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

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

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

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

### Homoiconicity

Homoiconic Languages

Languages have both a concrete syntax and an abstract syntax

- **Python Demo**

A language is homoiconic if the abstract syntax can be read from the concrete syntax

- **Scheme Demo**

Quotation is actually a combination in disguise

- **Quote Demo**

### 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 homoiconic languages

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

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

Evaluation procedure of a macro call expression:

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

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

### Problem 1

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)))
```

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

```scheme
(define-macro (for x (+ x x) '(2 3 4 5))
```

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

Define a function `nest` that builds a nested list containing its arguments:

```
(define (nest first rest)
  (if (null? rest)
      (list first)
      (list first (apply nest rest)))))
```

```scm
(scm> (nest 3)
      ()
(scm> (nest 3 4 5 6)
      (3 4 (5 (6))))
```

Temporary Symbols