A simple fact expression in the Logic language declares a relation to be true.

Language Syntax:
- A relation is a Scheme list.
- A fact expression is a Scheme list of relations.

```logi
logioc (fact (parent delano herbert))
logioc (fact (parent barack clinton))
logioc (fact (parent abraham clinton))
logioc (fact (parent fillmore delano))
logioc (fact (parent fillmore grover))
logioc (fact (parent eisenhower fillmore))
```

Relations can contain relations in addition to atoms.

```logi
logioc (fact (dog (name abraham) (color white)))
logioc (fact (dog (name barack) (color tan)))
logioc (fact (dog (name clinton) (color white)))
logioc (fact (dog (name delano) (color black)))
logioc (fact (dog (name eisenhower) (color tan)))
logioc (fact (dog (name fillmore) (color brown)))
```

Variables can refer to atoms or relations in queries.

```logi
logioc (query (parent abraham child))
Success!
child: barack
child: clinton
logioc (query (dog (name clinton) ?color))
Success!
info: (color white)
```

A fact can include multiple relations and variables as well:

```logi
logioc (fact conclusion <hypothesis> <hypothesis> ... <hypothesis>)
Means <conclusion> is true if all <hypothesis> are true.
logioc (fact (child ?x ?y) (parent ?p ?q))
logioc (query (child herbert delano))
Success!
```

The Logic interpreter performs a search in the space of relations for each query to find a satisfying assignment.

```logi
logioc (query (child ?x ?y))
Success!
child: abraham
child: delano
child: clinton
```

Two lists append to form a third list if:
- The first list is empty and the second and third are the same
- The rest of 1 and 2 append to form the rest of 3

```logi
logioc (fact (append-to-form (x) ?x))
logioc (fact (append-to-form (?a . ?r) ?y (7a . 7z))
(append-to-form ?r ?y ?z))
```

A generator is an iterator backed by a generator function.
- When a generator function is called, it returns a generator.
A basic interpreter has two parts: a parser and an evaluator. A Scheme list is written as elements in parentheses:

```
(element <element> ... <element>)
```

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

- `(+ (1 + (2 4 + 3)) (+ 7 10))`
- `((- (1 + (2 4 + 3 )) (+ 7 10 )))`

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.

```
Lexical analysis
Tokens
Syntactic analysis
Expression
```

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time
- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested. Each call to `scheme_read` consumes the input tokens for exactly one expression.

**Base case:** symbols and numbers

**Recursive call:** `scheme_read` subprocesses and combine them

A Scheme list represents the syntactic structure of an expression.

- Numbers
- Symbols
- Parentheses
- Commas
- Spaces

The structure of the Scheme interpreter creates new environments when applying user-defined procedures.

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)
```

```
(lambda (x y z) (+ x y (square z)))
```

### Lexical analysis tokens

- Symbol
- Operator
- Space
- Comma
- Frame

### Syntactic analysis expression

- Syntactic structure
- Trees
- Roots
- Leaves
- Branches

### Recursive call

- Environment
- Evaluation
- Procedure
- Arguments

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 & alternative) in a tail context if expression

### Example

#### Expressions

- `(define (factorial n) (+ (* n (- n 1)) (factorial (- n 1))))`

- `(define (length s) 0)`

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

#### Recursive call

A recursive call is a tail call.