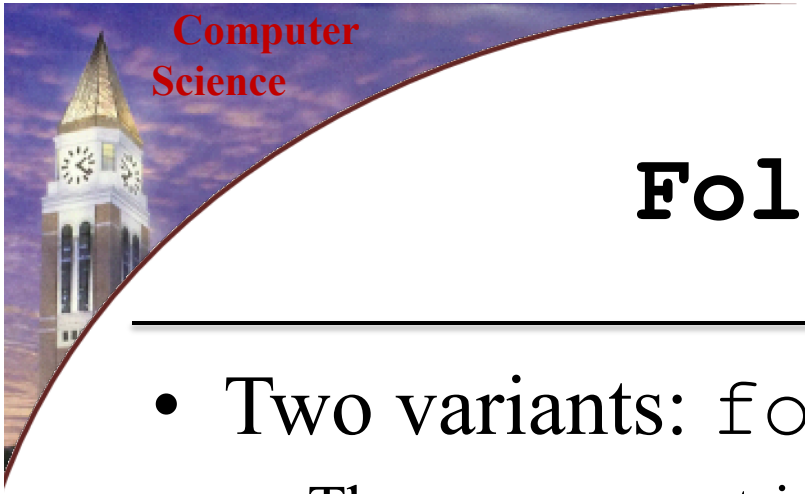
step-2: add `'(%%%%%%%%%)` to the beginning and the end of the result returned by step-1



Revisit map

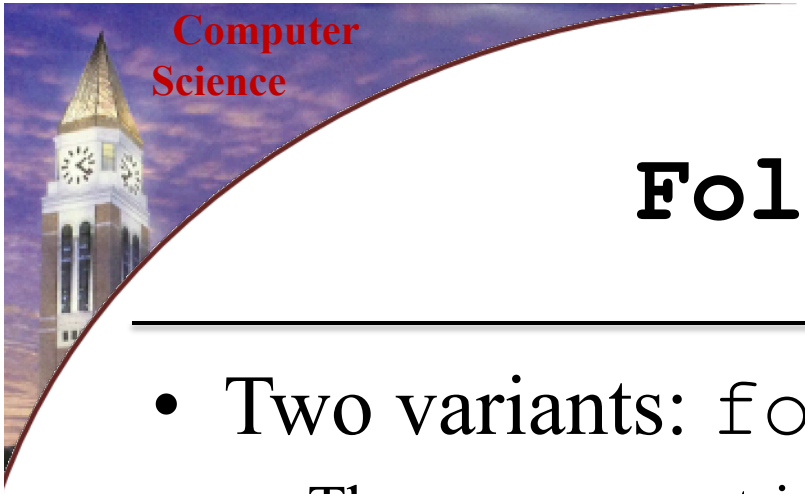
```
(define add2 (lambda (x) (+ x 2) )  )
(map add2 '(1 2 3) )
(map add2 '(1 2 3) )  =>  '(3 4 5)

(map + '(1 2 3) )  =>  ?

(map + '(1 2 3) '( 1 2 3) )  =>  ?

(map list '(1 2 3) '( 1 2 3) )  =>  ?
```

compare this with the `zip` function we coded (Sep 18 lecture notes).



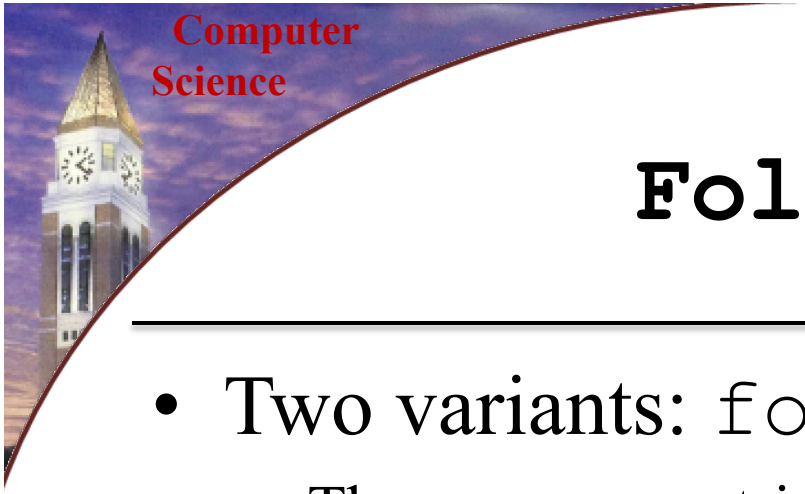
`Fold` Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric



