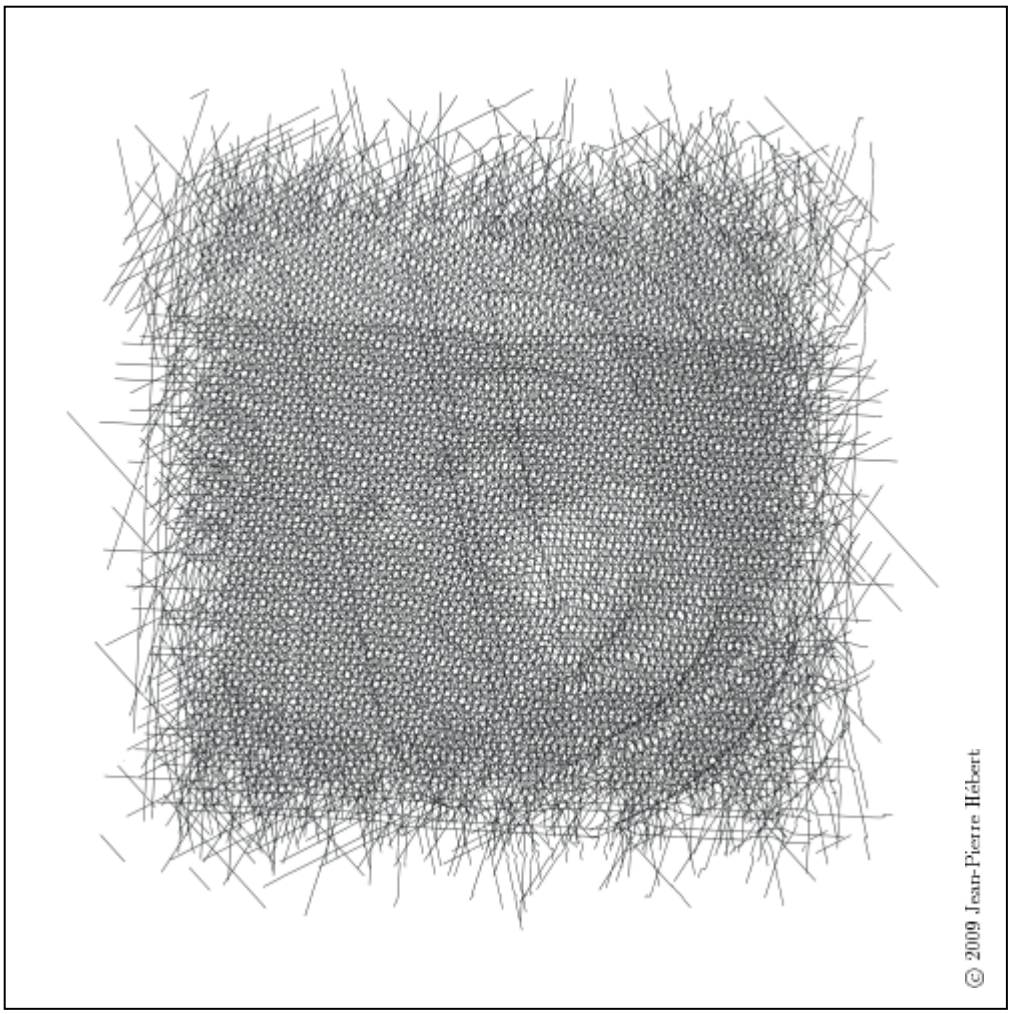

Chapter 10. Libraries and Top-Level Programs

Libraries and *top-level programs* are the basic units of portable code in the language defined by the Revised[6] Report on Scheme [24]. Top-level programs may import from one or more libraries, and libraries may import from other libraries.

Libraries are named using a parenthesized syntax that encloses a sequence of identifiers, optionally followed by a version; the version is itself a parenthesized form that encloses a sequence of subversions represented as exact nonnegative integers. So, for example, `(a)`, `(a b)`, `(a b ())`, and `(a b (1 2 3))` are all valid library names. Implementations typically treat the sequence of names as a path by which a library's source or object code can be found, possibly rooted in some standard set of locations in the host-machine's filesystem.

An implementation of the standard library mechanism is available with the portable implementation of `syntax-case` at *http://www.cs.indiana.edu/syntax-case/*.

