To evaluate a combination whose operator names a compound procedure, the interpreter follows much the same process as for combinations whose operators name primitive procedures i.e.

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

Consider the following procedure

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

(define (sum-of-squares x y)
   (+ (square x) (square y))
   )

(define (f a)
   (sum-of-squares (+ a 1) (+ a 2))
   )

(f 2)
```

Evaluation of (f 2)

(sum-of-squares (+ 2 1) (+ 2 2))
(sum-of-squares 3 4)
(+ square(3) square(4))
(+ (* 3 3) (* 4 4))
(+ 9 16)
25

The "evaluate the arguments and then apply" method that the interpreter actually uses, which is called applicative-order evaluation. Scheme uses this method for efficiency.

This is not the exact manner in which the interpreter works, but for ease of understanding this works just fine.

(f 2)
(sum-of-squares (+ 2 1) (+ 2 2))
(+ square (+ 2 1) square (+ 2 2))
(+ (* (+2 1) (+ 2 1)) (* (+ 2 2) (+ 2 2)))
(+ (* 3 3) (* 4 4)) (+ 9 16)
25

This "fully expand and then reduce" evaluation method is known as normal-order evaluation.

This type of evaluation also has its own uses, but for now we stick to applicative order evaluation.

The expressive power of the class of procedures that we can define at this point is very limited, because we have no way to make tests and to perform different operations depending on the result of a test.

To perform case analysis, Scheme provides us with cond
The general form of a conditional expression is

(cond $(< p_1 >\ \ < e_1 >)$
$(< p_2 >\ \ < e_2 >)$
$\ldots$
$(< p_n >\ \ < e_n >)$
)

$((< p_i >\ \ < e_i >))$ are called clauses. The first part is the predicate and the second part is the consequent.

Conditional expressions are evaluated as follows.

- The predicate $< p_1 >$ is evaluated first.
- If its value is false, then $< p_2 >$ is evaluated.
- If p_2's value is also false, then $< p_3 >$ is evaluated.
- This process continues until a predicate is found whose value is true, in which case the interpreter returns the value of the corresponding consequent expression $< e >$ of the clause as the value of the conditional expression.
- If none of the $< p >$'s is found to be true, the value of the cond is undefined.

The term **Predicate** is used for procedures/expressions that return true or false.

```
> (define (abs x)
          (cond((< x 0) (- x))
               (else x)
          )
  )
```

else is a special symbol that can be used in place of the $< p >$ in the final clause of a cond.

Alternatively, one could also use if conditional, if there are only two cases to be dealt with.
if has the following structure

(*if* < *predicate* >< *consequent* >< *alternative* >)

```
(define (abs x)
  (if(< abs 0) (- x) x)
)
```

If the < *predicate* > evaluates to a true value,
the interpreter then evaluates the < *consequent* > and returns its value.
Otherwise it evaluates the < *alternative* > and returns its value.

Notice that this form of if is different from the if conditional in some other languages.
In some languages, the $<$ *alternative* $>$ is optional and need not be specified all the time.
In scheme however, the $<$ *alternative* $>$ is not optional.
However, one can easily write a version of if which does not have this restriction.

