append

e.g. (append '(a b) '(c d)) => (a b c d)

```
(define (append L1 L2)
  (if (null? L1)
      L2
      (cons (car L1) (append (cdr L1) L2))))
```

e.g. (append '(a b) '(c d))

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

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

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

= (a b c d)

tail-recursive version

```
(define (append L1 L2)
  (define (append1 L1 L2)
    (if (null? L1)
        L2
        (append1 (cdr L1) (cons (car L1) L2))))
  (append1 (reverse L1) L2))
```

accumulator variable

assume (reverse '(a b c)) => (c b a)

e.g. (append '(a b) '(c d))

= (append1 '(b a) '(c d))

= (append1 '(a) '(b c d))

= (append1 () '(a b c d))

= (a b c d)

tail-recursive reverse

```
(define (reverse L)
  (define (reverse1 L1 L2)
    (if (null? L1)
        L2
        (reverse1 (cdr L1) (cons (car L1) L2))))
  (reverse L ()))
```

e.g. (reverse '(a b c))

= (reverse1 '(a b c) ())

= (reverse1 '(b c) '(a))

= (reverse1 '(c) '(b a))

= (reverse1 () '(c b a))

= (c b a)

Functions as first-class values

e.g. (map addone '(1 2 3)) => (2 3 4)

```
(define (addone n) (+ n 1))
(define (map f L)
  (define (map1 f L1 L2)
    (if (null? L1)
        L2
        (map1 f (cdr L1)
              (cons (f (car L1)) L2))))
  (map1 f (reverse L1) ()))
```

e.g. (map addone '(1 2 3))

= (map1 addone '(3 2 1) ())

= (map1 addone '(2 1) (cons (addone 3) ()))

= (map1 addone '(1) (cons (addone 2) '(4)))

= (map1 addone () (cons (addone 1) '(3 4)))

= (2 3 4)

Note that map expects a function of one parameter as its argument. If it doesn't then an error will occur.

```
(define (cons1 a) (cons a ()))
```

e.g. `(map cons1 '(1 a 2))`

=> `((1) (a) (2))`

Functions that take other functions as parameters are sometimes called "functionals"

filter – filters out items from a list according to some function

e.g. (filter odd? '(1 2 3 4)) => (2 4)

```
(define (odd? n) (= (mod n 2) 1))
(define (filter f L)
  (cond ((null? L) ())
        ((f (car L)) (filter f (cdr L)))
        (else (cons (car L) (filter f (cdr L))))))
```

e.g. (filter odd? '(1 2 3 4))
= (filter odd? '(2 3 4))
= (cons 2 (filter odd? '(3 4)))
= (cons 2 (filter odd? '(4)))
= (cons 2 (cons 4 (filter odd? ())))
= (cons 2 (cons 4 ())) = (2 4)

(filter const1 '(1 2 a))

Since const1 always returns a value, it is the same as returning #t (like C, when a non-zero value counts as true)

There is a "zero" value, called nil - this can serve as false (#f) if returned by a function and used in a test expression.

reduce – reduces a list to a single value using a binary function

e.g. (reduce + '(1 2 3 4) ∅) => 10

```
(define (reduce f L n)
  (if (null? L)
      n
      (f (car L) (reduce f (cdr L) n))))
```

e.g. (reduce + '(1 2 3) ∅)

= (+ 1 (reduce + '(2 3) ∅))

= (+ 1 (+ 2 (reduce + '(3) ∅)))

= (+ 1 (+ 2 (+ 3 (reduce + () ∅))))

= (+ 1 (+ 2 (+ 3 ∅)))

= 6

(reduce * '(1 2 3) ∅) => ∅

(reduce * '(1 2 3) 1) => 6

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

(This copies the list)

Semantics of Scheme

- could write an interpreter
- could use denotational semantics
- instead, we will look at a meta-circular interpreter — an interpreter for Scheme, written in Scheme

The meta-circular interpreter is an expression evaluator. In Scheme, this is called eval, and uses the fact that expressions and lists have the same syntax.

e.g. (define x '(+ 1 2))

x => (+ 1 2)

(eval x) => 3

(define x '(a b c))

(eval x) => error

Overview of meta-circular interpreter:

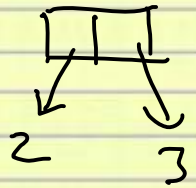
- use a binding environment for parameters
- represent it as a list

e.g. ((x . 1) (y . (a b c)) (z . a))

each item in the list is a binding pair, the LHS is a parameter name, the RHS is the value the parameter is bound to.

This is basically a cons cell

e.g. (cons 2 3) => (2 . 3)



The list above is