Exercise 1. Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
Scheme] 10

   10

Scheme] (+ 5 3 4)

   12

Scheme] (- 9 1)

   8

Scheme] (/ 6 2)

   3

Scheme] (+ (* 2 4) (- 4 6))

   6

Scheme] (define a 3)

   3

Scheme] (define b (+ a 1))

   4

Scheme] (+ a b (* a b))

   19

Scheme] (= a b)

   #f

Scheme] (if (and (> b a) (< b (* a b)))
            b
            a)

   4

Scheme] (cond ((= a 4) 6)
              ((= b 4) (+ 6 7 a))
              (else 25))

   16

Scheme] (+ 2 (if (> b a) b a))

   6

Scheme] (* (cond ((> a b) a)
                 ((< a b) b)
                 (else -1)))

   4
```

Exercise 2. Translate the following expression into prefix form: $\frac{5+4+\left(2-\left(3-\left(6+\frac{4}{5}\right)\right)\right)}{3(6-2)(2-7)}$.

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

   -37/150
```

```
Scheme]
```

Exercise 3. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

Solution.

```
Scheme] (define (ex3-sum_of_squares x y)
          (+ (* x x) (* y y)))

   ex3-sum_of_squares

Scheme] (define (ex3-larger x y z)
          (cond ((and (> y x) (> z x)) (values y z))
                ((and (> x y) (> z y)) (values x z))
                ((and (> x z) (> y z)) (values x y))))

   ex3-larger

Scheme] (ex3-sum_of_squares (ex3-larger 1 2 3))

   13

Scheme] (ex3-sum_of_squares (ex3-larger 2 1 3))

   13

Scheme] (ex3-sum_of_squares (ex3-larger 2 3 1))

   13
```

Exercise 4. Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

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

Solution.

Simplify (operator a b)

Operator is a subexpression, evaluating the value of left most subexpression (the operator).

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.)

Solution.

- applicative-order
    - evaluation: infinite recursion
    - When calling `(test 0 (p))`, it evaluates `(p)`, which leads to infinite recursion
- normal-order
    - evaluation: 0
    - When calling `(test 0 (p))`, expand to

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

      Obviously, 0=0, the ⟨predicate⟩ of `if` evaluates to a `#t` value. `if` statements do not evaluate `(p)`

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.

Solution.

- `if` version

```
Scheme] (define (ex6-square x) (* x x))

   ex6-square

Scheme] (define (ex6-average x y)
          (/ (+ x y) 2))

   ex6-average

Scheme] (define (ex6-improve guess x)
          (ex6-average guess (/ x guess)))

   ex6-improve

Scheme] (define (ex6-good-enough? guess x)
          (< (abs (- (ex6-square guess) x)) 0.001))

   ex6-good-enough?

Scheme] (define (ex6-sqrt-iter guess x)
          (if (ex6-good-enough? guess x)
              guess
              (ex6-sqrt-iter (ex6-improve guess x)
                             x)))

   ex6-sqrt-iter

Scheme] (define (ex6-sqrt x)
          (ex6-sqrt-iter 1.0 x))

   ex6-sqrt

Scheme] (ex6-sqrt 9)

   3.00009155413138
```

- `new-if` version

```
Scheme] (define (ex6-new-if predicate then-clause else-clause)
          (cond (predicate then-clause)
                (else else-clause)))

   ex6-new-if

Scheme] (define (ex6-new-sqrt-iter guess x)
          (ex6-new-if (ex6-good-enough? guess x)
                  guess
                  (ex6-new-sqrt-iter (ex6-improve guess x)
                                 x)))

   ex6-new-sqrt-iter

Scheme] (define (ex6-new-sqrt x)
          (ex6-new-sqrt-iter 1.0 x))

   ex6-new-sqrt

Scheme] ;(ex6-new-sqrt 9)

   #<eof>
```

- Conclusion
    - `new-if` is a procedure that evaluates each operand first when called.

- When evaluating `else-clause`, it will fall into infinite recursion

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?

Solution.

