When a Scheme expression has a single value, the programmer can give that value a name by embedding the expression in a definition, which comprises the keyword define, the name, and the expression. Programmers conventionally place a line break after the name and indent the expression slightly:

```scheme
(define sum
  (lambda addends
    (apply + addends)))
```

When the definition is processed, the value of the expression is placed in a storage location to which the name is bound. (In effect, the name is a variable with the value of the expression as its value.)

When the expression in a definition is a lambda-expression, one can write an equivalent definition in a “short form” that fuses the name and the formals specification and omits the keyword lambda, thus:

```scheme
(define (cube number)
  (* number number number))

(define (average . arguments)
  (/ (apply + arguments) (length arguments)))

(define (write-line first . rest)
  (display first)
  (for-each (lambda (arg)
      (display #\space)
      (display arg))
      rest)
  (newline))
```

Definitions are typically collected into thematically related groups called libraries. Syntactically, a library comprises the keyword define-library, a library specification, an export list, an import list, and finally a begin-expression containing a sequence of zero or more forms (definitions and expressions). The entire define-library-expression is enclosed within a pair of parentheses. Here’s a short example:

```scheme
(define-library (discrete powers)
  (export square cube zenzizenzic sursolid)
  (import (scheme base))
  (begin

    (define (cube number)
      (* number number number))

    (define (zenzizenzic number)
      (square (square number)))))
```
(define (sursolid number)
  (* (square number) (cube number))))

The first line indicates that this library is called powers and is included in the discrete collection. It exports the identifiers square, cube, zenzizenzic, and sursolid making them available for use in Scheme programs and in other libraries. This library imports definitions from the base library in the scheme collection, which includes the procedures * and square. Note that it is possible for a library to export a procedure that it has simply imported from another library, without providing a separate definition of it.

It is a good idea to put each library in its own file. In many implementations, libraries are supposed to have a distinct filetype, such as .sls ("Scheme library source"), but DrRacket does not recognize this filetype, so in this course we’ll use .ss both for libraries and for top-level Scheme programs.

DrRacket expects us to follow several other conventions about naming files as well: The part of the filename that precedes the .ss filetype should match the last identifier in the library specification. For example, the file containing the library shown above should be named powers.ss. In addition, the collection to which this library belongs should correspond to a directory containing the libraries in that collection, which should have the same name as the collection (in this case, discrete).

A collection can also contain other collections, just as a directory can include other directories. The two structures should match up, so that (for instance) a library specified as (discrete utilities lists), meaning that it is the lists library in the utilities subcollection within the discrete collection, might appear in the MathLAN file system as

/home/spelvin/MAT208/discrete/utilities/lists.ss

Thus a library specification is an S-structure comprising one or more Scheme identifiers, naming the collection or hierarchy of collections to which the library belongs, and ending with the identifier for the library proper.

An export list is an S-structure comprising the keyword export and the names of some or all of the items defined in the library, as those names appear in the definitions. These names, and the values they denote, can be imported by top-level programs and by other libraries, and used there as if they were predefined.

An import list is an S-structure comprising the keyword import and zero or more import sets. An import set names a set of bindings — names and the values associated with them — from another library and makes them available in this library, possibly renaming them along the way. Each import set has one of the following five forms:

- A library specification. In this case, all the bindings exported by the specified library are imported.
- An S-structure comprising the keyword only, a library specification, and zero or more identifiers. Each of the identifiers must be one of the names exported by the specified library. Only the bindings for those names are imported.
- An S-structure comprising the keyword except, a library specification, and zero or more identifiers. Each of the identifiers must be one of the names exported by the specified library. All of the bindings exported by the specified library, except the ones for the listed names, are imported.
- An S-structure comprising the keyword prefix, a library specification, and an identifier. All of the bindings exported by the specified library are imported, but the name of each one is changed to a name formed by concatenating the identifier in the import set with the exported name.
• An S-structure comprising the keyword rename, a library specification, and zero or more renaming specifications. Each renaming specification is an S-structure comprising two identifiers, the first of which must be the name of a binding exported by the specified library. The bindings associated with the names in the renaming specifications are imported, but the name of each one is replaced by the other identifier in the renaming specification.

Actually, it’s possible to nest import sets (that is, an import set can occur anywhere a library specification can occur in the preceding account), but there shouldn’t be any need for us to take advantage of the additional generality.

The begin-expression that follows the import list usually contains only definitions, but it may contain expressions as well. The expressions are evaluated when the library is imported, for side effects only (the values of the expressions are ignored). A typical use for such expressions is to initialize a complex data structure that is defined in the library.

A top-level program in Scheme begins with an import list, exactly like the import list in a library. Importing bindings from a library makes them available inside the program. A top-level program contains an import list and zero or more forms. The bindings created by the definitions, which are typically placed before the expressions, can be used in the expressions. The execution of the program consists of the evaluation of the expressions it contains, in order, for side effects only (again, the values of the expressions are ignored).
Exercises

1. Define a unary predicate that determines whether its argument is a natural number, that is, an exact non-negative integer.

2. Construct a definition for the name inches-in-a-foot, giving it the value 12.

3. Construct a definition for the name seconds-in-a-week, computing the value (an integer) by appropriate invocations of the * procedure.

4. Construct definitions of several other conversion factors that you know or can easily compute. Collect them into a (discrete conversions) library and export all of them. What libraries, if any, will (discrete conversions) have to import?

5. Write and test a top-level program that computes and displays the number of inches in 362 feet, importing appropriate definitions from your (discrete conversions) library.

6. Construct an import list that imports only the cube procedure from the (discrete powers) library.

7. Construct an import list that imports all of the procedures from the (discrete powers) library, adding the prefix raise- to the name of each one.