**Structures and Interpretation of Computer Program**

**Exercise Chapter 1.1 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 1.1**

* 10
* 12
* 8
* 3
* 6
* Value: a
* Value: b
* 19
* #f
* 4
* 16
* 6
* 16

**Exercise 1.2**

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

**Exercise 1.3**

(define (sum-largest-two a b c )  
 (cond ( (and ( < a b ) ( < a c )) (+ (square b) (square c)))   
 ( (and ( < b a ) ( < b c )) (+ (square a) (square c)))   
 ( (and ( < c a ) ( < c b )) (+ (square a) (square b)))))

(define (square x) (\* x x ) )

**Exercise 1.4**

The 'if' statement determine the operator of the next expression. If b is higher than 0, then a+b, otherwise, a-b. If b is positive, then just add the number. Otherwise, a-b, but b is guaranteed to be less than zero, which turns into a - (-b). (a + |b|)

**Exercise 1.5**

Interpreter that uses normal-order evaluation will evaluate to 0 as the procedure (test) will be evaluated first. And as x == 0, it will return 0. But interpreter that uses applicative-order evaluation will first expand p. As p is defined by itself, the interpreter will go into endless loop of (((p))).

**Exercise 1.6**

The program will undergo infinite recursion because the new-if will use applicative-order as it is a procedure.

To explain, applicative-order is a substitution model that will first evaluate the argument before applying the procedure. This new-if will evaluate **its argument first** before applying new-if. As a result, it will goes to infinite recursion because sqrt-iter will always be evaluated **first** regardless whether or not the (new if) statement is true. If form is special procedure to avoid this issue. The normal (if) will first check the condition and jump to the correct body. In short:

* New if:
  + Evaluate inner argument of new-if
  + Infinite loops because it tried to evaluate sqrt-iter every time
* If:
  + Evaluate the first condition.
  + If the first condition fails, check next condition and so on..
  + If one of the condition is true, apply its arguments.

**Exercise 1.7**

1. Explain how square root procedure failed for:
   1. Very small number

The square root procedure will fail because procedure of improve guess x improves the guess in very small iteration so that it will not satisfy good-enough procedure. For example, using sqrt(0.2) The guess procedure improve the guess from 1.0 to 0.6 to 0.467 to 0.447 to 0.457 to 0.447. The calculation of sqrt of 0.2 will not finish as the number will go forth and back between .457 to .447. This is because in improve guess x, dividing guess by 0.2 means that the procedure will improve guess very little and it will be hard to converge to a single number that satisfy tolerance 0.001.

* 1. Very large number

Since computer have limited precision, the sqrt algorithm will not converge as it will not reach the tolerance level required by good-enough procedure of 0.001. For example, using sqrt(900000), the procedure will go on to 948.683 and continue to 948.6833 and it will never reach a number that can satisfy tolerance of 0.001.

1. Design a procedure to fix this issue

To solve this issue, find small enough difference between previous guess and new guess instead of checking whether the guess satisfy certain tolerance value. In this way, when the algorithm detect that the next guess does not change that much to previous guess, it stops and return the final value.

(define (good-enough? guess oldguess)  
 (< ( abs ( - guess oldguess ) ) 0.001))

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

( define (sqrt x )  
 ( sqrt-iter 1.0 2.0 x ))

It works equally well enough for big and small number after tested using sqrt(0.2) and sqrt(900000) again.

**Exercise 1.8**

Design cube-root procedure.

(define (cube-root x)

(cube-root-iter 1.0 x))

(define (cube-root-iter guess x)

(if (good-enough? guess x)

guess

(cube-root-iter (improve guess x) x)))

(define (improve guess x)

(/ ( + ( / x ( sqr guess) ) ( \* 2 guess) ) 3 ))

(define (good-enough? guess x)

( < (abs (- ( cube guess ) x)) 0.001 ))

(define (good-enough? guess x)

( = (improve guess x ) guess ) )

(define (sqr x) (\* x x))

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

(display (cube-root 27))

Example: cube (9) -> 4 -> 2.8542 -> 2.269 -> 2.0953 -> 2.0802 -> 2.0801 (satisfy tolerance)