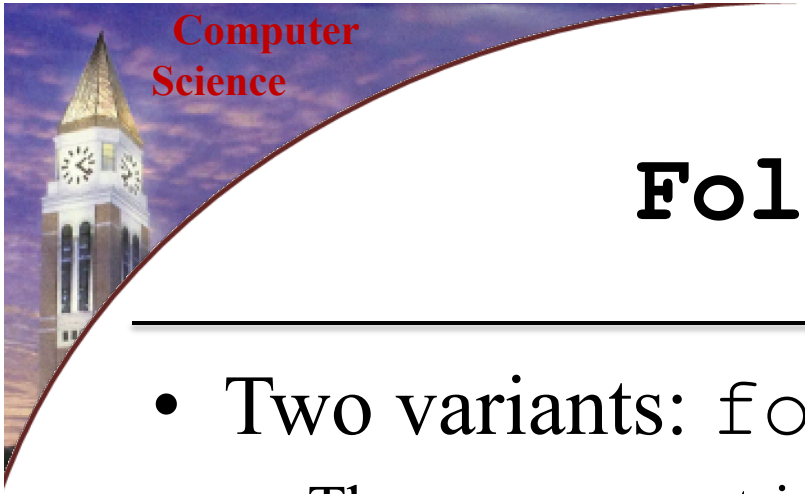
`Fold` Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric



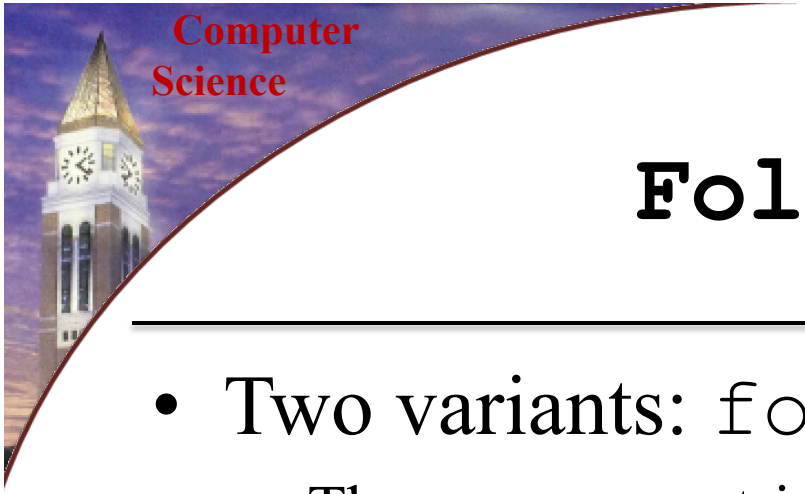
`Fold` Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric, understanding one of them can help quickly understand the other



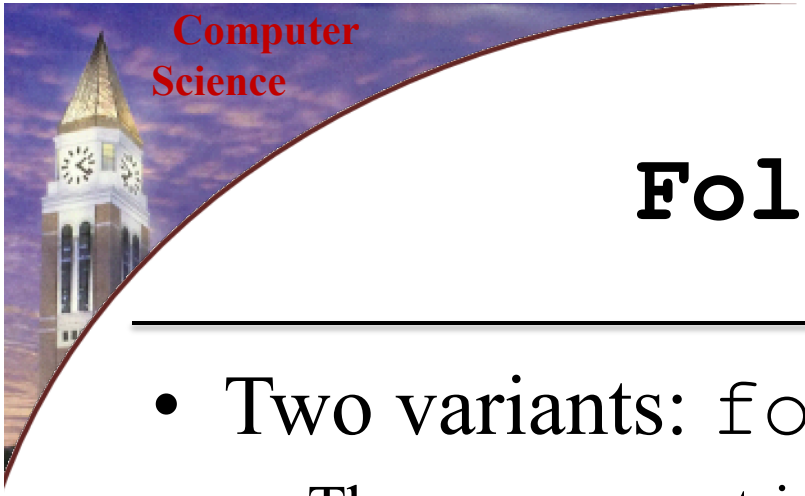
`Fold` Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric, understanding one of them can help quickly understand the other
  - Let us first look at `foldl`



