Exceptions are raised with a raise statement.

```python
raise <expression>
```

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

Exceptions are constructed like any other object. E.g., `TypeError('bad argument')`

```python
try:
    >>> try:
    x = 1/0
    >>> try:
    x = 1/0
except ZeroDivisionError as e:
    print(f"Handling {e} of type {e.__class__}")
    x = 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`, then the `try` suite is executed, with `name` bound to the exception.

```python
for name in <expression>:
    >>> for name in <expression>:
        try:
            x = 1/0
            >>> try:
            x = 1/0
            except ZeroDivisionError as e:
                print(f"Handling {e} of type {e.__class__}")
        >>> try:
        x = 1/0
            x = 0
        >>> try:
        x = 1/0
    >>> for name in <expression>:
```

A table has columns and rows. An exception is raised if and when there are fewer columns in the table than the number of values in any row.

```python
>>> counts = [1, 2, 3]
>>> for item in counts:
>>>     print(item)
```

A generator is an iterator backed by a function. Each time a generator function is called, it returns a generator.

```python
A stream is a Scheme pair, but the cdr is evaluated lazily
```

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 the procedure was defined. (lambda ...)

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

```python
> (define f (mu (x) (+ x y)))
> (define g (lambda (x y) (if (+ x y)))
> (g 3 7)
13
```
Scheme programs consist of expressions, which can be:

- Primitive expressions: `2 3 4.5 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 o pred conseq alternative)
- Binding names: `(define <name> <expression>)
- New procedures: `(define <name> <formal parameters> <body>)

Two equivalent expressions:

- `(define (plus4 x) (+ x 4))
- `(define plus4 (lambda (x) (+ x 4)))

An operator can be a combination too:

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

A Scheme list is written as elements in parentheses:

- Each `element` can be a combination or atom (primitive).
- A combination or atom is a Scheme expression.
- A 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.

### Syntax Tree

#### Base cases
- Primitive values (numbers)
- Look up values bound to symbols

#### Recursive calls
- Apply(operator, operands) of call expressions
- Apply(procedure, arguments)
- Apply(sub-expressions) of special forms

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

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

Base case: symbols and numbers

Recursive call: scheme_read_sub-expressions and combine them

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

```
(define (f s) (if (null? s) '3 (cons (car s) (f (cdr s)))))
(f (list 1 2))
```

### Representation as Pairs

#### Class Pair

```
```

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 expression

```
(define (factorial n k) (if (= 0 k) 1 (* n (factorial (- n 1) k)))))
```

A not a tail call

```
(define (length-s t s) (if (null? s) 0 (+ 1 (length (cdr s)))))
```