SECTION 1

RECURSION

in recursion we just call the function in on itself till a condition is met, this results in the program firstly processing *ALL* the data and then computing one by one

```
(define (factorial n)
    (if (= n 1)
        1
        (* n (factorial (- n 1)))))

;(factorial 6)
;(* 6 (factorial 5))
;(* 6 (* 5 (factorial 4)))
;(* 6 (* 5 (* 4 (factorial 3))))
;(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
;(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))  <-- accumulates all the processes
;(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
;(* 6 (* 5 (* 4 (* 3 2))))
;(* 6 (* 5 (* 4 6)))
;(* 6 (* 5 24))
;(* 6 120)
;720
```

ITERATION

Iteration means going forward as opposed to the recursion's going backwards at first, to the very lowest level, and then going forward calculating the result on the way back up, you *change* the value of the variable at every step:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))

;(fact 4)
;(iter 1  1  4)
;(iter 1 2 4)
;(iter 2 3 4)
;(iter 6 4 4)
;(iter 24 5 4)
;24
```

Product	Counter	Input
1	1	4
1	2	4
2	3	4
6	4	4
24	5	4

Exercise 1

Exercise 1.1

```
10
;10
(+ 5 3 4)
;12
(- 9 1)
;8
(/ 6 2)
;3
(+ (* 2 4) (- 4 6))
;6

(define a 3)
(define b (+ a 1))
(+ a b (* a b))
;(+ 3 (+ 3 1) (* 3 (+ 3 1)))
;(+ 3 4 12)
;(19)

(= a b)
;FALSE (amogus)
(if (and (> b a) (< b (* a b)))
b
a)
;(if (and true true) b a)
;(if true b a)
;(b)
;4

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
;(+ 6 7 3)
;16

(+ 2 (if (> b a) b a))
;(+ 2 (if true b a))
;(+ 2 4)
;(+ 6)

(* (cond ((> a b) a)
     ((< a b) b)
     (else -1))
   (+ a 1))
;(* 4 (+ 3 1))
;(* 4 4)
;16
```

Exercise 2

Translate the following expression into prefix

$$\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}.$$

```
(/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))
; -37/150
```

Exercise 3

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers

```
(define (f1 a b c)
  (if (< a b)
      (if (< a c)
          (+ (* b b) (* c c))
          (+ (* b b) (* a a)))
      (if (< b c)
          (+ (* a a) (* c c))
          (+ (* a a) (* b b)))))
```

Exercise 4

Observe that our model of evaluation allows for combinations whose operators are compound expres- sions. Use this observation to describe the behavior of the following procedure:

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

```
;this is just a+|b| by performing a+b if b>0, else a-b
(a-plus-abs-b 3 4)
;(7)
(a-plus-abs-b 3 -4)
```

Exercise 5

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

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

Then he evaluates the expression `(test 0 (p))` What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form if is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.

evaluating (p) will result in an infinite loop for applicative order because internal reduction happens **BEFORE** the values are evaluated, like:

```
(test 0 (p))
(if (= 0 0) 0 (p)) ;<---- here the program will go for an inifnite loop
because it can't evaluate `(p)`
```

for normal order eval this will happen:

```
(test 0 (p))
(if (= 0 0)
    0       ;<--- here the program will stop
    (p))    ;<--- the program will actually never reach this phase
```

Exercise 6

Alyssa P. Hacker doesn't see why if needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of cond?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of if:

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

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

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

What happens when Alyssa attempts to use this to compute square roots? Explain.

in the old `(if cond A B)` the B condition would only be evaluated if cond was false BUT in the new `(new-if cond A B)` as per the definition the arguement B needs to be evaluated regardless of if the cond was false because when a function is called the first thing that Scheme does with the argument list is evaluating every single arguement

do you remembah? the scheme uses applicative-order for function in septembah?

Exercise 7

The good-enough? test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing good-enough? is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

if y is our guess for $\sqrt{x}$ then, $y = \sqrt{x} + \delta y$ and $|y^2 - x| > \varepsilon$, then for small values:

$$\varepsilon = |y^2 - x|$$

$$\varepsilon = |(\sqrt{x} + \delta y)^2 - x|$$
$$\varepsilon = |(x + 2\delta y)\sqrt{x} - x|$$
$$\varepsilon = 2|\delta y|\sqrt{x}$$

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

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

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

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

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

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

; Given the last and 2nd last guesses, check if the fractional change is good
enough.
; I'm choosing the formula |guessL-guessLL|/guessL < 0.001
(define (good-enough2? guessL guessLL)
(< (/ (abs (- guessL guessLL)) guessL) 0.01))

(define (sqrt-iter2 guessL guessLL x)
(if (good-enough2? guessL guessLL)
guessL
(sqrt-iter2 (improve guessL x) guessL x)))

(define (my-sqrt2 x)
(sqrt-iter2 1.0 1.1 x))
```

