A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 true + quotient
- Combinations: (quotient 10 2) (not true)

The built-in Scheme list data structure (which is a linked list) can represent combinations:

```scheme
scm> (list 'quotient 10 2)
```

In such a language, it is straightforward to write a program that writes a program:

```scheme
(scm> (list 'quotient 10 2))
```

Macros

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation:

```scheme
(define-macro (twice expr)
  (list 'begin expr expr))
```

```scheme
(scm> (twice (print 2)))
```

Evaluation procedure of a macro call expression:

1. Evaluate the operator sub-expression, which evaluates to a macro.
2. Call the macro procedure on the operand expressions without evaluating them first.
3. Evaluate the expression returned from the macro procedure.

For Macro

```scheme
(define-macro (for sym vals expr)
  (list 'map (list 'lambda (list sym)) expr vals))
```

```scheme
(scm> (for x '(2 3 4 5) (* x x)))
```

Discussion Question

Define a macro that evaluates an expression for each value in a sequence:

```scheme
(define-macro (map fn vals)
  (if (null? vals)
      ()
      (cons (fn (car vals)) (map fn (cdr vals)))))
```

```scheme
(scm> (map (lambda (x) (* x x)) '(2 3 4 5)))
```

Quasi-Quotation

```scheme
(scm> (define-macro (for x vals expr)
  (list 'map (list 'lambda (list x)) expr vals)))
```

```scheme
(scm> (for x '(2 3 4 5) (+ x x)))
```

```scheme
(scm> (define-macro (for x vals expr)
  (list 'map (list 'lambda (list x)) expr vals)))
```

```scheme
(scm> (for x '(2 3 4 5) (+ x x)))
```