Structure of Lists

- Lisp has one data structure - list
- program and data are represented as lists
- a list is zero or more elements in parentheses
- elements are boolean, numbers, symbols, other lists, strings, characters, functions

```
()
(a b c)

(a (b c))

(((a) b) c)

(a ())
```

- `(+ 2 3)` - expression. `'( + 2 3)` - list
- quote stops evaluation of a list
  `quote (+ 2 (* 3 4))` ⇒ `(+ 2 (* 3 4))`

Operations on Lists

`(null? x)`	is list `x` empty
`(car x)`	first element of (non-empty) list `x`
`(cdr x)`	all elements of x except the first
`(cons x y)`	make a list with `x` as the first element of list `y`
`(list a b c)`	make a list with elements `a`, `b`, `c`

```
(list 'a 'b '(c d)))  ⇒
(list '() 'a))        ⇒
(list 'a '(a)))       ⇒
(car '(a b c)))       ⇒
(car '()))            ⇒
(car '((a) b c)))     ⇒
(cdr '(b c)))         ⇒
(cdr '(c)))           ⇒
(cdr '((c))))         ⇒
(cons 'a '(b c)))     ⇒
(cons 'a 'b))         ⇒
(cons '(a) '(b c)))   ⇒
(caar '((a) b c)))    ⇒
(cadr '((a)(b c))))   ⇒
(cdar '((a)(b c))))   ⇒
```


Other Operations on Lists

```
(reverse '(a b c)) ⇒ (c b a)
(append '(a b c) '(d e)) ⇒ (a b c d e)
(length '(a b c)) ⇒ 3
(min '(1 2 3)) ⇒ 1
(max '(1 2 3)) ⇒ 3
(apply + '(1 2 3)) ⇒ 6

(define (plus-2 x) (+ 2 x))
(map plus-2 '(1 2 3)) ⇒ (3 4 5)
```


Structure of Lists (Cont'd)

- `cons` is a list constructor in graphical representation!

Conditional Evaluation

If statement

(if *bool-expr then-expr else-expr*)

Examples:

```
(if (< 2 3) 5 7)
(if #t #t #f)
(* 2 (if (zero? x)
   (if (zero? (- x 1)) 1 2) 4))
```

Multiway branching -

(cond (T_1 E_1) ... (T_n E_n) (else E))

Example:

```
(cond ((equal? x 0) 'zero)
  ((< x 0) 'less)
  (else 'greater))
```


Recursive Programming

- Very powerful tool for processing *recursive types*
- In Scheme - mostly lists
- Can reason inductively about recursive types.

Example:

1. 0 is a number
2. if n is a number then (+1 n) is a number
3. nothing else is a number.

0 and +1 are *constructors* for the datatype pos. integer:

3 = (+1 (+1 (+1 0)))

Also need *accessors* - procedures to take datatype apart: zero?, pred:

```
(zero? 0) = #t
(zero? (+1 n)) = #f
(pred (+1 n)) = n
```


Recursion (Cont'd)

Now addition of numbers (binary + from unary +1):

```
n + 0 = n
n + m = (+1 n) + (pred m)
```

Convert into Scheme:

```
(define (+ m n)
  (cond ((zero? m) n)
  (else (+1 (+ (pred m) n)))))
```


Recursive Programming using Lists

1. '() is a list (the empty list)
2. if E is an expression, L is a list, then (cons E L) is a list
3. nothing else is a list.

'(a b c) = (cons 'a (cons 'b (cons 'c '())))

Accessors: null?, car, cdr

```
(null? '() ) = #t
(null? (cons E L)) = #f
(car (cons E L)) = E
(cdr (cons E L)) = L
```

`Length` function

```
(length '()) = 0
(length (cons E L)) = (+1 (length L))
```

And now in Scheme:

```
(define (length L)
  (cond ((null? L) 0)
  (else (+1 (length (cdr L))))))
```

Tracing:

```
(length '(1 2 3))
     = (+1 (length '(2 3)))
     = (+1 (+1 (length '(3))))
     = (+1 (+1 (+1 (length '()))))
     = (+1 (+1 (+1 0))) = 3
```


Function `sum`

`sum '(1 2 3)` ⇒ `6`

Definition:

```
(sum '()) = 0
(sum (cons E L)) = E + (sum L)
```

In Scheme:

```
(define (sum L)
  (cond ((null? L) 0)
  (else (+ (car L) (sum (cdr L))))))
```

So, most of functions for lists are:

`(define (f x)`
 if list `x` is empty then...
  otherwise do something involving `(car x)`, `(cdr x)` and `f`.