Section 10.1. Standard Libraries

The Revised[6] Report [24] describes a base library

```
(rnrs base (6))
```

that defines the most commonly used features of the language. A separate Standard Libraries document [26] describes the libraries listed below.

```
(rnrs arithmetic bitwise (6))
(rnrs arithmetic fixnums (6))
(rnrs arithmetic flonums (6))
```

```
(rnrs bytevectors (6))
(rnrs conditions (6))
(rnrs control (6))
(rnrs enums (6))
(rnrs eval (6))
(rnrs exceptions (6))
(rnrs files (6))
(rnrs hashtables (6))
(rnrs io ports (6))
(rnrs io simple (6))
(rnrs lists (6))
(rnrs mutable-pairs (6))
(rnrs mutable-strings (6))
(rnrs programs (6))
(rnrs r5rs (6))
(rnrs records procedural (6))
(rnrs records syntactic (6))
(rnrs records inspection (6))
(rnrs sorting (6))
(rnrs syntax-case (6))
(rnrs unicode (6))
```

One more library is described in the Standard Libraries document, a composite library

```
(rnrs (6))
```

that exports all of the `(rnrs base (6))` bindings along with those of the other libraries listed above, except those of `(rnrs eval (6))`, `(rnrs mutable-pairs (6))`, `(rnrs mutable-strings (6))`, and `(rnrs r5rs (6))`.

Although each of these libraries has the version `(6)`, references to them can and in most cases should leave the version out, e.g., the composite library should be referenced simply as `(rnrs)`.

Section 10.2. Defining New Libraries

New libraries are defined with the `library` form, which has the following syntax.

```
(library library-name
  (export export-spec ...)
  (import import-spec ...)
  library-body)
```

A *`library-name`* specifies the name and possibly version by which the library is identified by the `import` form of another library or top-level program. It also serves as kind of path that the implementation uses to locate the library, via some implementation-specific process, whenever it needs to be loaded. A *`library-name`* has one of the following two forms:

```
(identifier identifier ...)
(identifier identifier ... version)
```

where *`version`* has the following form:

```
(subversion ...)
```

and each *`subversion`* represents an exact nonnegative integer. A library name with no *`version`* is treated the same as a library name with the empty *`version`* `()`. For example, `(list-tools setops)` and `(list-tools setops ())` are equivalent and specify a library name with no version, while `(list-tools setops (1 2))` specifies a versioned library name, which can be thought of as Version 1.2 of the `(list-tools setops)` library.

The `export` subform names the exports and, optionally, the names by which they should be known outside of the library. Each *`export-spec`* takes one of the following two forms:

identifier
(rename (*internal-name export-name*) ...)

where each *internal-name* and *export-name* is an identifier. The first form names a single export, *identifier*, whose export name is the same as its internal name. The second names a set of exports, each of whose export name is given explicitly and may differ from its internal name.

The `import` subform names the other libraries upon which the new library depends and, possibly, the set of identifiers to be imported and the names by which they should be known inside the new library. It may also specify when the bindings should be made available for implementations that require such information. Each *import-spec* takes one of the following two forms:

import-set
(for *import-set import-level* ...)

where *import-level* is one of the following:

run
expand
(meta *level*)

and *level* represents an exact integer.

The `for` syntax declares when the imported bindings might be used by the importing library and thus when the implementation must make the bindings available. `run` and `(meta 0)` are equivalent and specify that the bindings imported from a library might be referenced by the run-time expressions (`define` right-hand-side expressions and initialization expressions) of the importing library. `expand` and `(meta 1)` are equivalent and specify that the bindings imported from a library might be referenced by the transformer expressions (`define-syntax`, `let-syntax`, or `letrec-syntax` right-hand-side expressions) of the importing library. `(meta 2)` specifies that the bindings imported from a library might be referenced by a transformer expression that appears within a transformer expression of the importing library, and so on for higher meta levels. Negative meta levels may also be specified and are needed in certain circumstances when a transformer expands into the transformer for another keyword binding used at a lower meta level.

A library export may have a non-zero *export* meta level, in which case the effective import level is the sum of the level specified by `for` and the export level. The exports of each standard library except `(rnrs base)` and `(rnrs)` have export level zero. For `(rnrs base)`, all exports have export level zero except for `syntax-rules`, `identifier-syntax`, and their auxiliary keywords `_`, `...`, and `set!`. `set!` has export levels zero and one, while the others have export level one. All exports of the `(rnrs)` library have export levels zero and one.

It can be difficult for the programmer to specify the import levels that allow a library or top-level program to compile or run properly. Moreover, it is often impossible to cause a library's bindings to be made available when they are needed without causing them to be made available in some cases when they are not needed. For example, it is not possible to say that the run-time bindings of a library A are needed when a library B is expanded without also having the run-time bindings of A made available when code importing B is expanded. Making bindings available involves executing the code for the right-hand sides of the bindings and possibly executing initialization expressions as well, so the inability to specify when bindings are needed precisely can add both compile- and run-time overhead to a program.

