Little Schemer
Chapter 4

A lot of Chapter 4 is based on trying to replicate arithmetic operations using only the primitives add1, sub1 and zero?

Note that this isn't more efficient - most arithmetic operations are built into processors and they will run faster than any substitutes you can define recursively. But it is interesting to see what you can build out of a few operations and it has consequences for the theory of programming languages. It is also great practice for recursion.

Assuming m and n are non-negative numbers

```
(define my+ (lambda (m n)
        (cond
                [(zero? m) n]
                [else (add1 (my+ (sub1 m) n)])))

(define my- (lambda (m n)
        (cond
                [(zero? n) m]
                [else (sub1 (my-  m (sub1 n)))])))
```

```
(define my* (lambda (m n)
        (cond
                [(zero? n) 0]
                [else (my+ m (my* m (sub1 n)))])))

(define my< (lambda (m n)
        (cond
                [(and (zero? m) (zero? m)) #f]
                [(zero? m) #t]
                [(zero? n) #f]
                [else (my< (sub1 m) (sub1 n))])))
```

```
(define my/ (lambda (m n)
        (cond
                [(my< m n) 0]
                [else (add1 (my/ (my- m n) n))])))

(define my% (lambda (m n)
        (cond
                [(my< m n) m]
                [else (my% (my- m n) n)])))
```

etc.