```
> (define (my-if x y)
    (cond (x y))
  )
> (my-if (< 14 10) 'a)
> (my-if (< 4 10) 'a)
a
```

Logical composition operations

($and < e_1 >< e_2 > \ldots < e_n >$).
If any $< e >$ evaluates to false, the value of the "*and*" expression is false, and the rest of the $< e >$'s are not evaluated

($or < e_1 >< e_2 > \ldots < e_n >$)
If any $< e >$ evaluates to true, the value of the "*or*" expression is true, and the rest of the $< e >$'s are not evaluated.

($not < e >$)
Returns true if $< e >$ evaluates to false and returns false otherwise.

A square root function is defined as

$$\sqrt{x} = y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

A square root function is defined as

$$\sqrt{x} = y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

The contrast between a mathematical function and procedure is a reflection of the general distinction between describing *properties of things* and describing *how to do things*. i.e declarative vs imperative knowledge.

How does one compute square roots?

How does one compute square roots?

One deterministic method is known as Newton's method of successive approximations.

If we have a guess y for the value of the square root of a number x, we can get a better guess (one closer to the actual square root) by replacing y with $\frac{1}{2}(y + \frac{x}{y})$.

How does one compute square roots?

One deterministic method is known as Newton's method of successive approximations.

If we have a guess y for the value of the square root of a number x, we can get a better guess (one closer to the actual square root) by replacing y with $\frac{1}{2}(y + \frac{x}{y})$.

It is also known that as long as the value of x is greater than 1, than the 1 can be used as an initial guess.

The underlying math is not our concern at the moment. We are assured that the math is sound and can be used to compute square roots.

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

(define (improve guess x)
 (/ (+ guess (/ x guess)) 2)
)

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

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

Note that we are not concerned about how the square or improve-guess procedure works. We have abstracted these procedures.

At this level of abstraction, any procedure that computes the square is equally good.

Someone using the procedure “square” should not need to know how the procedure is implemented in order to use it.

One detail of a procedure's implementation that should not matter to the user of the procedure is the implementer's choice of names for the procedure's formal parameters Therefore

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

should behave in exactly the same manner.

One detail of a procedure's implementation that should not matter to the user of the procedure is the implementer's choice of names for the procedure's formal parameters Therefore

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

should behave in exactly the same manner.

The meaning of a procedure should be independent of the parameter names used by its author.

This principle forces us to make sure that the parameter names of a procedure are *local* to the body of the procedure.

If the parameters were not local to the bodies of their respective procedures, then the parameter x in square could be confused with the parameter x in good-enough?, and the behavior of good-enough? would depend upon which version of square we used.

Thus, square would not be the black box we desired.

A parameter is a **formal parameter** of a procedure, if its name doesn't affect the execution of the procedure.
Such a name is called a **bound** variable, and procedure definition is to said to bind to its formal parameters.

The meaning of a procedure definition is unchanged if a bound variable is consistently renamed throughout the definition.

The set of expressions for which a binding defines a name is called the **scope** of that name.
In a procedure definition, the bound variables declared as the formal parameters of the procedure have the body of the procedure as their scope.

If a variable is not bound, we say that it is free.

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

In the definition of good-enough? procedure, guess and x are bound variables but $<, -$, abs, and square are free variables.

If a variable is not bound, we say that it is free.

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

In the definition of good-enough? procedure, guess and x are bound variables but $<, -$, abs, and square are free variables.

The meaning of good-enough? should be independent of the names we choose for guess and x so long as they are mutually distinct and different from $<, -$, abs, and square.

So you can consistently replace all instances of guess with say esti and all instances of x with say y, without affecting the procedure's execution/output.

However, the meaning of good-enough? is not independent of the names of its free variables.

One cannot freely replace the abs with some other arbitrary variable say sab unless both these procedures do the exact same thing.
In this example, $<, -$, abs, and square happen to be the free variables.

Since the user is primarily concerned with the sqrt procedure, the rest of the procedures only clutter up their minds/code.

It is desirable to localize the subprocedures, hiding them inside sqrt procedure, so that these subprocedures do not collide with other pre-existing library procedures with the same name.

To make this possible, we allow a procedure to have internal definitions that are local to that procedure.

Such nesting of definitions is called block structure.

```
(define (sqrt x)

        (define (square y)
                  (* y y)
        )

        (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 (improve guess x)
          (/ (+ guess (/ x guess)) 2)
          )

       (sqrt-iter 1.0 x)
)
```

The procedures have now become subprocedures of sqrt. These subprocedures can be re-ordered, however the last procedure is the one which returns the answer.

One can also improve this block structure by observing that x is bound in the definition of sqrt, the procedures good-enough?, improve, and sqrt-iter, which are defined internally to sqrt, are in the scope of x.

Thus, it is not necessary to pass x explicitly to each of these procedures.
This allows x to be a free variable in the internal definitions.
Variable x gets its value from the argument with which the enclosing procedure sqrt is called.
This discipline is called **lexical scoping**.

```
(define (sqrt x)

        (define (square y)
                   (* y y)
        )

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

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

        (define (improve guess)
          (/ (+ guess (/ x guess)) 2)
          )

       (sqrt-iter 1.0)
)
```

x need not be passed as a parameter to the sub-procedures as lexical scoping allows these sub-procedures to access the value of x.