Functional Programming

(Conditions and Function)

Professor: Suman Saha

Booleans

- Boolean values
  - #t, #f for true and false
- Predicates: funs that evaluate to true or false
  - convention: names of Scheme predicates end in "?"
  - number?: test whether argument is a number
  - equal?
    - ex: (equal? 2 2), (equal? x (* 2 y)), (equal? #t #t)
  - =, >, <, <=, >=
    - = is only for numbers
    - (= #t #t) won't work
  - and, or, not
    - (and (> 7 5) (< 10 20))

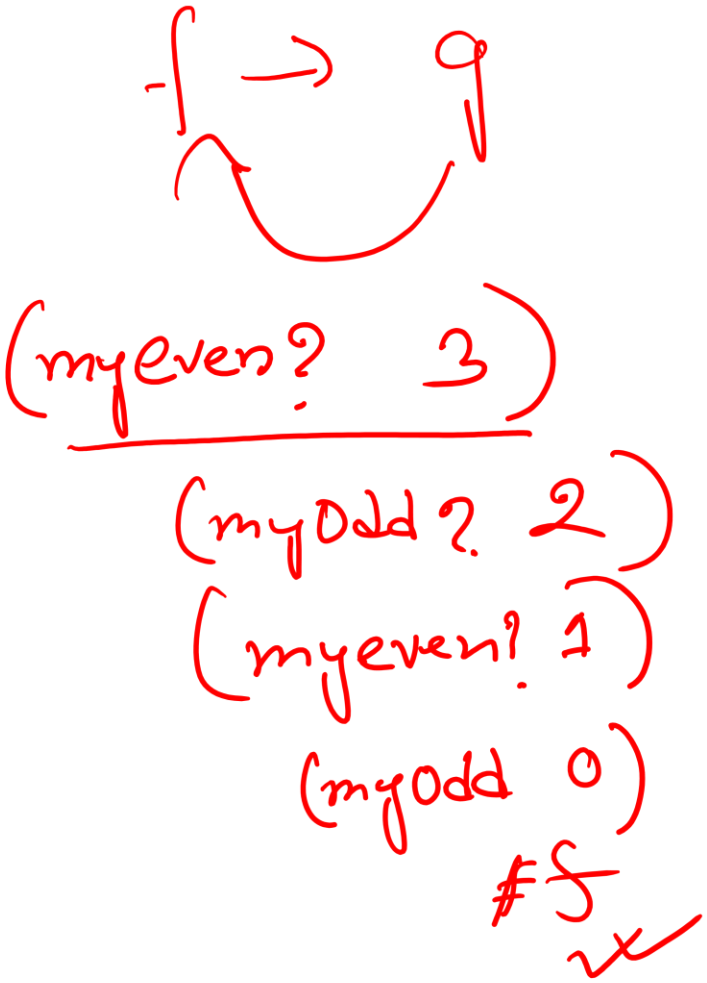
If expressions

- If expressions
  - (if P E1 E2)
    - eval P to a boolean, if it's true then eval E1, else eval E2
  - examples: max
    - (define (max x y) (if (> x y) x y))
  - It does not evaluate both branches
    - (define (f x) (if (> x 0) 0 (diverge x))
    - what is (f 1)? what is (f -1)

Mutual Rec. Functions

- even = true, if n =0
  odd(n-1), otherwise
- odd = false, if n =0
  even(n-1), otherwise

```
(define myeven?
  (lambda (n)
    (if (= n 0) #t (myodd? (- n 1)))))
(define myodd?
  (lambda (n)
    (if (= n 0) #f (myeven? (- n 1)))))
```



Multi-Case Conditionals

- (cond (P_1 E_1)
  …
  (P_n E_n)
  (else E_{n+1}))
  - "If P E_1 E_2" is a syntactic sugar
- examples
  - Problem: Write a function to assign a grade based on the value of a test score. an A for a score of 90 or above, a B for a score of 80-89, a C for a score of 70-79, a D for 60-69, a F otherwise.

```
(define (testscore x)
  (cond ((>= x 90) 'A)
        ((>= x 80) 'B)
        ((>= x 70) 'C)
        ((>= x 60) 'D)
        (else 'F)))
```

