Expressions are raised with a raise statement.

```scheme
<expr> must evaluate to a subclass of BaseException or an instance of one.
```

```scheme
try:
  <try suite> except <except class> as <name>
The <try suite> is executed first. If, during the course of executing the
<try suite>, an exception is raised that is not handled otherwise, and
if the class of the exception inherits from <except class>, then
the <except suite> is executed, with <name> bound to the exception.
```

The built-in Scheme list data structure can represent combinations
```
\[
\text{scm} \left( \text{list 'quotient 10 2} \right) = \text{scm} \left( \text{eval 'list 'quotient 10 2} \right)
\]
```

There are two ways to quote an expression
```
Quote: '(a b) => (a b)
Quasiquote: '(a ,(+ b 1)) => (a (unquote (+ b 1))
```

They are different because parts of a quasiquoted expression can be
unquoted with ,
```
Quote: '(a ,(+ b 1)) => (a (unquote (+ b 1))
Quasiquote: '(a ,(+ b 1)) => (a 5)
```

Quasiquote is particularly convenient for generating Scheme
expressions:
```
(define (make-add-procedure n) '(lambda (d (+ d ,n))))
(make-add-procedure 2) => (lambda (d (+ d 2))
```

; Sum the squares of even numbers less than 10, starting with 2
; x = 2
; total = 0
; while x < 10:
; total = total + x * x
; x = x + 2
; RESULT: 2 * 2 + 4 * 4 + 6 * 6 + 8 * 8 = 120

(begin)
(define (f x total)
  (if (< x 10)
      (f (+ x 2) (+ total (* x x)))
      total))
(f 2 0)

; Sum the numbers whose squares are less than 50, starting with 1
; x = 1
; total = 0
; while x * x < 50;
; total = total + x
; x = x + 1
; RESULT: 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

(begin)
(define (f x total)
  (if (< (* x x) 50)
      (f (+ x 1) (+ total (* x x)))
      total))
(f 1 0)

(define (sun-while starting-x while-condition add-to-total update-x)
  (eval (sun-while 2 '(< (* x x) 50) 'x '(+ x 1))) => 120
  (eval (sun-while 1 '(< (* x x) 50) 'x '(+ x 1))) => 28

(begin)
(define (f x total)
  (if ,while-condition
      (f ,update-x (+ total ,add-to-total))
      total))
(f ,starting-x 0))

<table>
<thead>
<tr>
<th>Latitude</th>
<th>Longitude</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>122</td>
<td>Berkeley</td>
</tr>
<tr>
<td>42</td>
<td>71</td>
<td>Cambridge</td>
</tr>
<tr>
<td>45</td>
<td>93</td>
<td>Minneapolis</td>
</tr>
</tbody>
</table>
```

A column has a name and a type
```
A row has a value for each column
```

CREATE TABLE parents AS
```
SELECT "abraham" AS parent, "barack" AS child UNION
SELECT "abraham", "clinton" UNION
SELECT "delano", "herbert" UNION
SELECT "fillmore", "abraham" UNION
SELECT "fillmore", "delano" UNION
SELECT "fillmore", "grover" UNION
SELECT "eisenhower", "fillmore";
```

CREATE TABLE dogs AS
```
SELECT "abraham" name, "long" as fur UNION
SELECT "barack", "short" UNION
SELECT "clinton", "long" UNION
SELECT "delano", "short" UNION
SELECT "eisenhower", "curly" UNION
SELECT "grover", "short" UNION
SELECT "herbert", "curly";
```

SELECT a.child AS first, b.child AS second
FROM parents a, parents as b
WHERE a.parent = b.parent and a.child = b.child;
```

The number of groups is the number of unique values of an expression
A having clause filters the set of groups that are aggregated
```
select weight/legs, count(*) from animals
having weight/legs
```

<table>
<thead>
<tr>
<th>kind</th>
<th>legs</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>dog</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>cat</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>ferret</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>parrot</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>penguin</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>t-rex</td>
<td>2</td>
<td>12000</td>
</tr>
</tbody>
</table>
Calculator Expression

Expression Tree

how to Design Functions:
1) Identify the information that must be represented and how it is represented. Illustrate with examples.
2) State what kind of data the desired function consumes and produces. Formulate a concise answer to the question what the function computes.
3) Work through examples that illustrate the function’s purpose.
4) Outline the function as a template.
5) Fill in the gaps in the function template. Exploit the purpose statement and the examples.
6) Convert examples into tests and ensure that the function passes them.