Because of this, implementations are permitted to ignore export levels and the `for` wrapper on an *import-set* and instead automatically determine, while expanding an importing library or top-level program, when an imported library's bindings must be made available, based on where references to the imported library's exports actually appear. When using such an implementation, the `for` wrapper need never be used, i.e., all *import-spec*s can be *import-set*s. If code is intended for use with systems that do not automatically determine when a library's bindings must be made available, however, the `for` must be used if the importing library's bindings would not otherwise be available at the right time.

An *import-set* takes one of the following forms:

library-spec
(only *import-set identifier* ...)

```
(except import-set identifier ...)
(prefix import-set prefix)
(rename import-set (import-name internal-name) ...)
```

where *prefix*, *import-name*, and *internal-name* are identifiers. An *import-set* is a recursive specification of the identifiers to be imported from a library and possibly the names by which they should be known within the importing library. At the base of the recursive structure must sit a *library-spec*, which identifies a library and imports all of the identifiers from that library. An `only` wrapper restricts the imported identifiers of the enclosed *import-set* to the ones listed, an `except` wrapper restricts the imported identifiers of the enclosed *import-set* to those not listed, a `prefix` wrapper adds a prefix to each of the imported identifiers of the enclosed *import-set*, and a `rename` wrapper specifies internal names for selected identifiers of the enclosed *import-set*, while leaving the names of the other imports alone. So, for example, the *import-set*

```
(prefix
  (only
    (rename (list-tools setops) (difference diff))
    union
    diff)
  set:)
```

imports only `union` and `difference` from the `(list-tools setops)` library, renames `difference` to `diff` while leaving `union` alone, and adds the prefix `set:` to the two names so that the names by which the two imports are known inside the importing library are `set:union` and `set:diff`.

A *library-spec* takes one of the following forms:

```
library-reference
(library library-reference)
```

where a *library-reference* is in either of the following two forms:

```
(identifier identifier ...)
(identifier identifier ... version-reference)
```

Enclosing a *library-reference* in a `library` wrapper is necessary when the first identifier of the *library-reference* is `for`, `library`, `only`, `except`, `prefix`, or `rename`, to distinguish it from an *import-spec* or *import-set* identified by one of these keywords.

A *version-reference* identifies a particular version of the library or a set of possible versions. A *version-reference* has one of the following forms:

```
(subversion-reference_1 ... subversion-reference_n)
(and version-reference ...)
(or version-reference ...)
(not version-reference)
```

A *version-reference* of the first form matches a *version* with at least *n* elements if each *subversion-reference* matches *version*'s corresponding *subversion*. An `and` *version-reference* form matches a *version* if each of its *version-reference* subforms matches *version*. An `or` *version-reference* form matches a *version* if any of its *version-reference* subforms matches *version*. A `not` *version-reference* form matches a *version* if its *version-reference* subform does not match *version*.

A *subversion-reference* takes one of the following forms:

```
subversion
(>= subversion)
(<= subversion)
(and subversion-reference ...)
(or subversion-reference ...)
(not subversion-reference)
```

A *subversion-reference* of the first form matches a *subversion* if it is identical to it. A `>=` *subversion-reference* matches a *version*'s *subversion* if the *version*'s *subversion* is greater than or equal to the

subversion appearing within the >= form. Similarly, a <= *subversion-reference* matches a *version*'s *subversion* if the *version*'s *subversion* is less than or equal to the *subversion* appearing within the >= form. An `and` *subversion-reference* form matches a *version*'s *subversion* if each of its *subversion-reference* subforms matches the *version*'s *subversion*. An `or` *subversion-reference* matches a *version*'s *subversion* if any of its *subversion-reference* subforms match the *version*'s *subversion*. A `not` *subversion-reference* matches a *version*'s *subversion* if its *subversion-reference* subform does not match the *version*'s *subversion*.

For example, if two versions of a library are available, one with version `(1 2)` and the other with version `(1 3 1)`, the version references `()` and `(1)` match both, `(1 2)` matches the first but not the second, `(1 3)` matches the second but not the first, `(1 (>= 2))` matches both, and `(and (1 (>= 3)) (not (1 3 1)))` matches neither.

When a library reference identifies more than one available library, one of the available libraries is selected in some implementation-dependent manner.

Libraries and top-level programs should not, directly or indirectly, specify the import of two libraries that have the same names but different versions. To avoid problems such as incompatible types and replicated state, implementations are encouraged, though not required, to prohibit programs from importing two versions of the same library.

