(cdr '(a b c)) => (b c)

(car '(a bc)) => a

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

Predicates

#t #f

(symbol? 'a) => #t

(_ 2) => #f

(_ '(a b)) => #f

(symbol? (car '(abc))) => #t a

(symbol? (cdr '(a bc))) => #f (bc)

(symbol? (+ 3 4)) => #f 7

(list? x) => #t x => a list

(list? '(A B)) => #t

(list? 3) => #f

(list? (+ 3 4)) => #f

(list? '(+ 3 4)) => #t

(list? '()) => #t

(Null? x) => #t iff x => ()

(null? (cdr '(a b))) => #f

(null? (cdr (cdr '(a b)))) => #t

(equal? x y) => #t iff x + y are the same exprs

(equal? 10 (+ 6 4)) => #t

(equal? '(b c) (cdr '(a b c))) => #t

(equal? (+ 5 5) (+ 6 4)) => #t

(equal? '(+ 5 5) '(+ 6 4)) => #f

Arithmetic Predicates

(zero? x) => #t iff x => 0

(zero? 0/6) => #t

(zero? 0.0) => #t

(zero? (-3 (+ 12))) => #t

(zero? (+ 3 3)) => #f

(zero? '(+ 3 3)) => error

(< x y) => #t iff x < y

(> x y)

(= x y)

(= 1/4 0.25) => #t

(equal? 1/4 0.25) => #f

(= 2 2.0) => #t

(equal? 2 2.0) => #f

Boolean operators

(And $x_1 x_2 \ldots x_n$) => #t iff every x_i => #t

(And (< 3 4)
     (> 5 6)
     (< > 8)) => ~~#t~~ #f

(Or $x_1 x_2 \ldots x_n$) => #t iff some x_i => #t

(Or (< 3 4)
    (> 5 6)
    (< > 8)) => #t

(or (and (list? '(a b)) (< 13 7))
    (and (< 4 5) (equal? 4 (+ 3 2)))) => #f

(not? x) => #t iff x => #f

(not? (> 1 3)) => #t

(not? (< 1 3)) => #f

Conditional Expressions

(cond (c_1 E_1) For the first c_i that evaluates to #t, evaluate E_i +
      (c_2 E_2) return its value
      ⋮

$(c_n \ldots E_n)$

```
(cond ((≥ 'Grade 80) 'A)
      ((≥ Grade 70 'B)
      ((≥ Grade 60) 'C)
      ((≥ Grade 50) 'D)
      ((≥ Grade 40) 'E)
      (#t 'f))
      else
```

```
(cond (c E1)          if c then E1 else E2
      (#t E2))
```

↙

```
(if c E1 E2)
```

```
(if (> x 0) x (- x)) => |x|          if x => -3 then x => (-x) => 3
```

Submachines

```
(define (name arg1 arg2 ... argn) expr)
```

name: function name; arg1 arg2 ... argn: arguments; expr: function body

```
(define (second list) (car (cdr list)))
```

```
(second '(a b c d)) => b
```

```
(define (abs x y)
        (if (> x y) (- x y)
```

(- y x))) => |x - y|

(abs 5 6) => 1 value of x becomes 5

value of y becomes 6

(first1 '(a b c d)) => (a)

(first2 '(a b c d)) => (a b)

⋮

```
(define (first1 L)  (cons (car L) '()))
(define (first2 L)  (cons (car L) adds element 1 to
                          (first1 (cdr L))))  first element excluding element 1
```

```
(define (first3 L) (cons (car L)
                         (first2 (cdr L))))
```

(last '(a b c d)) => d

```
(define (last L)
        (if (null? (cdr L)) (car L) (last (cdr L))))
```

Def: A function, f, is cdr recursive if given a list-valued argument, L, the function calls itself with argument (cdr l)

i.e. To evaluate (f L) we must first evaluate (f (cdr L))

(leftmost '(a b c)) => a

(leftmost '(((a b) c) d)) => a

(leftmost a) => a

```
(define (leftmost E)
        (if (symbol? E) E
            (number? E) E
            (leftmost (car E))))
```

```
(define (leftmost E)
        (cond ((symbol? E) E)
              ((number? E) E)
              ((null? E) E)
              (else (leftmost (car L)))
                    (() a b c)
```

Def: F is tail recursive, if at every call (except the last) the value of f is the value returned by another call to F.

eg. value of '(F L) is (F (cdr L))

—— (F L) is (F (car L))

Nontail recursion

```
(define (fact N)
        (if (zero? N)
            1
            (* N (fact (- N 1)))))
```

(double (1 2 3)) => (2 4 6)

(double ()) => 0

```
(define (double L)
        ( if (null? L)
             L
             (cons (* 2 (car L))
                   ( double (cdr L)))))
```