SICP Exercises

This is my solution to most of the exercises on the sicp book. The main difference with the book itself is that the book proposes using scheme and I solved them using common lisp, which has some different keywords and uses.

I will try to keep the code closer to scheme, keeping with the spirit of the chapter or exercise.

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

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)` $\rightarrow 1024$
- `(A 2 4)` $\rightarrow 65536$
- `(A 3 3)` $\rightarrow 65536$

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

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

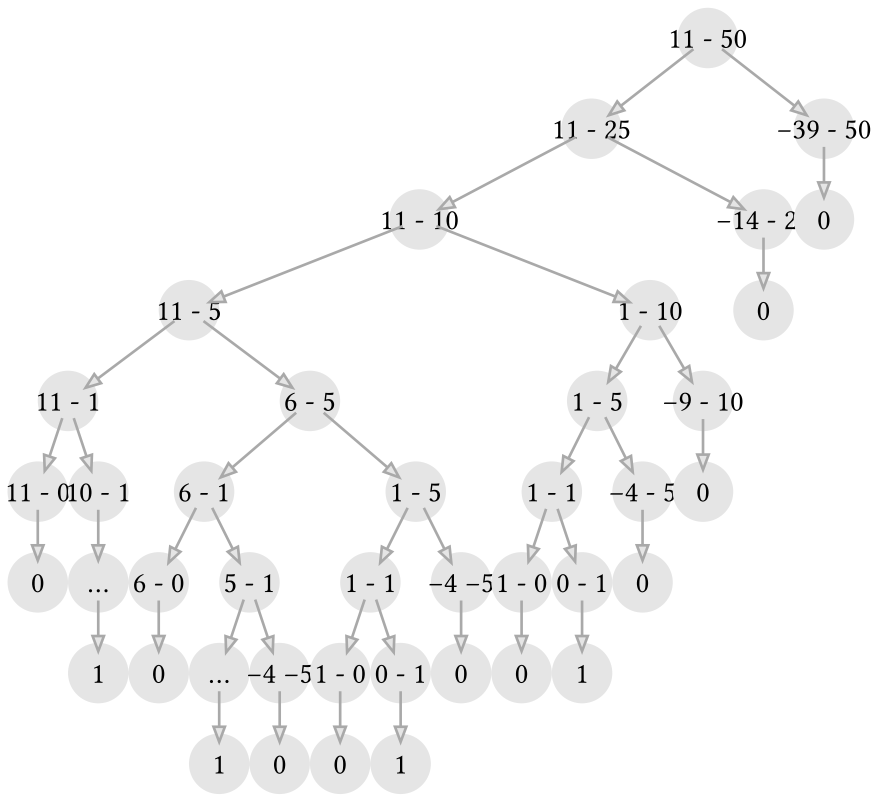
I separated the code into rows and number calculation but the execution is still recursive.

```
(defun pasc-num (row col)
  (cond ((< row 1) 1)
        ((or (<= col 1) (>= col row)) 1)
        (T (+ (pasc-num (- row 1) (- col 1)) (pasc-num (- row 1) col)))))

(defun pasc-row (row)
  (loop for item from 1 to row do
           (write (pasc-num row item))
        (write-char #\Space))
  (write-char #\Newline))

(defun pasc (row)
  (loop for i from 1 to row do
        (pasc-row i)))
```

Exercise 1.14



The amount of operations grows by one and then doubles every time a threshold is hit for every threshold.

Simplifying in big O notation, we can evaluate per number of types of coins.

For one type of coin we can see that it gets one deeper for every extra one value in the amount, for example $5 - 1$ takes 5 steps to evaluate. In big O notation:

$\Theta(n)$

For two types of coins, we add more depth every time we hit the threshold.

$\frac{n}{5} + 1 + \sum_{i=0}^{\frac{n}{5}} T(n - 5i, 1)$

Which means the second tree gets a depth of $\frac{n}{5}$ and another smaller tree per time it has hit the 5 coin threshold. For example in the 11 coins, you would get your normal depth *n* tree for the 11 one cent coins, a tree for two 5 cent coins and one 1 cent coin, and another for one 5 cent and six 1 cent.

Simplified and reduced for big O notation:

$\Theta(n^2)$

We could keep proving this upwards but with 5 types of coins we end up with:

$\Theta(n^5)$

Which grows very big when *n* is bigger.

Exercise 1.15

a. How many times is p applied when calling `(sine 12.15)`

   Appling trace to p, we can see:

```
|CL-USER> (trace p)
|(P)
|CL-USER> (sine 12.15)
|  0: (P 0.049999997)
|  0: P returned 0.1495
|  0: (P 0.1495)
|  0: P returned 0.43513453
|  0: (P 0.43513453)
|  0: P returned 0.9758465
|  0: (P 0.9758465)
|  0: P returned -0.7895632
|  0: (P -0.7895632)
```

   A total of 5 times (once per execution of `sine`)

b. The function grows in $O(\log(x))$

Exercise 1.16