`Member?` Function

```
(member? x '()) = false
(member? x (cons E L)) =
 if (x = E) then true else (member? x L)
```

And in Scheme:

```
(define (member? x L)
  (cond ((null? L) #f)
  ((equal? x (car L)) #t)
  (else (member? (x (cdr L))))))
```

Tracing:

```
(member? 'c '(a b c))
   = (member? ('c '(b c)))
   = (member? 'c '(c))
   = (member? 'c '()) = #t

(member? 'd '(a b c))
   = (member? 'd '(b c))
   = (member? 'd '(c))
   = (member? 'd '())
   = #f
```


`Append` Function

Append second list at the end of the first:

`(append '(a b c) '(d e f))` ⇒ `(a b c d e f)`

Definition:

```
(append '() B) = B
(append (cons E L) B) = (cons E (append L B))
```

In Scheme:

```
(define (append La Lb)
  (cond ((null? La) Lb)
  (else (cons (car La) (append (cdr La) Lb)))))
```

Tracing:

```
(append '(a b) '(c d))
   = (cons 'a (append '(b) '(c d)))
   = (cons 'a (cons 'b (append '() '(c d))))
   = (cons 'a (cons 'b '(c d)))
   = (cons 'a '(b c d))
   = '(a b c d)
```

`Reverse` Function

(reverse '(1 2 3)) ⇒ (3 2 1)

Definition:

```
(reverse '()) = '()
(reverse (cons E L)) = (append (reverse L)
(E))
```

In Scheme:

```
(define (reverse L)
  (cond ((null?  L) '())
    (else (append (reverse (cdr L)) (list (car
L))))))
```

What is complexity of `reverse`?

Improving Performance: Accumulators and Auxiliary Functions

Motivation:

(reverse (E_1 E_2 ... E_n) =
  (cons E_n (reverse (E_1 ... E_{n-1}))))

Have a function that has current list and result so far - accumulator:

```
(rev L Result) = if L = '() then Result
    else (rev cdr(L) (cons (car L) Result))
```

But we want our `reverse` to have only one parameter! Result:

```
(define (rev L Result)
  (cond ((null?  L) Result)
  (else (rev (cdr L) (cons (car L) Result)))))
(define (reverse-2 L) (rev L '()))
```

What is complexity of `reverse-2`?

Tracing `Reverse-2`

```
(define (rev L Result)
  (cond ((null?  L) Result)
  (else (rev (cdr L) (cons (car L) Result)))))
(define (reverse-2 L) (rev L '()))
```

```
(reverse-2 '(1 2 3)) = (rev '(1 2 3) '())
   = (rev '(2 3) '(1))
   = (rev '(3) '(2 1))
   = (rev '() '(3 2 1))
   = '(3 2 1)
```


Other Types of Recursion

Function `leftmost`

```
(leftmost '((((2 3)) (4 (5 6))) 7))
  ⇒ 2
```

```
(define (leftmost l)
  (cond ((null? l) '())
        ((not (list? (car l))) (car l))
        (else (leftmost (car l)))))
```

Trace:

```
(leftmost '((((2 3)) (4 (5 6))) 7))
  = (leftmost '(((2 3)) (4 (5 6))))
  = (leftmost '((2 3)))
  = (leftmost '(2 3))
  = 2
```

This is `car` recursion.

Other Types of Recursion

Function `flatten`

```
(flatten '(((2 3) (4 (5 6))) 7))
```

⇒ (2 3 4 5 6 7)

```
(define (flatten l)
 (cond
  ((null? l) '())
  ((not (list? (car l)))
    (cons (car l) (flatten (cdr l))))
  (else
    (append (flatten (car l))
            (flatten (cdr l))))))
```

Tracing:

```
(flatten '((2 3) (4 (5 6))))
 = (append (flatten '(2 3)) (flatten '(4 (5 6))))
 = (append (cons 2 (flatten (3))) (flatten '(4 (5 6))))
 = (append (cons 2 (cons (3 (flatten '()))))
           (flatten '(4 (5 6))))
 = (append (cons 2 (cons (3 '()))) (flatten '(4 (5 6))))
 = (append (cons 2 (3)) (flatten '(4 (5 6))))
 = (append (2 3) (flatten '(4 (5 6))))
 = ...
 = (append (2 3) (4 5 6))
 = (2 3 4 5 6)
```

This is `car` and `cdr` recursion.

food for slide eater

food for slide eater

food for slide eater