Contents

1 Chapter 1 Exercises

1.1 Exercise 1.1

1. 10
2. 12
3. 8
4. 3
5. 6
6. 3

7. 4
8. 19
9. false
10. 4
11. 16
12. 6
13. 16

1.2 Exercise 1.2

Refer to `code`

1.3 Exercise 1.3

Refer to `code`

1.4 Exercise 1.4

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

The procedure is used to add `a` and `b` together. The purpose of the `if` condition is to accomodate for negative values of `b` - effectively working with the absolute value of `b`.

For instance, if `b = -8` and `a = 9`, the predicate of the `if` condition will evaluate to `true` and so the resulting operator will be `-`. Therefore, the evaluated expression will be `(- 9 (- 8))`. When expanded to normal form `(9 - (-8) = (17)`.

1.5 Exercise 1.5

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

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

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

With applicative-order evaluation, the expression is first evaluated then each argument is applied - meaning that the expressions are evaluated as they appear, rather than waiting till the very end. Therefore, the expression will attempt to evaluate `p`, which in this case if a function call. Since `p` is recursive, the interpreter will be stuck in an infinite loop.

However, with normal-order evaluation, the expression is expanded first and is only evaluated when needed - meaning that the interpreter will not attempt to evaluate `(p)` before fully expanding the procedure definition of `test`. This leads to the `if` condition being evaluated first and having the expression return `0` instead of being stuck in an infinite loop.

```
; Applicative-order evaluation
(test 0 (p))
(test 0 (p))
...
(test 0 (p))

; Normal-order evaluation
(test 0 (p))
(if (= 0 0)
    0
    (p))
>>> 0
```

1.6 Exercise 1.6

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))

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

To understand how this new function will compute the square roots, we need to first see how the function will be evaluated. For this, we apply the **applicative-order evaluation**, the same one that lisp uses.

When we run the code in our terminal, it doesn't return anything and is instead stuck processing it. So let's investigate why. Unlike the built in if statement, `new-if` is a procedure defined by the developer. This means that when evaluating the expression, we first evaluate the arguments of `new-if` before determining what `new-if` does, and this causes it to hang because we're never actually comparing the arguments of `new-if` since the procedure will continue to recurse.

1.7 Exercise 1.7

For small numbers, our limit is too large to allow for an accurate reading. If the guesses reach a certain limit that exceeds the built in 0.001 limit, we will get false positives that are not accurate enough enough.

For large numbers, our limit is far too small for the system to appropriately measure the square root within a decent period of time since it will continue to refine the square root till it hits the 0.001 limit.

The solution would be to modify `good-enough?` to look at the difference between iterations.

```
; Old version
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

; New version
(define (good-enough? guess x)
  (< (abs (- (improve guess x) guess))
     (* guess 0.001)))
```

In the old version, we compare the original number to the square of the guess. However, this is too strict of a requirement for the guesses to be accurate. The new version rectifies this issue by fatoring in two key components.

1. The size of the leeway or limit
2. How much of a fit the guess was

This way, we are more flexible with the way we determine the limit for what qualifies as a `good-enough?` guess.

1.8 Exercise 1.8

Refer to `code`

1.9 Exercise 1.9

```
(define (inc x)
  (+ x 1))

(define (dec x)
  (- x 1))

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

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

For the first implementation, the growth of + will look like (note that we omit the calculation of `dec` and take that the numbers decrease automatically):

