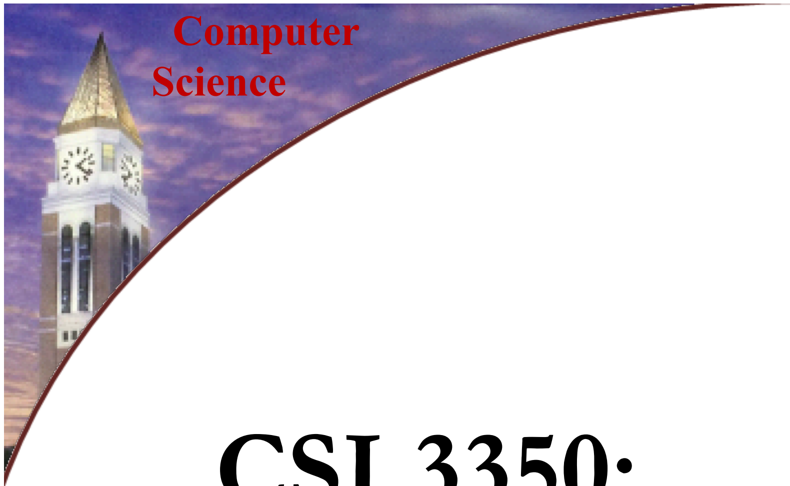


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University

Anonymous Functions

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

Computer Science

Lambda Expression

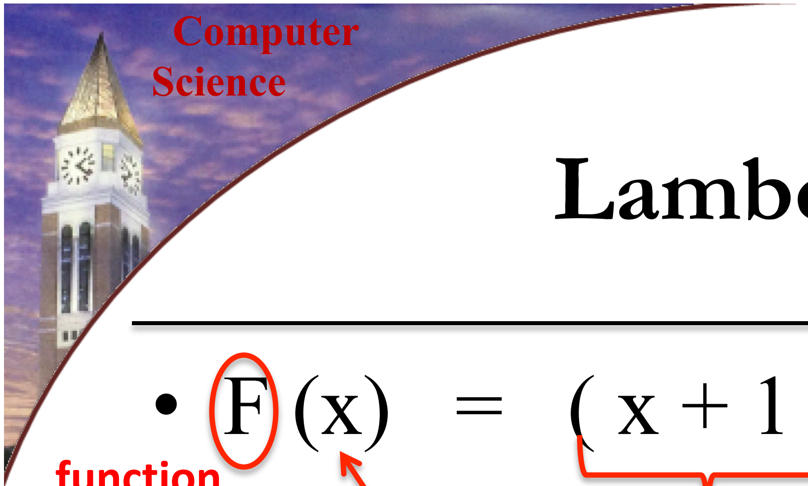
- $F(x) = \underbrace{(x + 1)}_{\text{function body}}$

function name

binding variable

$F\underbrace{(x) = (x + 1)}_{\text{function definition}}$

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



Lambda Expression

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

function name

input parameter

output

function definition

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

Computer Science

Lambda Expression

- $$\underbrace{F}_{\text{function name}}\ \underbrace{\underbrace{(x)}_{\text{input}} \;=\; \underbrace{(\,x+1\,)}_{\text{output}}}_{\text{function definition}}$$

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

Lambda Expression

- $$\underbrace{F}_{\text{function name}}\underbrace{(\underbrace{x}_{\text{input parameter}}) \;=\; \underbrace{(\,x + 1\,)}_{\text{output}}}_{\text{function definition}}$$

```
(lambda (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)
```

In general, proving any two functions are the same (equivalent) or not is undecidable !

```
(define
     (fact-new n )
     (fact-tail n 1)
)
```

```
(define
     (fact-tail n prod)
       (if
           (= n 0)
           prod
           (fact-tail (- n 1) (* n prod) )
       )
)
```

tail recursion!

```
(define
     (fact n )
       (if
           (= n 0)
           1
           (* n (fact (- n 1))))
)
```

```scheme
(define
    (fact-new n )
    (fact-tail n 1)
)
```

```scheme
(define
    (fact-tail n prod)
      (if
         (= n 0)
         prod
         (fact-tail (- n 1) (* n prod) )
      )
)
```

```scheme
(define
    (fact n )
      (if
         (= n 0)
         1
         (* n (fact (- n 1))))
)
```

Computer Science

```
(define
    (fact-new n )
    (fact-tail n 1)
)
```

`(fact-new n)` and `(fact n)` are the same, but proving any two functions are equivalent or not is undecidable !

```
(define
    (fact-tail n prod)
      (if
         (= n 0)
         prod
         (fact-tail (- n 1) (* n prod) )
      )
)
```

```
(define
    (fact n )
      (if
         (= n 0)
         1
         (* n (fact (- n 1))))
)
```

To decide their equivalence between `(fact-new n)` and `(fact n)` is a lot more direct and involved than the simple example of `f(x) = x + 1`.

