Exceptions are raised with a raise statement.

```
raise <expression>
```

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

Exceptions are constructed like any other object. E.g.,
```
Typeobject('Bad argument!'...)`
```

try:
    <try suite>
    except <exception class> as <name>:
    ...  

The <try suite> is executed first. If, during the course of 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.

```
for <name> in <expression>:
``` 

1. Evaluate the header <expression>, which yields an iterable object.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the first frame of the current environment.
   B. Execute the <suite>.

An iterable object has a method __iter__ that returns an iterator.

```
>>> item = item1._iter__()
>>> for item in items:
    print(item)
```

```>print('i', item1._iter__())</>```

A stream is a linked list, but the rest of the list is computed on demand. Once created, Streams and Rlists can be used interchangably using first and rest.

```
class Stream:
    # A lazily computed linked list.
    class empty:
        def __repr__(self):
            return 'Stream.empty'
    empty = empty

def __init__(self, first=empty, rest=None):
    self.first = first
    self.rest = rest

def filter_stream(f, s):
    if s is Stream.empty:
        return s
    return filter_stream(f, rest)

def map_stream(f, s):
    if s is Stream.empty:
        return s
    return map_stream(f, rest)

def first_iterator(s):
    return Stream(first, second_iterator(rest))

def second_iterator(s):
    return Stream(first, second_iterator(rest))

def first(s):
    return first_iterator(s)

def rest(s):
    return second_iterator(s)

def inner_filter_stream(f, s, r):
    if r is Stream.empty:
        return Stream.empty
    return inner_filter_stream(f, r, rest(r))

def compute_rest(s):
    return Stream(first, first_iterator(rest(s)))

def primes(positives):
    if not positives:
        return []
    return compute_rest(compute_rest(first(rest(positives)))).
``` 

The number of groups is the number of unique values of an expression. A having clause filters the set of groups that are aggregated.
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 (if predicate consequent alternative)
- **Binding names**: (define <name> <expression>)
- **New procedures**: (define (name <formal parameters>) <body>)

`pi` is either a well-formed list or nil.

```
> (define pi 3.14)
> (define (abs x)
   (if (< x 0)
       (- x)
       x))
> (abs -3)
3
```

Lambda expressions evaluate to anonymous procedures:

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

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

An operator can be a combination too:

```
((lambda (x y z) (+ x 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
- **null**: 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 null 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
> (car x)
1
> (cdr x)
2
> (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)
```

Quotation can also be applied to combinations to form lists.

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

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

```
> (cdr '(1 . 3))
3
```

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

```
> '(1 2 . 3)
(1 2)
> '(1 . 3)
(...)
> '(1 2 3 . nil)
(1 2 3)
> '(1 . 3 . nil)
(...)
```

### Class Pair

```
"""A Pair has first and second attributes.

For a Pair to be a well-formed list, second is either a well-formed list or nil."
"""
```

```
def __init__(self, first, second):
    self.first = first
    self.second = second

>>> p = Pair(1, Pair(2, Pair(3, nil)))
>>> print(p)
[(1, 2, 3)]
>>> len(p)
3
>>> print(Pair(1, nil))
[(1,)]
```

The calculator language has primitive expressions and call expressions.

```
Calculator Expression
(+ 3 (+ 4 5))
```

```
Expression Tree
* 3
+ 4 5
```

A Scheme list is written as elements in parentheses:

```
(list 'a 'b 'c)
```

Each `<element>` can be a combination or atom (primitive).

```
(+ 3 (+ 2 4) (+ 3 5))
```

### A Scheme Interpreter

1. **Lexical analysis**
   - **Tokens**: Symbols and numbers
   - **Syntax analysis**: Recursive call: `scheme_read_sub-expressions and combine them`

2. **Base cases**: `evaluate values` and `lookup symbols`

3. **Eval**: `apply procedures` and `apply user-defined procedures`

4. **Apply**: `create a new environment each time a user-defined procedure is applied`

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 (factorial n)
  (if (null? n) 1
      (factorial (- n 1))))
```

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

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 (lambda-proc s)
  (lambda (x)
    (if (null? s) nil
        (cons (car s) (lambda-proc (cdr s))))
  )
```

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

A tail call is a tail call.