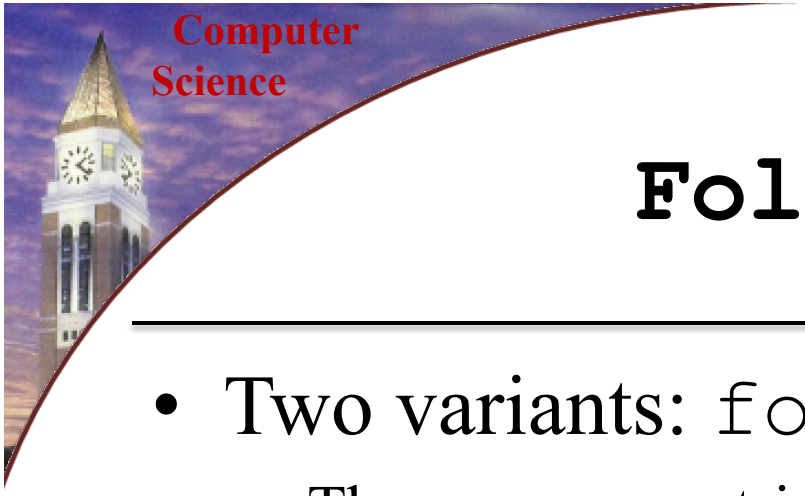
`Fold` Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric, understanding one of them can help quickly understand the other
  - Let us first look at `foldl`
    - `(foldl proc default list`$_1$ `... list`$_n$`)`



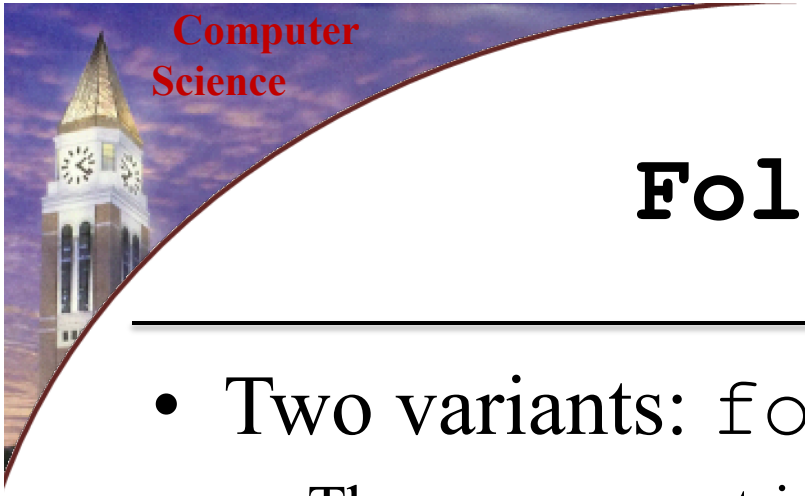
Fold Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric, understanding one of them can help quickly understand the other
  - Let us first look at `foldl`
    - `(foldl proc default list`$_1$ `... list`$_n$`)`
    - the number of parameters that `proc` have must be **n+1**



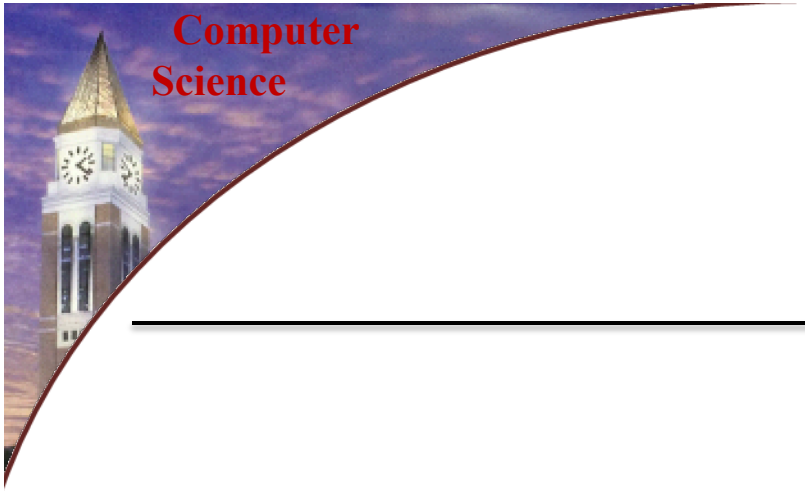
Fold Functions

- Two variants: `foldl` & `foldr`
  - They are symmetric, understanding one of them can help quickly understand the other
  - Let us first look at **`foldl`**
    - `(foldl proc default list`$_1$ `... list`$_n$`)`
    - the number of parameters that **`proc`** have must be **n+1**
    - the computation goes left to right



