Natural Numbers (Ch. 11)

- 0, 1, 2, ...
- non-negative integers.

  - 0 is a natural number.
  - if *n* is a natural number, so is one more than *n*.

Scheme definition

- A natural number is:
  - 0
  - `(add1 n )` if `n` is a natural number

```
0
(add1 0)
(add1 (add1 0))
(add1 (add1 (add1 0)))
```

Natural Numbers vs. Lists

- `cons` vs. `add1`
- `rest` vs. `sub1`
- `empty?` vs. `zero?`

```
(rest (cons x l)) => l
(sub1 (add1 n)) => n
(zero? n) => boolean
```

Using natural numbers

- A function that will produce a list of *n* copies of the symbol `'hello`

```
(define (hellos n)
  (cond
    [(zero? n) empty]
    [ else
      (cons 'hello (hellos (sub1 n)))]))
```

Exercise

- Without using '+', write a function named sum that computes the sum of two natural numbers:

```
(sum 2 3) => 5

(sum 0 1) => 1

(sum 0 0) => 0
```

Sum function

```
(define (sum x y)
  (cond
    [(zero? y) x]
    [ else
      (sum (add1 x) (sub1 y))]))
```

product exercise

- Using the function sum (and no other arithmetic), write a function that computes the product of two natural numbers.

```
(product 3 4) => 12

(product 0 2) => 0

(product 1234 0) => 0
```

Sample `product`

```
(define (product x y)
  (cond
    [(zero? x) 0]
    [(zero? y) 0]
    [(= 1 x) y]
    [else
     (sum y (product (sub1  x)  y))]))
```

`listinsert` exercise

- write a function named listinsert that will add a new element to a list at a specified position:

```
(listinsert elem pos list)

(listinsert 2 3 (list 'a 'b 'c 'd 'e)) =>

   (list 'a 'b 3 'c 'd 'e)
```

listinsert solution

```
(define (listinsert x pos l)
  (cond
    [(empty? l) empty]
    [(= 1 pos) (cons x l)]
    [else
     (cons (first l)
           (listinsert x (sub1 pos)
                       (rest l)))]))
```

Exercise

- Create a function named countdown that creates a list of natural numbers from n down to (and including) 1:

```
(countdown 3) =>
   (cons 3 (cons 2 (cons 1 empty)))

(countdown 0) =>
   empty
```

Sample Solution

```
(define (countdown n)
  (cond
    [(zero? n) empty]
    [ else
      (cons n (countdown (sub1 n)))]))
```

countup ?

- Create a function named countup that creates a list of natural numbers from 1 up to (and including) n:

```
(countup 3) =>
   (cons 1 (cons 2 (cons 3 empty)))

(countup 0) =>
   empty
```

A Different countup

- Creates a list of numbers from *start* to *end*

```
(define (countupnew start end)
  (cond
    [(= start end) (cons end empty)]
    [ else
      (cons  start
             (countupnew (add1 start) end))]))
```

countupnew

```
(countupnew 4 6) =>
  (cons 4 (cons 5 (cons 6 empty)))

(countupnew 1 3) =>
  (cons 1 (cons 2 (cons 3)))
```

This is what we want for
(countup 3)

`countup` using `countupnew`

```
(define (countup n)
  (countupnew 1 n))
```

- There are many situations where we can't write a function without first writing a *helper* function.

countup helper

```
(define (countup-helper start end)
  (cond
    [(= start end) (cons end empty)]
    [ else
      (cons  start
         (countup-helper (add1 start) end))]))

(define (countup n) (countup-helper 1 n))
```

Exercises

- reverse – reverses a list
- integer division (natural numbers) with no -
  - (divide x y) => x/y
- sorted list of number:
  - add a new number to a sorted list of numbers
  - sort a list of numbers
  - search a sorted list for a number (produces a boolean).