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”

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

Lists

- The diff between
  - (it seems that you like me)
  - ((it seems that) you (like) me)
- The diff between (a) and (a ())
- 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

Operations On lists

- (null? x)
- (null? ‘())
- (car ‘(a)) = a; (cdr ‘(a)) = ()
- (define x ‘((it seems that) you (like) me))
  - (car x)
  - (car (car x))
  - (cdr (car x))
  - (cdr x)
  - (car (cdr x))
  - (cdr (cdr x))
- Syntactic sugars: (car (cdr x)) as (cadr x)

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

(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)

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))))))

```
(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))
  - (length ‘((a) b (a (b) c)))

Appending Two Lists

- (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)

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

Appending Two Lists

- Invocation graph for (append ‘(a b c) ‘(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)

```
(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)

```
(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) ()))

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

Your Turn

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

- 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)))
    - (bind ‘a 10 ‘((a 1) (b 2) (c 3)))

Association Lists

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