Fold Functions

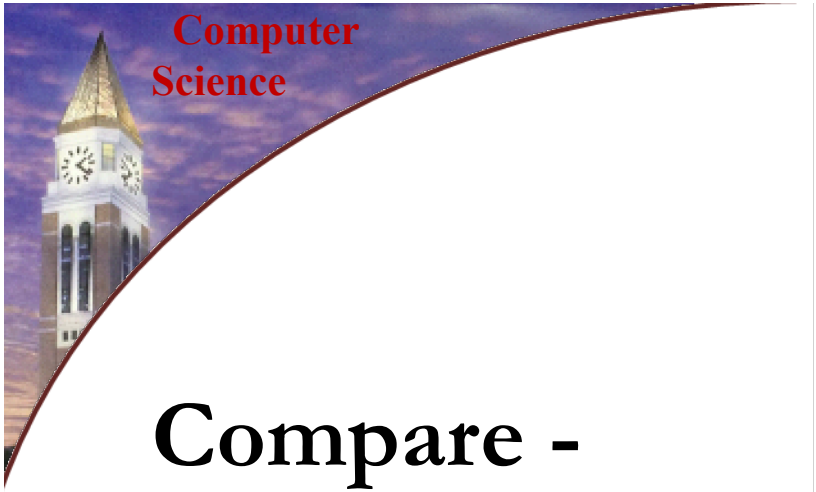
- Two variants: `foldl` & `foldr`
  - They are symmetric, understanding one of them can help quickly understand the other
  - Let us first look at **`foldl`**
    - `(foldl proc default list`$_1$ `...  list`$_n$`)`
    - the number of parameters that **`proc`** have must be **n+1**
    - the computation goes left to right
    - the intermediate result of `(proc x`$_1$ `...  x`$_n$ `default)` is stored and was used as the `default` for the next round



foldl

```
(foldl string-append "" '("1") '("2"))
```

```
(foldl string-append "" '("C" "E" "4") '("S" "3" "3"))
```

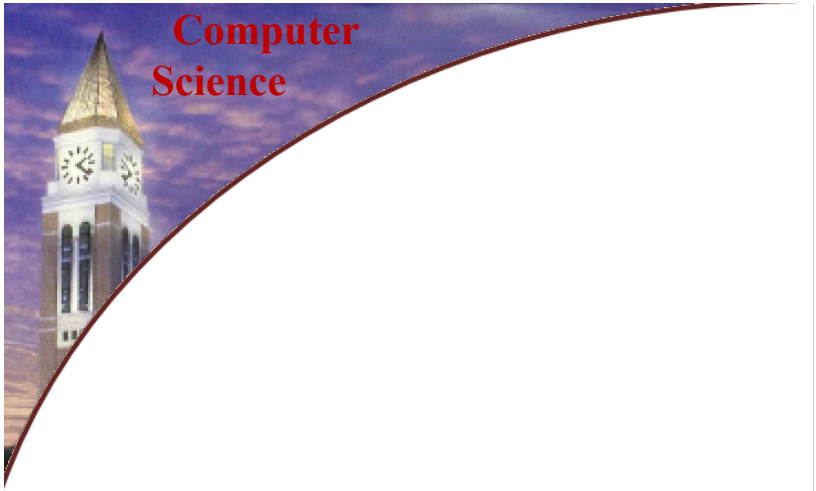


How to make your code more reusable and flexible?

Compare -

(define (sum-square-between a b) ...)