The **general answer** is that to decide in finite amount of time that any two functions are equivalent or not is **a task too hard** for a modern computer (Turing machine) to finish, so-called **undecidable**.

To decide their equivalence between `(fact-new n)` and `(fact n)` is a lot more direct and involved than the simple example of `f(x) = x + 1`.

The **general answer** is that to decide in finite amount of time that any two functions are equivalent or not is **a task too hard** for a modern computer (Turing machine) to finish, so-called **undecidable**.

as a result

In Scheme language, the language designer just let the comparison of any functions to return a negative result, even for the simplest case i.e.,

```
(equal? (lambda (x)  (+ x 1)) (lambda (x) (+ x 1))  =  #f
```

HW02

HW02

Read the test file first !

Carpet

(carpet 3)

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

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

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

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

(carpet 4)

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

Computer Science

Carpet

(carpet 3)

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

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

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

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

(carpet 4)

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

Computer Science

Pascal Triangle

(define (pascal n) ...)

```
n = 1     '( (1) )

n = 2     '( (1)
             (1 1) )          <-> '( (1) (1 1) )

n = 3     '( (1)
             (1 1)            <->  '( (1) (1 1) (1 2 1) )
             (1 2 1)  )
```

(pascal 4) = inserting '(1 3 3 1) to the end of (pascal 3)

n = 3

```
n = 4     '( (1)
             (1 1)
             (1 2 1)
             (1 3 3 1) )

      <->  '( (1) (1 1) (1 2 1) (1 3 3 1) )
```

Pascal Triangle

(define (pascal `n`) ...)

```
n = 3   '( (1)                    n = 4   '( (1)
           (1 1)                             (1 1)
           (1 2 1)  )                        (1 2 1)
                                             (1 3 3 1) )
```

get the last element from (pascal 3)

↓

generate `'(1 3 3 1)`

↓

insert it into `(pascal 3)`

☺

(pascal 4) = inserting '(1 3 3 1) to the end of (pascal 3)

Pascal Triangle

- Generate `'(1 3 3 1)` from `'(1 2 1)`

```
       +    +
'(  1 → 2 → 1  )
        ↓   ↓
        3   3
```

Pascal Triangle

- Generate `'(1 3 3 1)` from `'(1 2 1)`

```
    +   +
'( 1 → 2 → 1 )
       ↓   ↓
   '( 3   3)
```

We first generate the core sub-list `'(3 3)`

Pascal Triangle

- Generate `'(1 3 3 1)` from `'(1 2 1)`

```
      +   +
'(  1   2   1  )
'(  1   3   3   1  )
```

At the very end, we just insert these two `1`'s into the beginning and end of `'(3 3)`

Pascal Triangle

(define (pascal n) ...)

```
n = 3    '(  (1)
             (1 1)
             (1 2 1)   )
```

```
n = 4    '(  (1)
             (1 1)
             (1 2 1)
             (1 3 3 1) )
```

get the last element from (pascal 3)

↓

generate '(1 3 3 1)

↓

insert it into (pascal 3)

☺

(pascal 4) = inserting '(1 3 3 1) to the end of (pascal 3)

Pascal Triangle

- Generate `'(1 3 3 1)` from `'(1 2 1)`

```
       +   +
'(  1  2  1  )
'(  1  3  3  1  )
```

At the very end, we just insert these two `1`'s into the beginning and end of `'(3 3)`

How to code it ?

Computer Science

Create-mapping

```
'(I II III IV V  )   key-list

'(1  2  3  4  5  )   value-list
```

Search-key: `IV`

Create-mapping

```
'(I  II  III  IV  V   )    key-list

'(1   2   3   4   5   )    value-list
```

Search-key: `IV`

Create-mapping

```
'(II III IV V  )   key-list
'(2  3  4  5  )   value-list
```

Search-key: `IV`

Computer Science

Create-mapping

```
'(II III IV V  )   key-list
'(2  3   4  5  )   value-list
```

Search-key: `IV`

Create-mapping

```
'(III IV V  )    key-list
'(3   4   5  )    value-list
```

Search-key: `IV`

Create-mapping

```
'(III IV V  )   key-list
 '(3  4  5  )   value-list
```

Search-key: `IV`

Create-mapping

```
'(IV V  )    key-list
'(4  5  )    value-list
```

Search-key: `IV`

Create-mapping

```
'(IV V  )   key-list
'(4  5  )   value-list
```

Search-key: `IV`

Computer Science

Map Function

```
'(1 2 3)
'(1 8 27)
```

cubic each number in a list

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