Chapter 1: Building Abstractions with Procedures

Exercises for SICP chapter 1.

Exercise 1.1

```
10
12
8
3
6
a = 3
b = 4
19
nil
4
16
6
16
```

Exercise 1.2

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

Exercise 1.3

```
(define (sum-of-squares a b)
  (+ (* a a) (* b b)))

(define (ex1_3 a b c)
  (cond ((and (< a b) (< a c)) (sum-of-squares b c))
        ((and (< b a) (< b c)) (sum-of-squares a c))
        ((and (< c a) (< c b)) (sum-of-squares a b))))
```

Exercise 1.4

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

This procedure returns the sum of `a` and `b` if `b` is over zero, if its under zero it substracts them.

Exercise 1.5

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
    0
    y))

(test 0 (p))
```

In an interpreter with applicative-order evaluation, Ben will observe the following substitution/evaluation behaviour:

```
(test 0 (p))

(test 0 (p))

(test 0 (p))
```

```
(test 0 (p))
```

While in an interpreter with normal-order evaluation, Ben will observe the following substitution/ evaluation behaviour:

```
(test 0 (p))

(if (= 0 0)
  0
  (p))

(if t
  0
  (p))

0
```

Exercise 1.6

If is a special form because it needs to not evaluate the `then-clause` and `else-clause` until we determine wether the predicate is true or false. If we use this `new-if` procedure, it will be stuck evaluating forever due to the recursive call used in the `else-clause`:

```
(sqrt-iter
  (improve guess x)
  x)
```

Exercise 1.7

The `good-enough?` procedure is ineffective with small numbers because when they are close to 0.001 or less, `good-enough?` will return `true` even if its not even close.

```
> (sqrt 0.0001)
0.03230844833048122

> (sqrt 0.000000001)
0.03125001065624928
```

On the other hand, with very big numbers, the algorithm requires a precission too high (relative to the square root we are trying to find) so it takes a lot of operations to get to the results.

```
> (sqrt 10000000)
This takes too long...
```

This is the implementation of the new way of computing square roots:

```
(define (square x)
  (* x x))

(define (average x y)
  (/ (+ x y) 2))

(define (improve guess x)
  (average guess (/ x guess)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (good-enough2? new-guess old-guess)
```

```scheme
  (< (/ (abs (- new-guess old-guess)) new-guess) 0.01))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))

(define (sqrt-iter2 guess old-guess x)
  (if (good-enough2? guess old-guess)
      guess
      (sqrt-iter2 (improve guess x)
                  guess
                  x)))

(define (sqrt x)
  (sqrt-iter 1 x))

(define (sqrt2 x)
  (sqrt-iter2 2 1 x))
```

```
> (* (sqrt2 10000000) (sqrt2 10000000))
10000000.XXX
```

Exercise 1.8

Implementation code:

```scheme
(define (square x)
  (* x x))

(define (cube x)
  (* x x x))

(define (average x y)
  (/ (+ x y) 2))

(define (improve guess x)
  (/ (+ (/ x (square guess))
        (* 2 guess))
     3))

(define (good-enough? guess x)
  (< (abs (- (cube guess) x)) 0.001))

(define (cube-root-iter guess x)
  (if (good-enough? guess x)
      guess
      (cube-root-iter (improve guess x)
                      x)))

(define (cube-root x)
  (cube-root-iter 1 x))
```

Proof of the results:

```
> (cube (cube-root 50))
50.XXX
```

Exercise 1.9

The procedure:

```
(define (+ a b)
  (if (= a 0)
    b
    (inc (+ (dec a) b))))
```

Would be expanded like so:

```
(+ 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
```

So this is a recursive procedure that implements a recursive process.

The second procedure:

```
(define (+ a b)
  (if (= a 0)
    b
    (+ (dec a) (inc b))))
```

Would be expanded like so:

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

Exercise 1.10

The results are:

```
1024
65536
65536
```

The definitions would be:

$$f(n) = 2n$$

$$g(n) = 2^n$$

$$h(n) = 2^{h(n-1)}$$

$$k(n) = 5n^2$$

Exercise 1.11

```scheme
(define (sum-mul a b c)
  (+ (* 1 a)
     (* 2 b)
     (* 3 c)))

(define (recursive-f n)
  (if (< n 3) n
      (sum-mul (recursive-f (- n 1))
               (recursive-f (- n 2))
               (recursive-f (- n 3)))))