Exercise 8

Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value

$$\frac{\frac{x}{y^2}+2y}{3}$$

Use this formula to implement a cube-root procedure analogous to the square-root procedure.

```
(define (improve y x)
  (/ (+ (/ x (* y y)) (* 2 y)) 3))

; Fractional good-enough like in ex 1-7
(define (good-enough? guessL guessLL)
  (< (/ (abs (- guessL guessLL)) guessL) 0.01))

(define (curt-iter guessL guessLL x)
  (if (good-enough? guessL guessLL)
      guessL
      (curt-iter (improve guessL x) guessL x)))

(define (my-curt x)
  (curt-iter 1.0 1.1 x))

; 1.4422495703074083
(my-curt 3)
```

Exercise 9

Each of the following two procedures defines a method for adding two positive integers in terms of the procedures `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

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

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

Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

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

```
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
;(+ 4 5)
;(+ 3 6)
;(+ 2 7)
;(+ 1 8)
;(+ 0 9)
;9
```

Exercise 10

The following procedure computes a mathematical function called Ackermann's function.

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

What are the values of the following expressions?

```
(A 1 10)
(A 2 4)
(A 3 3)
```

Consider the following procedures, where A is the procedure defined above:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures f, g, and h for positive integer values of n. For example, `(k n)` computes $5n^2$.

```
(A 1 10)
;1024
(A 2 4)
;65536
(A 3 3)
;65536

(define (f n) (A 0 n))
;this is just x 0 always so it is just a function that doubles the arguement
```

```
i.e n*2
(define (g n) (A 1 n))
;(A 1 0) is 0
;(A 1 1) is 2
;(A 1 2) is (* 2 2) = 4
;(A 1 3) is (* 2 (* 2 2)) = 8
;(A 1 n) is 2^n
(define (h n) (A 2 n))
;(A 2 0) is 0
;(A 2 1) is 2
;(A 2 2) is 4
;(A 2 3) is 16
;this is (A 1 (A 2 (- n 1)))
;or square of (A 2 (- n 1))
;so this is just square of square of square of square and so on
```

Exercise 11

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$. Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

recursive is pretty easy

```
(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1)) (* (f (- n 2)) 2) (* (f (- n 3)) 3))))
```

for iterative. We need three pieces of state to track the current, the last and the values of f. (that is, f(n-1), f(n-2) and f(n-3).)

```
(define (f n)
  (f-it-helper 2 1 0 n))

(define (iterator a b c counter)
  (if (= counter 0)
      c
      (iterator (+ a (* 2 b) (* 3 c))
                a
                b
                (- counter 1))))
;think like this
;f(0) = 0  |
;f(1) = 1  | all known values
;f(2) = 2  |

;f(3) = f(2) + 2f(1) + 3f(0)
```

```
;f(4) = f(3) + 2f(2) + 3f(1)
;f(5) = f(4) + 2f(3) + 3f(2)
;f(6) = f(5) + 2f(4) + 3f(3)
```

Exercise 12

Write a recursive procedure that computes elements of Pascal's triangle.

```
TODO
```

Exercise 13

Prove that $\text{fib}(n)$ is the closest integer to $\frac{\varphi^n}{\sqrt{5}}$ where $\varphi = \frac{1+\sqrt{5}}{2}$.

Hint: Let $\psi = \frac{1\sqrt{5}}{2}$. Use induction and the definition of the Fibonacci numbers to prove that $\text{Fib}(n) = \frac{\varphi^n - \psi^n}{\texttt{sqrt}\{5\}}$.

TODO