A *library-body* contains definitions of exported identifiers, definitions of identifiers not intended for export, and initialization expressions. It consists of a (possibly empty) sequence of definitions followed by a (possibly empty) sequence of initialization expressions. When `begin`, `let-syntax`, or `letrec-syntax` forms occur in a library body prior to the first expression, they are spliced into the body. Any body form may be produced by a syntactic extension, including definitions, the splicing forms just mentioned, or initialization expressions. A library body is expanded in the same manner as a `lambda` or other body (page 292), and it expands into the equivalent of a `letrec*` form so that the definitions and initialization forms in the body are evaluated from left to right.

Each of the exports listed in a library's `export` form must either be imported from another library or defined within the *library-body*, in either case with the internal rather than the export name, if the two differ.

Each identifier imported into or defined within a library must have exactly one binding. If imported into a library, it must not be defined in the library body, and if defined in the library body, it must be defined only once. If imported from two libraries, it must have the same binding in both cases, which can happen only if the binding originates in one of the two libraries and is reexported by the other or if the binding originates in a third library and is reexported by both.

The identifiers defined within a library and not exported by the library are not visible in code that appears outside of the library. A syntactic extension defined within a library may, however, expand into a reference to such an identifier, so that the expanded code does contain a reference to the identifier; this is referred to as an *indirect export*.

The exported variables of a library are *immutable* both inside the library and outside, whether they are explicitly or implicitly exported. It is a syntax violation if an explicitly exported variable appears on the left-hand side of a `set!` expression within or outside of the exporting library. It is also a syntax violation if any other variable defined by a library appears on the left-hand side of a `set!` expression and is indirectly exported.

Libraries are loaded and the code contained within them evaluated on an "as needed" basis by the implementation, as determined by the import relationships among libraries. A library's transformer expressions (the expressions on the right-hand sides of a library body's `define-syntax` forms) may be evaluated at different times from the library's body expressions (the expressions on the right-hand side of the body's `define` forms, plus initialization expressions). At a minimum, the transformer expressions of a library must be evaluated when (if not before) a reference to one of the library's exported keywords is found while expanding another library or top-level program, and the body expressions must be evaluated when (if not before) a reference to one of the library's exported variables is evaluated, which may occur either when a program using the library is run or when another library or top-level program is being expanded, if the

reference is evaluated by a transformer called during the expansion process. An implementation may evaluate a library's transformer and body expressions as many times as it pleases in the process of expanding other libraries. In particular, it may evaluate the expressions zero times if they are not actually needed, exactly one time, or one time for each meta level of the expansion. It is generally a bad idea for the evaluation of a library's transformer or body expressions to involve externally visible side effects, e.g., popping up a window, since the time or times at which these side effects occur is unspecified. Localized effects that affect only the library's initialization, e.g., to create a table used by the library, are generally okay.

Examples are given in Section 10.4.

Section 10.3. Top-Level Programs

A top-level program is not a syntactic form per se but rather a set of forms that are usually delimited only by file boundaries. Top-level programs can be thought of as library forms without the `library` wrapper, library name, and export form. The other difference is that definitions and expressions can be intermixed within the body of a top-level program but not within the body of a library. Thus the syntax of a top-level program is, simply, an `import` form followed by a sequence of definitions and expressions:

```
(import import-spec ...)
definition-or-expression
...
```

An expression that appears within a top-level program body before one or more definitions is treated as if it appeared on the right-hand side of a definition for a dummy variable that is not visible anywhere within the program.

procedure: `(command-line)`
returns: a list of strings representing command-line arguments
libraries: `(rnrs programs)`, `(rnrs)`

This procedure may be used within a top-level program to obtain a list of the command-line arguments passed to the program.

procedure: `(exit)`
procedure: `(exit obj)`
returns: does not return
libraries: `(rnrs programs)`, `(rnrs)`

This procedure may be used to exit from a top-level program to the operating system. If no *obj* is given, the exit value returned to the operating system should indicate a normal exit. If *obj* is false, the exit value returned to the operating system should indicate an abnormal exit. Otherwise, *obj* is translated into an exit value as appropriate for the operating system.

Section 10.4. Examples

The example below demonstrates several features of the `library` syntax. It defines "Version 1" of the `(list-tools setops)` library, which exports two keywords and several variables. The library imports the `(rnrs base)` library, which provides everything it needs except the `member` procedure, which it imports from `(rnrs lists)`. Most of the variables exported by the library are bound to procedures, which is typical.