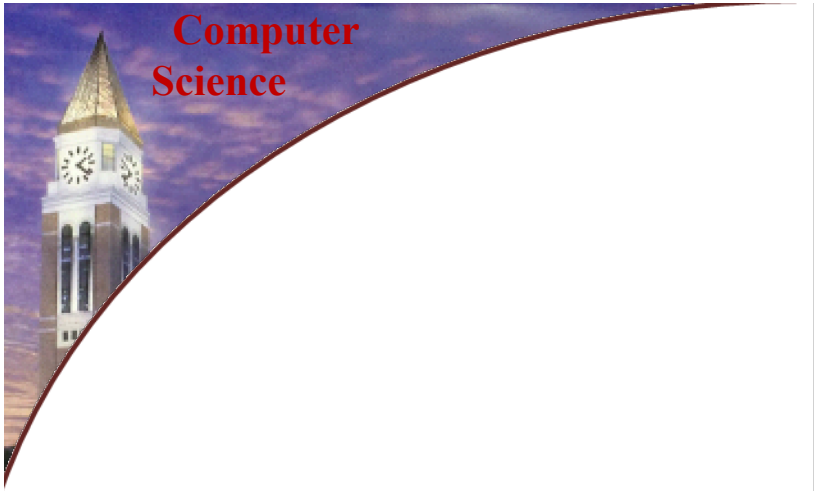
(define (sum-cubes-between a b) ...)



How to make your code more reusable and flexible?

(define (sum-cubes-between a b) ...)

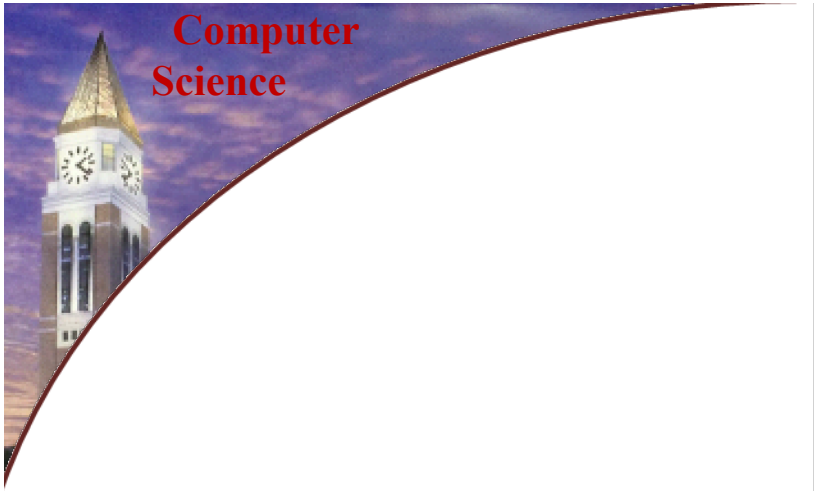
(sum-cubes-between 1 3) → $1^3+2^3+3^3$



How to make your code more reusable and flexible?

(define (sum-squares-between a b) ...)

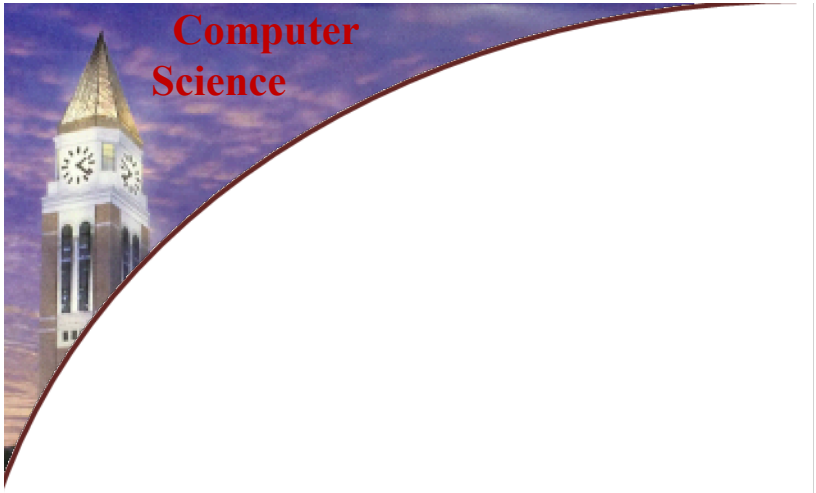
(sum-squares-between 1 3) → $1^2+2^2+3^2$



How to make your code more reusable and flexible?

(define (sum-between op a b) ...)