```
(+ 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
```

For this, we see that the first + is a recursive process described by a recursive procedure.

The growth of the second implementation of + looks like:

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

For this, we see that the second + is an iterative process described by a recursive procedure.

1.10 Exercise 1.10

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1) (A x (- y 1))))))
```

```
   (A 1 10)
```

```
(A 1 10)
(A 0 (A 1 9))
(A 0 (A 0 (A 1 8)))
(A 0 (A 0 (A 0 (A 1 7))))
(A 0 (A 0 (A 0 (A 0 (A 1 6)))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 1 5))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 4)))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 3))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 2)))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 1))))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 2)))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 4)))))))) 
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 8)))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 16))))))
(A 0 (A 0 (A 0 (A 0 (A 0 32)))))
(A 0 (A 0  (A 0 (A 0 64))))
(A 0 (A 0 (A 0 128)))
(A 0 (A 0 256))
(A 0 512)
(1024)
```

```
   (A 2 4)
```

```
(A 2 4)
(A 1 (A 2 3))
(A 1 (A 1 (A 2 2)))
(A 1 (A 1 (A 1 (A 2 1))))
(A 1 (A 1 (A 1 2)))
(A 1 (A 1 (A 0 (A 1 1))))
(A 1 (A 1 (A 0 2)))
(A 1 (A 1 4))
(A 1 (A 0 (A 1 3)))
```

```
(A 1 (A 0 (A 0 (A 1 2))))
(A 1 (A 0 (A 0 (A 0 (A 1 1)))))
(A 1 (A 0 (A 0 (A 0 2))))
(A 1 (A 0 (A 0 4)))
(A 1 (A 0 8))
(A 1 16)
...
(65536)
```

```
   (A 3 3)
```

```
(A 3 3)
(A 2 (A 3 2))
(A 2 (A 2 (A 3 1)))
(A 2 (A 2 2))
(A 2 (A 1 (A 2 1)))
(A 2 (A 1 2))
(A 2 (A 0 (A 1 1)))
(A 2 (A 0 2))
(A 2 4)
...
(65536)
```

The following procedures are associated to the following mathematical definitions.

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
```

`f` is the same as n^2 `g` is the same as 2^n `h` is the same as 2^2 for `n - 1` times

1.11 Exercise 1.11

$$f(n) = \begin{cases} n & \text{if } n < 3, \\ f(n-1) + 2f(n-2) + 3f(n-3) & \text{if } n \geq 3. \end{cases}$$

Refer to `code`

To begin to understand how this pattern works, we start off by listing the first *6* `n` values for the procedure.

n	f(n)
1	1
2	2
3	4
4	11
5	25
6	59

1.12 Exercise 1.12

Refer to `code`

1.13 Exercise 1.13

1.13.1 TODO Complete this when analysing algorithms is completed

$$\text{Fib(n)} \approx \frac{\phi^n}{\sqrt{5}} \tag{1}$$

1.14 Exercise 1.14

Expand this on your own, it's not that hard

1.15 Exercise 1.15

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

a. 5

```
(sine 12.15)
(p (sine 4.05))
(p (p (sine 1.35)))
(p (p (p (sine 0.45))))
(p (p (p (p (sine 0.15)))))
(p (p (p (p (p (sine 0.05))))))
(p (p (p (p (p 0.05))))
   ...
```

b. Explanation found here and additional explanations can be found here
Order of growth for space and time are equal at Θ(log a)

- As the recursion continues, the value of `a` decreases, so the number of steps vary logarithmically with `a`

1.16 Exercise 1.16

Refer to `code`

1.17 Exercise 1.17

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

Refer to `code`

1.18 Exercise 1.18

Refer to `code`

1.19 Exercise 1.19

To solve this problem, we will use vectors
Explanation can be found here.

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   ...
                   ...
                   (/count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.20 Exercise 1.20

Iterative GCD procedure:

```
(define (gcd a b)
  (if (= b 0)
      a
    (gcd b (remainder a b))))
```

Using normal-order substitution for `(gcd 206 40)` - which should also include the expansion of the `if` statements.

```
(gcd 206 40) ;; 0
(gcd 40
     (remainder 206 40)) ;; 1 = 0 + 1
(gcd (remainder 206 40)
     (remainder 40 (remainder 206 40))) ;; 3 = 0 + 1 + 2
(gcd (remainder 40 (remainder 206 40))
     (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))) ;; 6 = 0 + 1 + 2
(gcd (remainder (remainder 206 40) (remainder 40 (remainder 206 40)))
     (remainder (remainder 40 (remainder 206 40)) (remainder (remainder 206 40) (remain
(2)
```

Using applicative-order substitution for `(gcd 206 40)`

```
(gcd 206 40)
(gcd 40 (remainder 206 40))
(gcd 40 6)
(gcd 6 (remainder 40 6))
(gcd 6 4)
(gcd 4 (remainder 6 4))
(gcd 4 2)
(gcd 2 (remainder 4 2))
(gcd 2 0)
(2)
```

When using the normal-order evaluation, a total of 11 `remainder` calls are made. While using the applicative-order evaluation, a total of 4 `remainder` calls are made. However, if we count the `if` statements used for the normal-order evaluation, a total of 18 `remainder` calls are made.

The number of remainder calls are in Fibonacci sequence. $R = \sum_{i=1}^{n} fib(n)$