- Showing how the test fails for small and large numbers.

```
Scheme] (ex6-square (ex6-sqrt 1.123456789e-12))

   9.765625007482396e-4

Scheme] (ex6-square (sqrt 1.123456789e-12))

   1.1234567889999999e-12

Scheme] ;(ex6-square (ex6-sqrt 9.98765432198e+12))
        ; Busy
        ; Can't compute

   #<eof>

Scheme] (ex6-square (sqrt 9.98765432198e+12))

   9987654321980.0
```

- Improved `good-enough?`

```
Scheme] (define (ex7-square x) (* x x))

   ex7-square

Scheme] (define (ex7-average x y) (/ (+ x y) 2))

   ex7-average

Scheme] (define (ex7-improve guess x)
          (ex7-average guess (/ x guess)))

   ex7-improve

Scheme] (define (ex7-good-enough? pre-guess guess)
          (< (/ (abs (- pre-guess guess)) guess) 0.001))

   ex7-good-enough?

Scheme] (define (ex7-sqrt-iter pre-guess guess x)
          (if (ex7-good-enough? pre-guess guess)
              guess
              (ex7-sqrt-iter guess
                             (ex7-improve guess x)
                             x)))

   ex7-sqrt-iter
```

```
Scheme] (define (ex7-sqrt x)
          (ex7-sqrt-iter 0.0 1.0 x))

   ex7-sqrt

Scheme] (ex6-square (ex7-sqrt 1.123456789e-12))

   1.1234568740513339e-12

Scheme] (ex6-square (sqrt 1.123456789e-12))

   1.1234567889999999e-12

Scheme] (ex6-square (ex7-sqrt 9.98765432198e+12))

   9987654345933.438

Scheme] (ex6-square (sqrt 9.98765432198e+12))

   9987654321980.0
```

- Conclusion

Compare with standard library sqrt

Value	`sqrt`	`ex6-sqrt`	`ex7-sqrt`
1.123456789e-12	1.1234567889999999e-12	9.765625007482396e-4	1.1234568740513339e-12
9.98765432198e+12	9987654321980.0	Busy (Can't compute)	9987654345933.438

The gap with the standard library sqrt

```
Scheme] (abs (- 1.123456789e-12 1.1234567889999999e-12))

   0.0

Scheme] (abs (- 1.123456789e-12 9.765625007482396e-4))

   9.765624996247828e-4

Scheme] (abs (- 1.123456789e-12 1.1234568740513339e-12))

   8.50513339973876e-20

Scheme] (abs (- 9.98765432198e+12 9987654321980.0))

   0.0

Scheme] (abs (- 9.98765432198e+12 9987654345933.438))

   23953.4375
```

Value	`sqrt`	`ex6-sqrt`	`ex7-sqrt`
1.123456789e-12	0.0	9.765624996247828e-4	8.50513339973876e-20
9.98765432198e+12	0.0	FIN	23953.4375

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 $\frac{x/y^2+2y}{3}$. Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In Section 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

Solution.

```
Scheme] (define (ex8-square x) (* x x))

   ex8-square

Scheme] (define (ex8-improve x y)
          (/ (+ (/ x (ex8-square y)) (* 2 y)) 3))

   ex8-improve

Scheme] (define (ex8-good-enough? pre-guess guess)
          (< (/ (abs (- pre-guess guess)) guess) 0.001))

   ex8-good-enough?

Scheme] (define (ex8-cube-root-iter pre-guess guess x)
          (if (ex8-good-enough? pre-guess guess)
              guess
              (ex8-cube-root-iter guess
                                  (ex8-improve x guess)
                                  x)))

   ex8-cube-root-iter

Scheme] (define (ex8-cube-root x)
          (ex8-cube-root-iter 0.0 1.0 x))

   ex8-cube-root

Scheme] (ex8-cube-root 27)

   3.0000005410641766

Scheme] (ex8-cube-root 1.123456789e-12)

   1.0395661470382135e-4

Scheme] (ex8-cube-root 9.98765432198e+12)

   2.153547730824831e+4
```