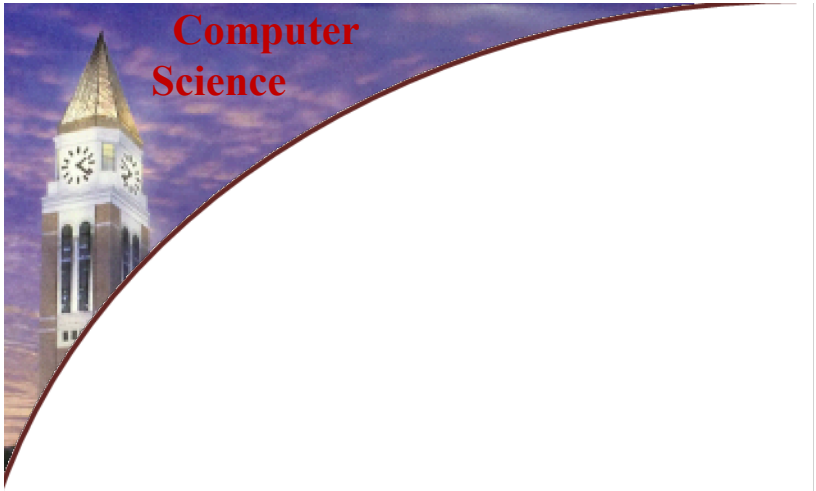
?



How to make your code more reusable and flexible?

(define (multiply-cubes-between a b) ...)

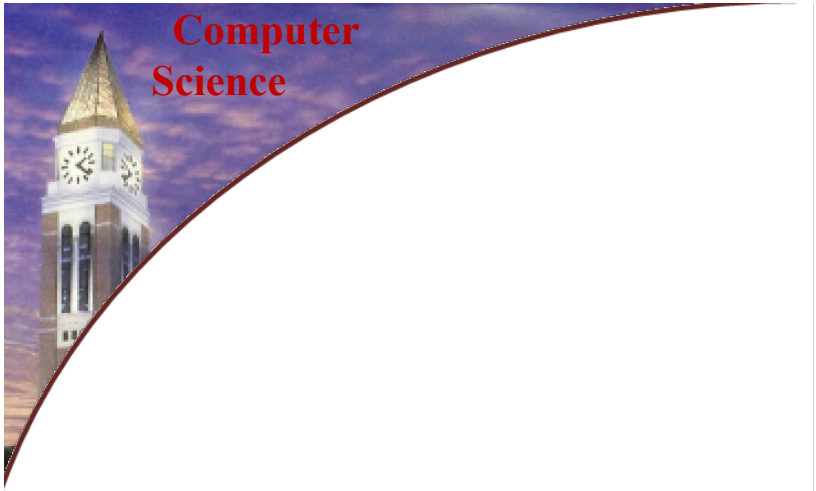
(multiply-cubes-between 1 3) → $1^3*2^3*3^3$



How to make your code more reusable and flexible?

(define (multiply-squares-between a b) ...)

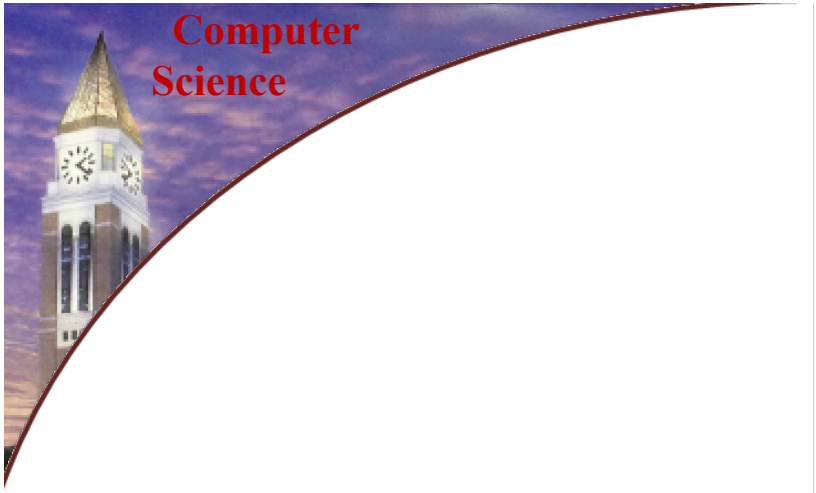
(multiply-squares-between 1 3) → $1^2*2^2*3^2$



How to make your code more reusable and flexible?