The syntactic extension `set` expands into a reference to the variable `list->set`, and `member?` similarly expands into a reference to the variable `$member?`. While `list->set` is explicitly exported, `$member?` is not. This makes `$member?` an indirect export. The procedure `u-d-help` is not explicitly exported, and since neither of the exported syntactic extensions expand into references to `u-d-help`, it is not indirectly exported either. This means it could be assigned, but it is not assigned in this example.

```
(library (list-tools setops (1))
  (export set empty-set empty-set? list->set set->list
```

```
        union intersection difference member?)
  (import (rnrs base) (only (rnrs lists) member))

  (define-syntax set
    (syntax-rules ()
      [(_ x ...)
       (list->set (list x ...))]))

  (define empty-set '())

  (define empty-set? null?)

  (define list->set
    (lambda (ls)
      (cond
        [(null? ls) '()]
        [(member (car ls) (cdr ls)) (list->set (cdr ls))]
        [else (cons (car ls) (list->set (cdr ls)))])))

  (define set->list (lambda (set) set))

  (define u-d-help
    (lambda (s1 s2 ans)
      (let f ([s1 s1])
        (cond
          [(null? s1) ans]
          [(member? (car s1) s2) (f (cdr s1))]
          [else (cons (car s1) (f (cdr s1)))]))))

  (define union
    (lambda (s1 s2)
      (u-d-help s1 s2 s2)))

  (define intersection
    (lambda (s1 s2)
      (cond
        [(null? s1) '()]
        [(member? (car s1) s2)
         (cons (car s1) (intersection (cdr s1) s2))]
        [else (intersection (cdr s1) s2)])))

  (define difference
    (lambda (s1 s2)
      (u-d-help s1 s2 '())))

  (define member-help?
    (lambda (x s)
      (and (member x s) #t)))

  (define-syntax member?
    (syntax-rules ()
      [(_ elt-expr set-expr)
       (let ([x elt-expr] [s set-expr])
         (and (not (null? s)) (member-help? x s)))])))
```

The next library, `(more-setops)`, defines a few additional set operations in terms of the `(list-tools setops)` operations. No version is included in the library reference to `(list-tools setops)`; this is equivalent to an empty version reference, which matches any version. The `quoted-set` keyword is interesting because its transformer references `list->set` from `(list-tools setops)` at expansion time. As a result, if another library or top-level program that imports from `(more-setops)` references `quoted-set`, the run-time expressions of the `(list-tools setops)` library will have to be evaluated when the other library or top-level program is expanded. On the other hand, the run-time expressions of the `(list-tools setops)` library need not be evaluated when the `(more-setops)` library is itself expanded.

```
(library (more-setops)
  (export quoted-set set-cons set-remove)
  (import (list-tools setops) (rnrs base) (rnrs syntax-case))
```

```
  (define-syntax quoted-set
    (lambda (x)
      (syntax-case x ()
        [(k elt ...)
         #`(quote
             #,(datum->syntax #'k
                 (list->set
                   (syntax->datum #'(elt ...))))))])))

  (define set-cons
    (lambda (opt optset)
      (union (set opt) optset)))

  (define set-remove
    (lambda (opt optset)
      (difference optset (set opt)))))
```

If the implementation does not automatically infer when bindings need to be made available, the `import` form in the `(more-setops)` library must be modified to specify at which meta levels the bindings it imports are used via the `for` *`import-spec`* syntax as follows:

```
(import
  (for (list-tools setops) expand run)
  (for (rnrs base) expand run)
  (for (rnrs syntax-case) expand))
```

To complete the example, the short top-level program below exercises several of the `(list-tools setops)` and `(more-setops)` exports.

```
(import (list-tools setops) (more-setops) (rnrs))
(define-syntax pr
  (syntax-rules ()
    [(_ obj)
     (begin
       (write 'obj)
       (display " ;=> ")
       (write obj)
       (newline))]))
(define get-set1
  (lambda ()
    (quoted-set a b c d)))
(define set1 (get-set1))
(define set2 (quoted-set a c e))

(pr (list set1 set2))
(pr (eq? (get-set1) (get-set1)))
(pr (eq? (get-set1) (set 'a 'b 'c 'd)))
(pr (union set1 set2))
(pr (intersection set1 set2))
(pr (difference set1 set2))
(pr (set-cons 'a set2))
(pr (set-cons 'b set2))
(pr (set-remove 'a set2))
```

What running this program should print is left as an exercise for the reader.

Additional library and top-level program examples are given in Chapter 12.

ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
to order this book / about this book

http://www.scheme.com