Linear Recursion

A recursive function is linear if it calls itself at most once at each level of recursion.

Non linear recursion

```
(count '(A B)) => 2
(count '((A B)(C D))) => 4
(count '(1 a 2 b)) => 2
(count '(a () b)) => 0
(count 1) => 0
(count '()) => 0
(count '()) => 0
```

```
(define (count E)
    (cond ((null? F) 0)
          ((number? E) 0)
          ((symbol? E) 1)
          (else (+ (count (cor E))
                   (cont (cdr E)))))))
```

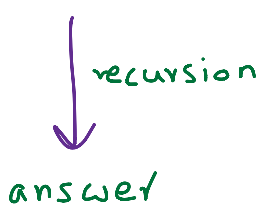
```
              (count ((1 a) b c))

(count (1 a))          +          (count (b c))

(count 1) + (count (a))   (count b)  +  (count (c))
```

0 1

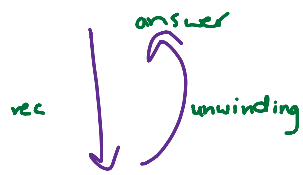
(count a) + (count ()) (count c) + (count ())

1 0 1 0

Tail Recursion



Linear (non-tail) recursion



non-tail recursion

Mutual Recursion

Function F_1 calls F_2 calls F_3 … calls F_k calls F_1

$F_1, F_2, \ldots, F_k$ are mutually recursive

(even? L)

(odd? L)

(even? '(a b c d)) => #t

(even? '(a b c)) => #f

(odd? '(a b c)) => #t

(odd? '(ab)) => #f

(even? '()) => #t

(odd? '()) => #f

```
(define (even L)
        (if (null? L) #t (odd?(cdr ())))

(define (odd? L)
         (if (null? #f (even?(cdr ()))))

(even? '(a b c d))
=> (odd? '(b c d))
=> (even? '(c d))
=> (odd? '(d))
-> (even? '())
=> #t                          (even? '())=> #t
                               (odd? '())=> #f

(reverse '(a b c d)) => (d c b a)
-> (reverse L)
-> (reverse2 L1 L2)
(reverse? '(a b c d) '(1 2 3 4))=> (d c b a 1 2 3 4)
(reverse L)= (reverse2 L '())

(define (reverse L)
        (reverse2 L '()))

(reverse '() L)=> L
```

```
(reverse2 '(a b c) '(1 2 3)) => (reverse '(b c) '(a 1 2 3))

(define (reverse2 L1 L2)
        (if (null? L1)
            L2
            (reverse 2 (cdr L1)
                        (cons (car L1) L2))))
```

```
(reverse (a b c d))
=> (reverse2 (a b c d) ())
=> (reverse2 (b c a) (a))
=> (reverse2 (c d) (b a))
=> (reverse2 (d) (c b a))
=> (reverse2 () (d c b a))
=> (d c b a)
```

```
(3sum '(1 2 3 4 5)) ==> 3 + 4 + 5 = 12

(define (3sum L)
      (+ (first (reverse L))
         (second (reverse L))
         (third (reverse L))))
```

```
(define (3sum L)
        (3sum_help (reverse L)))

(define (3sum_help RL)
        (+ (first RL)
```

(second RL)

(third RL)))

Let expressions

```
(Let ((var1 exp1)
      (var2 exp2)
       ⋮
      (varn expn)
      exp)
```

First create new variables, $var_1 \dots var_n$

Then evaluate $exp_1 \dots exp_n$

Then initialize each var_i to the value of exp_i

Evaluate exp + return its value

```
(define (sum L)
        (let ((RL (reverse L))
             (+ (first RL)
                (second RL)
                (third RL)))
```

$(x+y)^3 + (x+y)^3$

```
(define (dcube x y)
        (let ((D (-x y))
              (S (+ S Y)
```

```
(+ (* D D D) (* S S S))))
```

```
(Let * ((v1 E1)
        (v2 E2)
        (v3 E3))
```

Scoping Rules

```
(let ((x 1) (y 2))
11
     (+ y (let ((Y 5)(z 3))
  11   2   9
                (+ x y z))))
                   1 5 3
        3+5+1
        = 9
```

```
(let ((x 3))
9
     (* x
     9
        (let ((x 2)) (-x 1))
        1            1  2
         x))
        3
```

```
(define (remove-even L)
        (cond ((null? L) '())
              ((even? (car L))(remove-even (cdr L)))
              (else (cons (car L)(remove-even (cdr L)))
```

```
(define (remove-odd L)
        (cond ((null? L) '())
```

```
((odd? (car L)) (remove-odd cdr L)))
(else (cons (car L)(remove-odd cdr L))))
```

Higher order

```
(define (remove-if P L)
    (cond ((null? L) '())
    ((P (car L))(remove-if P (cdr L)))
    (else (cons (car L)(remove-if P (cdr L)))

(define (square-all L)
    (cond ((null? L) '())
        (else (cons (square (car L))
                (square-all (cdr L))))))

(define (map F L)
    (cond ((null? L) '())
        (else (cons (F (car L))
                (map F (cdr L))))))

(let ((F car) (G cdr))
    (F (g '(a b c))))
=> (car (cdr '(a b c d))
        b c d
         b
```

```
(Let ((sq (lambda (x) (* x x)))
      (cube (lambda (x) (* x x x))))
      (+ (sq 3) (cube 2)))
```

$\Longrightarrow 3^2 + 2^3 = 17$

Lexical Scoping

```
(Let ((f (lambda (x) (* x 3))))
     (f 2)
```

$2 \times 3 = 6$

```
(let ((z 3))
     (let ((f (lambda (x) (* x z))))
          (let ((z 1))
               (f z))))
```

Global variables contained within brackets