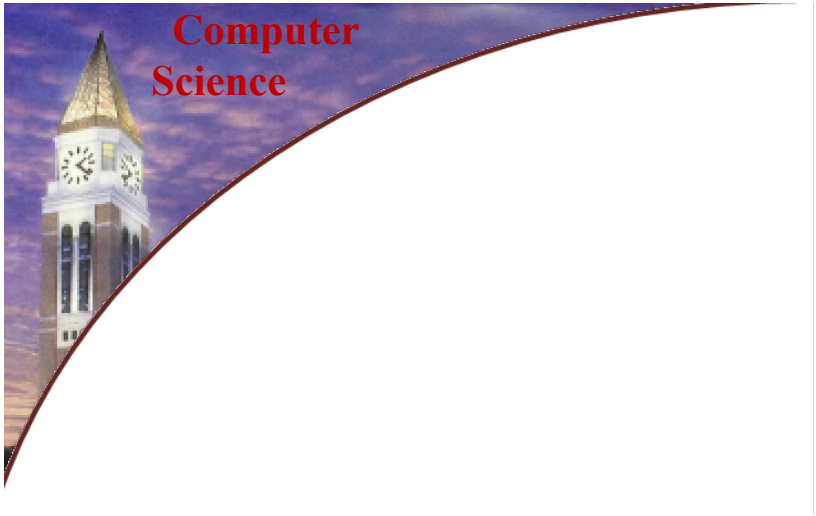
(define (sum-between op a b) ...)

?



How to make your code more reusable and flexible?

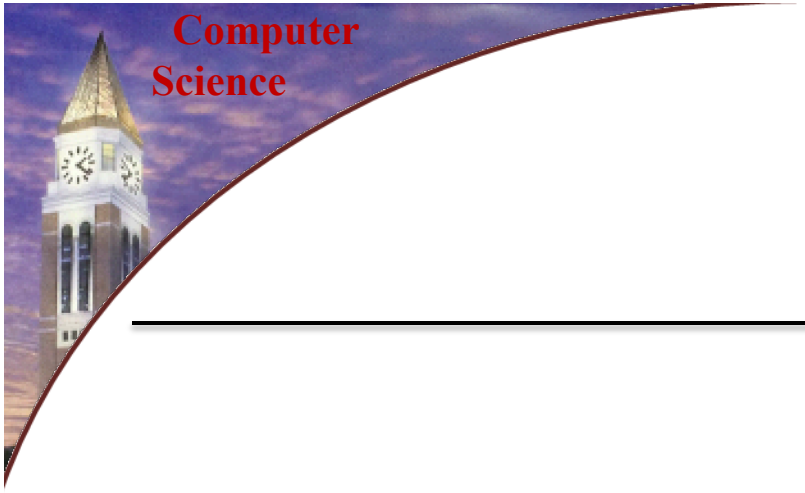
```
(define (op-op-between ….)
        ; your definition here
)
```



