Exceptions are raised with a raise statement.

\[ \text{raise <expr> \rangle} \]

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

```
try:
  <try suite> as <name>:
    <except clause> as <name>:
      except BaseException as e:
        print('handling a', type(e))
        handling a <class 'ZeroDivisionError'>
        x = 8
      except ZeroDivisionError as e:
        print('Handling divide by zero')
        x = 1/0

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 <exception class>, then 
The <except suite> is executed, with <name> bound to the exception.
```

The built-in Scheme list data structure can represent combinations
\[ \text{sccm (list 'quotient 10 2) \text{ sccm (eval (list 'quotient 10 2)) (quotient 10 2) 5}} \]

There are two ways to quote an expression
```
Quote:    '(a b) == (a b)
Quasiquote:    (a b) == (a (quote b))
```

They are different because parts of a quasiquoted expression can be unquoted with,
```
(define b 4)
(define (mu (x) (+ x y)))
```

\[ \text{Quasiquote is particularly convenient for generating Scheme expressions:} \]
```
(define (make-add-procedure n) ('(lambda (d) (+ d n)))
(make-add-procedure 2) == (lambda (d) (+ d 2))
```

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls.

A tail call is a call expression in a tail context, which are:
- The last body expression in a lambda expression
- Expressions 2 & 3 (consequent & alternative) in a tail context if
  all final sub-expressions in a tail context cond
- The last sub-expression in a tail context cond, and, or, begin, or let
```
(define factorial n k)
(define (length s)
  ; Not a tail call
  (if (null? s) 0
    (+ 1 (length (cdr s))))

(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n)))))

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope).

Lexical scope: The parent of a frame is the environment in which a procedure was defined. \( \text{(lambda ...)} \)

Dynamic scope: The parent of a frame is the environment in which a procedure was called. \( \text{(mu ...)} \)

The | character matches either of two sequences
\[ \text{fall(Spring) 28\{\text{a|d|e}\}} \text{ matches either Fall 2021 or Spring 2021} \]

A whole group can be repeated multiple times
\[ \text{\{lol|pop|tlo\}} \text{ matches lol and lolol and lololol but not lolo} \]

The ^ and $ anchors correspond to the start and end of the full string
The \b anchor corresponds to the beginning or end of a word

The number of groups is the number of unique values of an expression.
A having clause filters the set of groups that are aggregated
\[ \text{\{dog, weight/legs\} \text{\{weight/legs, count(*)\}} \text{\{weight/legs, count(*)\}} \text{\{weight/legs, count(*)\}} \text{\{weight/legs, count(*)\}} \}
\]

```
A table has columns and rows

<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 row has a value for each column
```

```
SELECT [expression] AS [name], [expression] AS [name], ...
SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order];
```

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

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

```
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 | character matches either of two sequences
\[ \text{fall(Spring) 28\{\text{a|d|e}\}} \text{ matches either Fall 2021 or Spring 2021} \]

A whole group can be repeated multiple times
\[ \text{\{lol|pop|tlo\}} \text{ matches lol and lolol and lololol but not lolo} \]

The ^ and $ anchors correspond to the start and end of the full string
The \b anchor corresponds to the beginning or end of a word
Scheme programs consist of expressions, which can be:
- Primitive expressions: 2, 3.3, true, +, quotient, ...  
- Combinations: (quotient 10 2) (not true).  

Numbers are self-evaluating; symbols are bound to values.  

Call expressions have an operator and 0 or more operands.  

A combination that is not a call expression is a special form:
  
- If expression: (if (and <predicate> <consequent> <alternative>) ...  
- Binding names: (define <name> <expression>)  
- New procedures: (define <name-alias> (<name> <formal-parameters>) <body>)
  
```scheme
(define pi 3.14)  
(define abs x)  
(if (< x 0)       
  (- x)  
  x))
```

Lambda expressions evaluate to anonymous procedures:
```
(lambda (<formal-parameters>) <body>)
```

Two equivalent expressions:
```
(define plus4 (lambda (x) (+ x 4)))
(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a combination too:
```
(l (x y z) (+ y (square z))) 1 2 3
```

In the late 1950s, computer scientists used confusing names.
- cons: Two-argument procedure that creates a pair  
- car: Procedure that returns the first element of a pair  
- cdr: Procedure that returns the second element of a pair  
- nil: The empty list  

They also used a non-obvious notation for linked lists.
- A (linked) Scheme list is a pair in which the second element is nil or a Scheme list.  
- A list is written as space-separated combinations.  
- A dotted list has an arbitrary value for the second element of the last pair.  

Dotted lists may not be well-formed lists.  
```
> (define x (cons 1 nil))  
> x
[0]
> (car x)  
1
> (cdr x)  
()  
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

Symbols normally refer to values; how do we refer to symbols?
```
> (define a 1)  
> (define b 2)  
> (list a b)
(1 2)
```

Quotation is used to refer to symbols directly in Lisp.  
```
> (list 'a 'b)
(a b)
> (list 'a 'b)
(a b)
```

Quotation can also be applied to combinations to form lists.  
```
> (car (a b c))
(a b)
> (cdr (car b c))
(b c)
```

Symbol references are instances of item:
```
(car (cons 1 nil)) => 1  
(cdr (cons 1 nil)) => ()  
(cdr (cons 1 (cons 2 nil))) => (2)
```

A Scheme list is written as elements in parentheses:
```
’((’’a’’) ’’b’’))  
’((’’a’’) ’’a’’)  
’((’’a’’) ’’b’’ ’’c’’))
```

Each element can be a combination or a pair (primitive).
```
(+ 3 (+ 2 4) (+ 3 5)) (+ 10 7 6)
```

The task of parsing a language involves coercing a string representation of an expression to the expression itself.  

Parsers must validate that expressions are well-formed.  

A Parser takes a sequence of lines and returns an expression.
```
(define (f s) (if (null? s) ’(3) (cons (car s) (f (cdr s)))))
```

Each call to scheme_read consumes the input tokens for exactly one expression.

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them

A Scheme list is written as elements in parentheses:
```
’((’’a’’) ’’b’’) ’’c’’)
```

Each element can be a combination or a pair (primitive).
```
(+ 3 (+ 2 4) (+ 3 5)) (+ 10 7 6)
```

The task of parsing a language involves coercing a string representation of an expression to the expression itself.  

Parsers must validate that expressions are well-formed.  

A Parser takes a sequence of lines and returns an expression.
```
(define (f s) (if (null? s) ’(3) (cons (car s) (f (cdr s)))))
```

Each call to scheme_read consumes the input tokens for exactly one expression.

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them

A Scheme list is written as elements in parentheses:
```
’((’’a’’) ’’b’’) ’’c’’)
```

Each element can be a combination or a pair (primitive).
```
(+ 3 (+ 2 4) (+ 3 5)) (+ 10 7 6)
```

The task of parsing a language involves coercing a string representation of an expression to the expression itself.  

Parsers must validate that expressions are well-formed.  

A Parser takes a sequence of lines and returns an expression.
```
(define (f s) (if (null? s) ’(3) (cons (car s) (f (cdr s)))))
```

Each call to scheme_read consumes the input tokens for exactly one expression.

**Base case:** symbols and numbers

**Recursive call:** scheme_read sub-expressions and combine them