Here's a brief summary of Scheme for the purposes of project 1.

Scheme data

There are two main data types in Scheme: symbols (which include numbers), and lists. A list is displayed with parentheses enclosing the list elements, for example:

```
(a b 3)
```

A list may be empty; the empty list, displayed as ( ), is named NIL. List elements may also be lists themselves, for example:

```
((a b) (x ( ) (3 4 5)))
```

Evaluation of Scheme expressions

A Scheme expression is either a symbol or a list that may be provided as input to the interpreter. The interpreter executes a loop with the following steps:

1. read the next symbol or list from the user;
2. evaluate it;
3. print the result.

Evaluation is conceptually a big switch:

1. A numeral's value is the corresponding numeric value.
2. A symbol's value is what it gets initialized to via the define function (see below). If the symbol has not been provided as an argument to define, the error message "undefined variable" results.
3. A quoted expression's value is the expression itself, taken literally. For example, the value of '(a b) is the list (a b). Within a quoted expression, the quote translates to a call to the function quote; thus '(a b '(x y)) would evaluate to the list (a b (quote (x y))).
4. A parenthesized expression is evaluated differently, depending on whether or not it is a *special form*. In a special form, the word following the left parenthesis is either quote or define.

   The quote function takes a single argument, the "quoted" expression; the value of a call to quote is the quoted expression. Thus the value of the expression (quote (a b)) is the list (a b).

   The define function declares and initializes a variable. The variable declared/initialized is the first argument of define; the value it's initialized to is the result of evaluating the second argument. If a variable is used before it appears in a define expression, an "undefined variable" error message results. A second define for a variable merely replaces the value associated with that variable.
5. Otherwise, the first thing after the left parenthesis must name one of the functions +, cons, car, or cdr, and the remaining things that precede the right parenthesis are arguments. The value is computed by recursively evaluating the arguments, then applying the function to them.

Builtin functions

The cons function (short for "construct") takes two arguments, a Scheme expression and a Scheme list. It returns a *pair* whose "car" (first element) is the value of the first argument and whose "cdr" (second element) is the value of the second argument. A pair whose second element is a list is itself a list. Thus lists may be defined recursively:

> A list is either empty—printed as ( )—or the result of evaluating a call to cons with a list as the second argument.

A pair that's not a list is displayed with a dot between the elements of the rightmost cons.

Here are some examples.

expression	displayed value
`(cons 1 (cons 2 3))`	`(1 2 . 3)`
`(cons 1 '(2 3))`	`(1 2 3)`
`(cons 'a '(b c))`	`(a b c)`
`(cons '(b c) '(a x))`	`((b c) a x)`

Note that leaving out the quotes in (cons 'a '(b c)), thus trying to evaluate the expression (cons a (b c)), would result in trying to evaluate the symbol a, which is fine if a has been define'd, and the expression (b c), which is a call to the function named b and is thus not allowed in this project.

The car and cdr functions each take a pair—the result of some cons operation—as argument, and return the first or second element of the pair, respectively. Examples:

expression	displayed value
`(car (cons 'a '(a b c)))`	`a`
`(car '((x y) z))`	`(x y)`
`(cdr (cons 'a '(a b c)))`	`(a b c)`
`(cdr '((x y) z))`	`(z)`

The + function takes two numeric values as arguments and returns their sum.