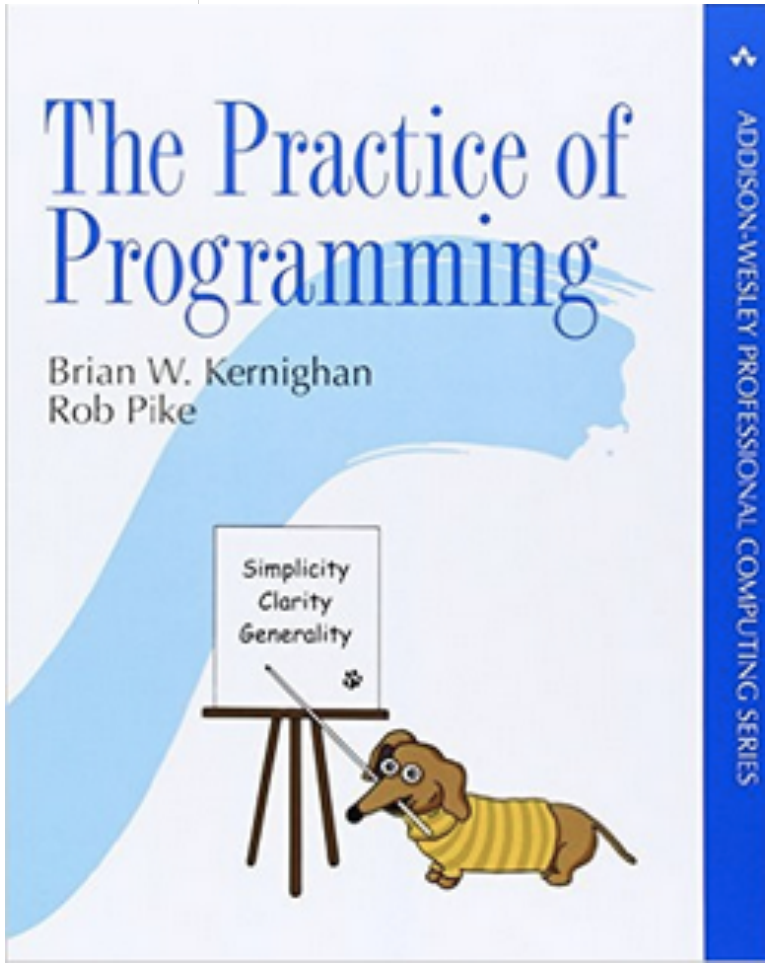

```
(define (op-op-between op1 op2 lower-bound higher-bound default)
  (foldl op1 default (map op2 (range lower-bound (+ higher-bound 1)))))
```

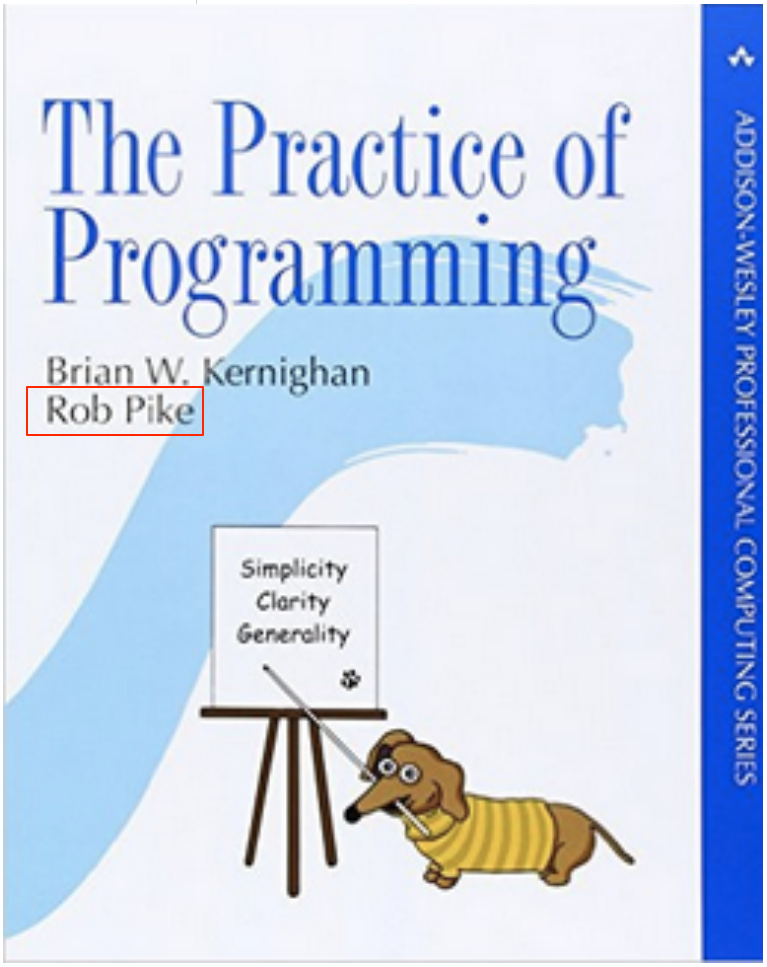
gives you `(Sum-square-between 1 3)`

```
> (op-op-between + (lambda (x) (* x x)) 1 3 0)
14
```



A Different Way of Thinking About Programming & Computing


The Practice of Programming
Brian W. Kernighan
Rob Pike
Simplicity
Clarity
Generality
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


The Practice of Programming
Brian W. Kernighan
Rob Pike
Simplicity
Clarity
Generality
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES