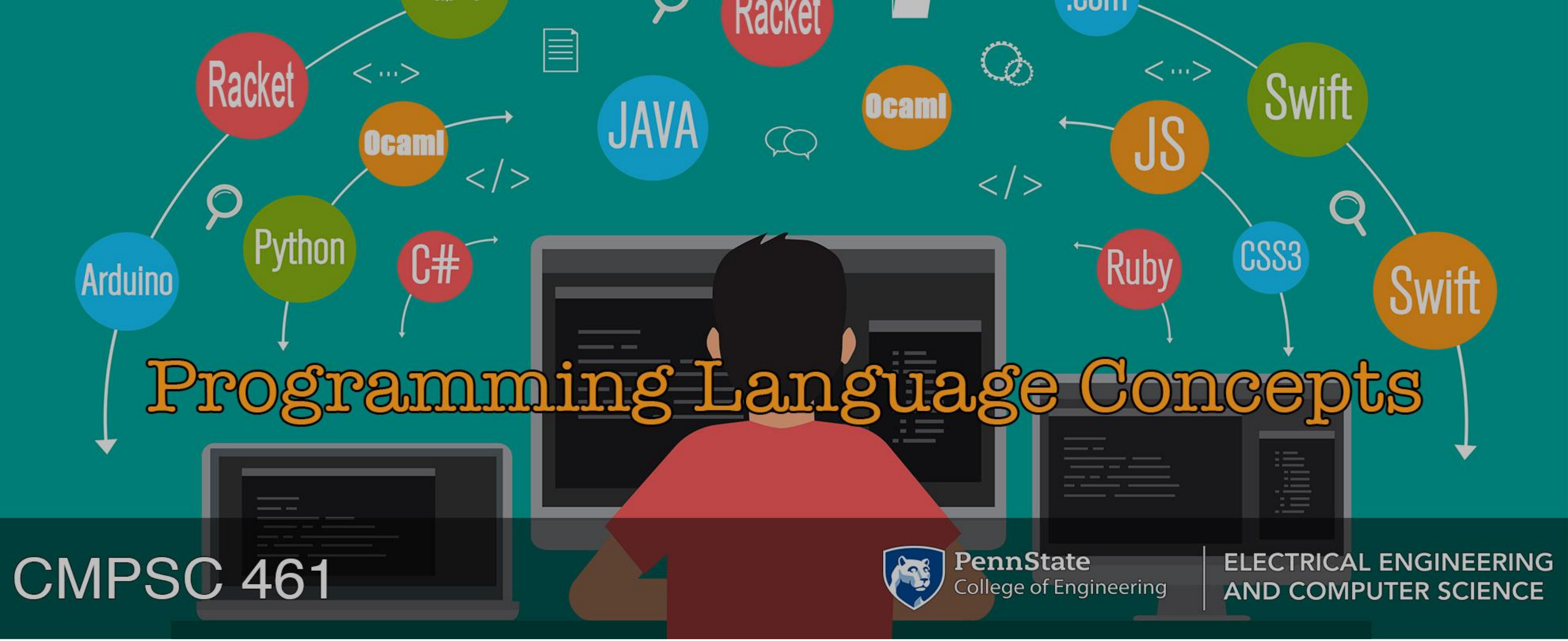

Functional Programming

(Lists)

Professor: Suman Saha

How to Distinguish Between Function Application and Data: Quoting

- A quoted item evals to itself
  - treating expressions as data
  - (+ 2 3) to 5
  - (quote (+ 2 3)) to (+ 2 3)
  - (quote pi) to pi
  - 'pi to pi
- Example
  - (fun A)
    - will try to apply function fun to the variable (parameter A)
    - will evaluate A to it’s value
  - (fun 'A): will apply it to the symbol “A”

'A
'B
'C

Lists

- LISP stands for list processing
- A list is a sequence of zero or more items
- In scheme
  - null list: `()
  - `(it seems that): a three element list
  - `((it seems that) you (like) me)
    - four elements, the first and third of which are lists
  - Parentheses are important
    - like is a symbol/atom, but (like) is a list with one element

( )   '( )

1 2 3

[ ]

[a]

[a b c]

[[ ] b [ ]]

'( )

'(a)

'(a ()())

(+ a b)

'(+ a b)

like

(like)

Lists

- The diff between
  - (it seems that you like me) 6
  - ((it seems that) you (like) me) → 4
- The diff between (a) and (a ()) 2 1
- Is (+ 2 3) a exp or a list?
  - Both
  - Scheme interpreter interprets (+ 2 3) as an exp, and responds with its value 5
  - It's also a list: three elements
  - quoting tells Scheme to interpret it as a list
    - '(+ 2 3) gets (+ 2 3)

(Built-in) Operations On lists

- (null? x): true if x is the empty list and false otherwise
- (car x): the first element of a nonempty list x
- (cdr x): the rest of the list x without the first element
  - It always returns a list
- (cons a x): returns a list whose car (head) is a and cdr (tail) is x

(cons 1 x) => (1 1 2 3)

(cons (1) x) => ((1) 1 2 3)

x = (1 2 3)

(car x) => 1 → head

(cdr x) => (2 3) → tail

Operations On lists

x = (1 2 3)

- (null? x) #f
- (null? ‘()) #t
- (car ‘(a)) = a; (cdr ‘(a)) = ()
- (define x ‘((it seems that) you (like) me))

  1 2 3 4
  - (car x) → (it seems that)
  - (car (car x)) → it
  - (cdr (car x)) → (seems that)
  - (cdr x) → (you (like) me)
  - (car (cdr x)) → you
  - (cdr (cdr x)) → ((like) me)
- Syntactic sugars: (car (cdr x)) as (cadr x)

C

Your Turn

(define x ‘((it seems that) you (like) me))

What’s the result of (car (car x))?

A. (it seems that)
B. (it)
C. it
D. seems
E. it seems

Your Turn

PennState

(define x ‘((it seems that) you (like) me))

What’s the result of (cdr (cdr x))?

A. you
B. (like me)
C. ((like) me)
D. (like) me
E. (like)

Cons

- (cons a l)
  - cons takes two arguments: the first one is any exp, the second one is usually a list
  - Returns a list whose head is a and tail is l
- Examples
  - (cons ‘a ‘()) = (a)
  - (cons ‘(a b (c)) ‘()) = ((a b (c)))
  - (cons ‘a (car ‘((b) c d))) = (a b)
  - (cons ‘a (cdr ‘((b) c d))) = (a c d)

□ For any a and x, (car (cons a x)) = a; (cdr (cons a x)) = x

- ‘(it seems that) same as
  - (cons ‘it (cons ‘seems (cons ‘that ‘())))
  - ‘(it . (seems . (that . ())))
  - same as (list ‘it ‘seems ‘that)

(car ‘((b) c d)))

(cons ‘a (b)) => (a b)

cdr

(c d))

(cons ‘a (c d))

(a c d)

→(that)

→(seem that)

(it seem that)

Your Turn

What's the result of (cons ‘a ‘())?

A. a
B. (a)
C. ((a))
D. ()
E. None of the above

Your Turn

How should we use cons to produce ‘(a b)?

A. (cons ‘a ‘b)
B. (cons a b)
C. (cons ‘a (cons ‘b ‘()))
D. (cons a (cons b ‘()))
E. None of the above

List Manipulation: Length

- (define (length lst)
  (cond ((null? lst) 0)
  (else (+ 1 (length (cdr lst))))))
- Programming pattern: case analysis and recursion
- Two cases
  - When lst empty, return 0
  - When lst is nonempty, the length is one plus the length of the tail of lst
- Examples:
  - (length ‘(a b c)) ⇒ 3
  - (length ‘((a) b (a (b) c))) = 3

1 2 3

(length '(a b c))

3=2+1 (+1 (length (b c)))

2=1+1 (+1 (length (c)))

1=0+1 (+1 (length ()))

0

Appending Two Lists

(cons a x)

(append L1 L2)

L1 L2

- (append ‘() ‘ (a b c d)) = (a b c d)
- (append ‘(a b c) ‘(d)) = (a b c d)
  - Note append is different from cons
- Two cases for (append l1 l2)
  - When l1 is null, then return l2
  - When l1 is not null, put (car l1) and (append (cdr l1) l2)) together via cons
- (define (append l1 l2)

```
(define (append l1 l2)
    (if (null? l1) l2
        (cons (car l1) (append (cdr l1) l2))))
```

Appending Two Lists

- Invocation graph for (append ‘(a b c) ‘(d)) => (a b c d)

L1 L2
(append '(a b c) '(d)) =

(a b c d) ← (cons a (append '(b c) '(d)))   [L1 = (b c), L2 = (d)]

(b c d) ← (cons b (append '(c) '(d)))   [L1 = (c), L2 = (d)]

(c d) ← (cons c (append '( ) '(d)))

'(d)

Your Turn

What is the result of (append ‘(a b) ‘(c d))?

A. ‘(a b c d)
B. `((a b) c d)
C. ‘(((a b)) c d)

Member

- (define (member? a lst)

  (cond ((null? lst) #f)

  ((equal? a (car lst)) #t)

  (else (member? a (cdr lst)))))
- Examples
  - (member? 3 ‘(1 3 2)) returns #t
  - (member? ‘a ‘(a b c)) returns #t
  - (member? ‘(a) ‘((a) b c))) returns #t
- Note that equal? can also compare lists
  - In contrast, = compares only numbers

Mapping a function across list elements

- (map square '(1 2 3 4)) = (1 4 9 16)
- (map plusOne '(3 7 8 9)) = (4 8 9 10)
- Two cases
  - (map f ()) = ()
  - (map f (cons a y)) = (cons (f a) (map f y))
- (define (map f x)

```
      (if (null? x) '()
          (cons (f (car x)) (map f (cdr x)))))
```

Mapping a function across list elements: Examples

- (map square ‘(1 2 3)) = (1 4 9)
  - draw the invocation graph
- Examples
  - (map (lambda (x) (> x 10)) ‘(3 7 12 9))
  - (map (lambda (x) (if (even? x) ‘Even ‘Odd)) ‘(3 7 12 9))
  - (map length ‘((a) (a b) (a b c) ()))

(map sq '(1 2 3))

(1 4 9)   (cons (sq 1) (map sq '(2 3)))

(4 9)   (cons (sq 2) (map sq '(3)))

(9)   (cons (sq 3) (map sq '()))

()

Your Turn

What's the result of

(map (lambda (x) (list x (+ x 1))) '(3 7 12 9))?

A. (4 8 13 10)
B. (3 7 12 9)
C. (3 4 7 8 12 13 9 10)
D. ((3 4) (7 8) (12 13) (9 10))
E. None of the above

Reduce

- (reduce + ‘(2 4 6) 0) = 2 + 4 + 6 + 0 = 12
- (reduce * ‘(2 4 6) 1) = 2 * 4 * 6 * 1 = 48
- (define (reduce f l v)

      (if (null? l) v

         (f (car l) (reduce f (cdr l) v))))

draw the invocation graph

(reduce + (2 4 6) 0)

12 (+ 2 (reduce + (4 6) 0))

10 (+4 (reduce + (6) 0))

6 (+6 (reduce + () 0))

0

Your Turn

#f

#f

#t

What's the result of

(reduce (lambda (x y) (and x y)) '(#t #f #t) #t)?

A. #t

B. #f

C. (#t #f #t #t)

D. Runtime error

E. None of the above

Association Lists

PennState

- A list of pairs
  - ((a 1) (b 2) (c 3) …)
  - Called dictionaries in some languages: map keys to values
  - Can be used to implement symbol tables: map a var to its associated bindings
- bind: returns an association list with a new binding for a key
  - What happens if there is already a binding for the key
    - Two choices: remove the old binding, or keep it
  - (define (bind key value env)
    (cons (list key value) env))
  - Examples
    - (bind ‘d 4 ‘((a 1) (b 2) (c 3))) => ((d 4) (a 1) (b 2) (c 3))
    - (bind ‘a 10 ‘((a 1) (b 2) (c 3))) => ((a 10) (a 1) (b 2) (c 3))

((a 6) (b 6) (e 9))

7

(a 7)

Association Lists

- lookup: look up the value for a key in an association list; return the key-value pair
  - b

    ```
    (define (lookup key al)
        (cond ((null? al) #f)
              ((equal? key (caar al)) (car al))
              (else (lookup key (cdr al)))))
    ```
  - a built-in Scheme function called assoc
  - Examples
    - (lookup ‘a ‘((a 1) (b 2) (a 3))) -> ‘(a 1)
    - (lookup ‘b ‘((a 1) (b 2) (a 3))) -> ‘(b 2)
    - (lookup ‘c ‘((a 1) (b 2) (a 3))) -> #f

car ((a 5) (b 6) (c 8))

al

car (a 5)

a

(and #t (look))