Exceptions are raised with a raise statement.

Exceptions are raised whenever next is called on an empty iterator.

A StopIteration exception is raised whenever next is called on an empty iterator.

A generator function is a function that yields values instead of returning them.

A normal function returns once; a generator function can yield multiple times.

A generator is an iterator created automatically by calling a generator function when a generator function is called, it returns a generator.

A StopIteration exception is raised whenever next is called on an empty iterator.

A column has a name and a type.

A row has a value for each column.

A table has columns and rows.
Scheme programs consist of expressions, which can be:

- **Primitive expressions**: `2 3.1 true`. `...`
- **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 (op pred) <element 1> <element 2>)`
- **Binding names**: `(define <name> <expression>)`
- **New procedures**: `(define (<name> <formal parameters>) <body>)`

Two equivalent expressions:

```scheme
(define x (cons 1 2))
> x
(1 2)
> (cdr x)
2
```

An operator can be a combination too:

```scheme
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

Lambda expressions evaluate to anonymous procedures.

Two equivalent expressions:

```scheme
lambda (def-formal-parameters) body
```

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.
- Scheme lists are 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 2))
> x
(1 2)
> (cdr x)
2
```

Symbols normally refer to values; how do we refer to symbols?

```scheme
> (define a 1)
> (define b 2)
> (list a b) 1 2 3
```

Quotation is used to refer to symbols directly in Lisp:

```scheme
> (list 'a 'b) a b
> (list 'a 'b) a b
```

Quotation can also be applied to combinations to form lists:

```scheme
> (car '(a b c))
> (cdr '(a b c))
```

Dots can be used in a quoted list to specify the second element of the final pair.

```scheme
> (cdr (list 1 2 3))
3
```

However, dots appear in the output only of ill-formed lists.

```scheme
> 'a 'b
(a b)
> (list 'a 'b)
(a b)
```

A basic interpreter has two parts: a parser and an evaluator.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Parser</th>
<th>Evaluator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(2 3)</code></td>
<td>Pair(<code>2</code>, Pair(0, Pair(2, nil)))</td>
<td>4</td>
</tr>
<tr>
<td><code>(3 6)</code></td>
<td>Pair(<code>3</code>, Pair(<code>6</code>, nil))</td>
<td>4</td>
</tr>
</tbody>
</table>

Each `<element>` can be a combination or atom (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 Scheme list is written as elements in parentheses:

```scheme
(lambda (<formal-parameters>) <body>)
```

Expression Trees

```
       *     + 5 6
       / + 4 7
      / + 3 8
    + 2 1
```

Base cases: **Primitive values (numbers)**

- **Look up values bound to symbols**
- **Recursive calls:** Apply(procedure, arguments)
- **Eval(sub-expressions) of special forms**

To apply a user-defined procedure, create a new frame in which all parameters are bound to argument values, whose parent is the environment of the procedure, then evaluate the body of the procedure in the environment that starts with this new frame.

```scheme
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
```

```scheme
(f (list 1 2))
```

Expression Trees

```
       *     + 5 6
       / + 4 7
      / + 3 8
    + 2 1
```

The structure of the Scheme Interpreter

```
       *     + 5 6
       / + 4 7
      / + 3 8
    + 2 1
```

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 and 3 (consequent and alternative) in a tail context
- if expression

```scheme
(define (factorial n k)
  (if (= k 0) k
    (* (factorial (- n 1) (* k n))
      (+ 1 (length (cdr s)))))
```

A tail call is a recursive call that has not yet been evaluated.

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

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

A recursive call is a tail call

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

A recursive call is a tail call

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

A recursive call is a tail call