(define (iterative-f n)
  (define (aux n i a b c)
    (cond ((< n 3) n)
          ((= n i) (sum-mul a b c))
          (else (aux n (+ i 1) (sum-mul a b c) a b))))
  (aux n 3 2 1 0))
```

Exercise 1.12

Function definitions:

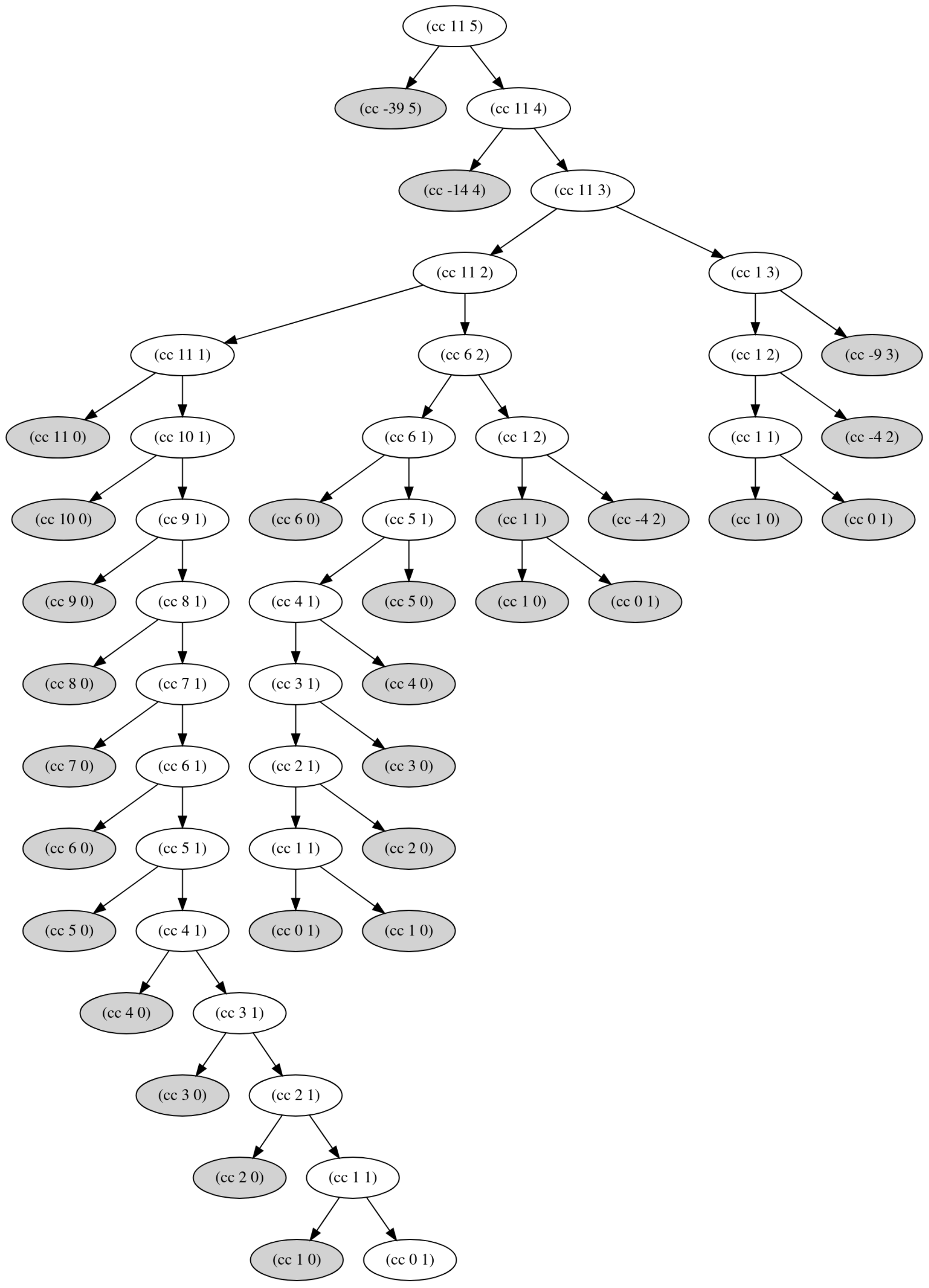
```scheme
(define (pascal-next l)
  "Returns the next list in the pascal triangle"
  (define (pascal-next-iter l f)
    (if (= 1 (length l))
        l
        (append
         (if (= f 0)
             (list (car l) (+ (car l) (cadr l)))
             (list (+ (car l) (cadr l))))
         (pascal-next-iter (cdr l) 1))))
  (pascal-next-iter l 0))

(define (pascal-list n)
  (cond ((= n 2)
         (display '(1))
         (newline)
         (display '(1 1))
         (newline)
         '(1 1))
        (else (let ((res (pascal-next (pascal-list (- n 1)))))
                (display res)
                (newline)
                res))))
```

Results:

```
> (pascal-list 20)
(1)
(1 1)
(1 2 1)
(1 3 3 1)
(1 4 6 4 1)
(1 5 10 10 5 1)
(1 6 15 20 15 6 1)
...
```

1.14

The process tree is the following:



The ammount of space is $\Theta(n)$ because in the worse case (changing with only one type of coin) we will get a tree of depth n and this is a recursive procedure.

1.15

The number of times this procedure is applied is as many as it takes to make its value < 0.1 by dividing it by 3. And then one more. More concisely:

$n = \left\lceil \log_3\left(\frac{0.1}{n}\right) \right\rceil + 1$

The space taken and the operations made by this procedure are $\log(n)$.