Chapter 1: Building Abstractions with Procedures

Exercise 1.1

- `10`
- `(+ 5 3 4)` → 12
- `(- 9 1)` → 8
- `(/ 6 2)` → 3
- `(+ (* 2 4) (- 4 6))` → 6
- `(define a 3)` → Stores 3 into var *a*
- `(define b (+ a 1))` → Stores 4 `(+ 3 1)` into var *b*
- `(+ a b (* a b))` → 19
- `(= a b)` → NIL
- ```
  (if (and (> b a) (< b (* a b)))
      b
      a)
  ```
  ↪ 4
- ```
  (cond ((= a 4) 6)
        ((= b 4) (+ 6 7 a))
        (else 25)))
  ```
  ↪ 16
- `(+ 2 (if (> b a) b a))` → 6
- ```
  (* (cond ((> a b) a)
           ((< a b) b)
           (else -1))
     (+ a 1))
  ```
  ↪ 16

Exercise 1.2

```
(/ (+ 5 4 (- 2
             (- 3
                (+ 6
                   (/ 4 5)))))
   (* 3
      (- 6 2)
      (- 2 7)))
```

Exercise 1.3

```
(define ex1.3 (x y z)
      (cond ((> x y)
              (if (> y z)
                  (+ (* x x) (* y y))
                  (+ (* x x) (* z z))))
            (t
              (if (> x z)
                  (+ (* y y) (* x x))
                  (+ (* y y) (* z z))))))
```
```

Exercise 1.4

The function `a-plus-abs-b` utilizes the if condition to change the operation to a sum if b is positive or a substraction otherwise, acting as $|b|$.

Mathematically:

$$\text{a-plus-abs-b}(a, b) = \begin{cases} a+b \text{ if } b>0 \\ a-b \text{ if } b<0 \end{cases} \equiv a + |b|$$

Exercise 1.5

With an applicative order evaluation, the test function will not run properly because `(p)` will loop on itself, continiously running `(test 0 (p))`. Using normal order evaluation, because y is not utilized on the `test` function, the `if` clause will be executed and resolve to 0.

Exercise 1.6

The new if does not work in the `sqrt-iter` function, it throws a *stack overflow* type error.

This is because the special form `if` runs in applicative order, thus evaluating the predicate and only running `then` or `else` when needed. In the case of `new-if`, because of the recursive call, it will be stuck evaluating that.

Exercise 1.7

Trying out the newton method, on very low numbers (0.0001) returns not very accurate results, compared to an actual square root method, comparing it with the common lisp `sqrt`:

- `(sqrt 0.0001)` $\to 0.01$
- `(newton-sqrt 0.0001)` $\to 0.032308448$

Now, with large numbers, what happens is that the number of operations exponentially increases and gets stuck evaluating. So, if we were to try and fix the first issue with smaller numbers, making our `good-enough?` function use a lower boundary, we would eventually reach the second problem, getting stuck in recursion.

Implementing the new `not-better?` function:

```
(defun not-better? (guess prev-guess)
  (< (abs (/ (- guess prev-guess) guess)) 0.0000000001))
```

And changing `sqrt-iter` accordingly:

```
(defun sqrt-iter (guess x)
  (if (not-better? (improve guess x) guess)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

Our results err much less *relative* to the values, thus fixing our problems with disproportionately large and small numbers

Exercise 1.8

To change this we reimplement the functions, which are very similar. The only notable change is the new improve function:

```
(defun improve-cube (guess x)
  (/ (+ (/ x (square guess)) (* 2 guess)) 3))
```

The rest of the `cube-iter` function is identical to sqrt-iter. See code.

Exercise 1.9

The first implementation follows a recursive structure:

```
(+ 4 5)
(inc (+ 3 5))
(inc (inc (+ 2 5)))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

Second implementation is an iterative process:

```
(+ 4 5)
(+ 3 6)
(+ 2 7)
(+ 1 8)
(+ 0 9)
9
```

Exercise 1.10

- `(A 1 10)` $\longrightarrow 1024$
- `(A 2 4)` $\longrightarrow 65536$
- `(A 3 3)` $\longrightarrow 65536$

- `(f n)` $\longrightarrow 2 * n$
- `(g n)` $\longrightarrow 2^n$
- `(h n)` $\longrightarrow 2^{h(n-1)}$

Exercise 1.11

Recursive version:

```
(defun f-1.11-rec (n)
  (if (< n 3)
      n
      (+
       (f-1.11-rec (- n 1))
       (* 2 (f-1.11-rec (- n 2)))
       (* 3 (f-1.11-rec (- n 3))))))
```

Iterative version:

```
(defun f-1.11 (n)
  (f-1.11-iter 2 1 0 n))

(defun f-1.11-iter (a b c count)
  (if (= count 0)
      c
      (f-1.11-iter (+ a (* 2 b) (* 3 c)) a b (- count 1))))
```

Exercise 1.12