Higher-Order Functions

- Functions that
  - take functions as arguments
  - return functions as results
- Example:
  - $g(f,x) = f(f(x))$
  - if $f_1(x) = x + 1$,
    then $g(f_1,x) = f_1(f_1(x)) = f_1(x+1) = (x+1) + 1 = x + 2$
  - if $f_2(x) = x^2$,
    then $g(f_2,x) = f_2(f_2(x)) = f_2(x^2) = (x^2)^2 = x^4$

Higher-Order Functions in Scheme

PennState

- The ability to write higher-order functions
- Functions are first-class citizens in Scheme
- Examples:

```
(define (twice f x) (f  (f  x)))
(define (plusOne x) (+ 1 x))
(twice plusOne 2)
(twice square 2)
(twice (lambda (x) (+ x 2)) 3)
```

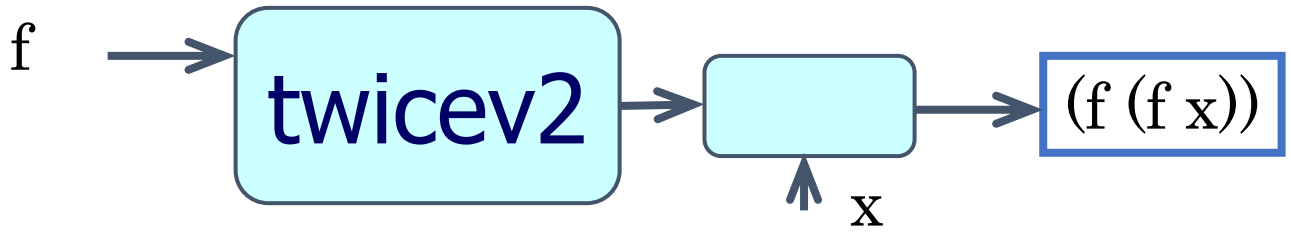
A Graphical Representation of Twice

- (define (twice f x) (f (f x)))
  - It takes a function f and an argument x, and returns the result of applying f to x twice

Q: Would Scheme accept (twice plusOne)?

Writing Twice in a Different Way

- (define (twiceV2 f)
  (lambda (x) (f (f x))))



- twiceV2 takes a function f as its argument, and returns a function, which takes x as its argument and returns

Q: Would Scheme accept ((twiceV2 plusOne) 3)?

Exercise

- What is the difference between the following two functions?
  - (lambda (x y) (+ x y))
  - (lambda (x)
      (lambda (y)
        (+ x y)))

Let constructs

- (let ((x_1 E_1) (x_2 E_2) … (x_k E_k)) E)
  - Semantics
    - E_1, …, E_k are all evaled; then E is evaled, with x_i representing the value of E_i. The result is the value of E
    - The scope of x_1, …, x_k is E
  - Simultaneous assignment
  - examples
    - (* (+ 3 2) (+ 3 2)) is OK, but repetitive
    - writing (let ((x (+ 3 2)) (* x x))) is better
    - (+ (square 3) (square 4)) to
      - (let ((three-sq (square 3)) (four-sq (square 4))) (+ three-sq four-sq))
    - (define x 0)
      (let ((x 2) (y x)) y) to 0

(define x 5)

x = 0

x = 2

y = x

Top Hat

PennState

Let* constructs

- (let* ((x1 E1) (x2 E2) … (xk Ek)) E)
  - binds x_i to the val of E_i before E_{i+1} is evaled
  - The scope of x_1 is E_2, E_3,... and E_k and E
  - example:

    (define x 0)

    (let ((x 2) (y x)) y) to 0

    (let* ((x 2) (y x)) y) to 2
  - let* is a syntactic sugar
    - (let* ((x 2) (y x)) y)

      = (let ((x 2)) (let ((y x)) y)

```
int x = 5;
int y = 6;
int z = 7;
```

```
x = 2;
{
y = x;
```

Your Turn

```
(define x 0)
(define y 1)
(let* ((x y) (y x)) y)
```

A. 1
B. 0
C. 2
D. Neither

Letrec constructs

- (letrec ((x1 E1) (x2 E2) … (xk Ek)) E)
  - The scope of x_1 is E_1, E_2,… and E_k and E
- (letrec
  ((fact (lambda (n)
    (if (= n 0) 1 (* n (fact (- n 1)))))))
  (fact 3))

the let won't work