Larceny User Manual

Table of Contents

1. Larceny

Larceny implements the Scheme programming language as defined by the Revised^7 Report, the Revised^6 Report, the Revised^5 Report, and IEEE Standard 1178-1990. Those language standards serve as Larceny's primary documentation.

This manual describes aspects of Larceny that are not described by the Revised Reports or IEEE-1178. For the most current version of this manual, please see Larceny's online documentation page [http://larceny.ccs.neu.edu/doc]. For links to the Common Larceny User Manual and the Larceny mailing list, please visit Larceny's main web page [http://www.larcenists.org/].

To report bugs, please send email to the Larceny developers at `<larceny@ccs.neu.edu>`, or submit a bug ticket at Larceny's GitHub site [https://github.com/larcenists/larceny].

2. Installing Larceny

2.1. Varieties of Larceny

There are two main varieties of Larceny.

Native Larceny is the fastest and most convenient variety of Larceny. It compiles directly to native machine code for Intel x86 microprocessors running Linux, Apple OS X, or Windows operating systems.

Petit Larceny compiles to C instead of machine code. It can be made to run on most Unix machines.

2.2. System requirements

Binary distributions of native Larceny are available for just about any Intel x86-compatible microprocessor running a Linux, Apple OS X, or Windows operating system. Although Larceny still uses 32-bit pointers, it will run on 64-bit machines provided the appropriate 32-bit libraries have been installed.

Binary distributions of Petit Larceny are available for x86 machines running Linux. Petit Larceny requires the `gcc` compiler as well as the appropriate 32-bit libraries.

For more details, see `doc/HOWTO-INSTALL` [http://larceny.ccs.neu.edu/doc/HOWTO-INSTALL]. If you want to build Larceny or Petit Larceny from source code, see `doc/HOWTO-BUILD` [http://larceny.ccs.neu.edu/doc/HOWTO-BUILD].

2.3. Downloading

The current versions of Larceny are available for download at Larceny's main web page [http://www.larcenists.org/].

Larceny is distributed in two forms: as a precompiled binary, or as source code that can be used to reconstruct any of the precompiled binary distributions. Unless you intend to modify Larceny yourself, you do not need to download the source code.

2.4. Installing the programs

Unpack the distribution files with an appropriate command such as one of the following, substituting the version number (such as 0.98) for "X.Y":

```
tar -xzf larceny-X.Y-bin-native-ia32-linux86.tar.gz
tar -xzf larceny-X.Y-bin-native-ia32-macosx.tar.gz
tar -xzf larceny-X.Y-bin-native-ia32-win32.tar.gz
tar -xzf larceny-X.Y-bin-petit-stdc-macosx.tar.gz
tar -xzf larceny-X.Y-src.tar.gz
```

That will create a directory with a similar name (but without the `.tar.gz` suffix) in your current working directory. That is the Larceny root directory, which you may rename to something shorter, such as larceny; the rest of this section will refer to it by that name.

Assuming you have unpacked a binary distribution for Linux or OS X, the larceny directory will contain the following files:

```
larceny.bin          Run-time system
larceny.heap         Heap image with preloaded libraries and compiler
larceny              Shell script that runs the two files listed above
scheme-script        Shell script that runs Scheme scripts
compile-stale        Scheme script that compiles R7RS/R6RS libraries
startup.sch          Pathnames for the autoload and require features
```

If you unpacked a binary distribution, then you should be able to run it immediately by making the larceny directory your current working directory and invoking ./larceny. (If that does not work, you may need to install some 32-bit libraries on your machine. See doc/HOWTO-INSTALL [http://larceny.ccs.neu.edu/doc/HOWTO-INSTALL].)

Binary distributions for Windows will include a larceny.bat file in addition to the files listed above, so you can run Larceny by invoking larceny. (If that does not work, you may need to tell the DEP feature [http://www.thewindowsclub.com/turn-off-data-execution-prevention-dep] to let Larceny opt out.)

If you unpacked the source code there will be many other files and directories, but larceny.bin and larceny.heap will not be present.

Tip

You can reconstruct the larceny.bin and larceny.heap files from their source code, but that process requires a working version of Larceny. Unless you're porting Larceny or Petit Larceny to a brand new target architecture, it's easier to obtain those files from a binary distribution of Larceny.

You may add the larceny directory to your standard path, or you may install Larceny into a directory that is already part of your standard path.

Suppose, for example, that you want to install Larceny in /usr/local/bin and /usr/local/lib/larceny. Copy the larceny and scheme-script files to /usr/local/bin and edit the definition of LARCENY_ROOT at the head of each file to point to the correct directory:

```
LARCENY_ROOT=/usr/local/lib/larceny
```

Then move the entire larceny directory to /usr/local/lib/larceny.

You should now be able to run Larceny from any directory by typing "larceny" at a prompt.

2.5. Compiling the R7RS/R6RS standard libraries

If you are installing Petit Larceny, then you will have to compile the R7RS/R6RS runtime and standard libraries before you can run Larceny in R7RS or R6RS modes.

This step is also required if you are building any variety of Larceny from source code. With the prebuilt native varieties of Larceny, however, this step should not be necessary unless you change one of the files in lib/R7RS, lib/R6RS, or lib/SRFI.

Tip

If the lib/R7RS, lib/R6RS, and lib/SRFI directories are read-only, you will be less likely to touch, modify, or compile the standard libraries by accident.

Compiling the R7RS/R6RS runtime and standard libraries is accomplished as follows:

```
    $ ./larceny
    Larceny v0.98 "General Ripper" (...)
    > (require 'r7rsmode)
    > (larceny:compile-r7rs-runtime)
    > (exit)
```

Warning

Compiling the R7RS/R6RS runtime as shown above causes all previously compiled R7RS/R6RS libraries and top-level programs to become stale. That means those previously compiled files will need to be recompiled or removed.

3. Running Larceny

Larceny can run in any of these distinct modes:

```
R5RS                traditional read/eval/print loop (the default)
R6RS                batch execution of R6RS top-level programs
R7RS                R7RS read/eval/print loop or batch execution
Scheme script       batch execution of R7RS/R6RS Scheme scripts
```

R5RS mode extends the Scheme language described by the R5RS and IEEE/ANSI Std 1178 by adding R7RS/R6RS lexical syntax and most of the procedures described by the newer R6RS and R7RS standards.

R6RS mode is largely redundant with Larceny's R7RS mode (because every reasonable R6RS library and program could just as well be executed in R7RS mode). There is only one difference between those two modes: R6RS mode enforces the R6RS mandates that, among other things, forbid read/eval/print loops and most extensions to R6RS lexical syntax.

R7RS mode will accept any combination of R7RS and R6RS libraries and programs. In Larceny, R6RS Scheme becomes a proper subset of R7RS Scheme.

Scheme scripts are directly executable R7RS/R6RS programs.

3.1. R5RS mode

When you start Larceny in R5RS mode (the default), you will be presented with a banner message and the read-eval-print loop's prompt:

```
    % larceny
    Larceny vX.Y "<version_name>" (MMM DD YYYY HH:MM:SS, ...)
    larceny.heap, built ...

    >
```

You can enter a Scheme expression at the prompt. After a complete expression has been read, it will be evaluated and its results printed.

Note

In native Larceny, the expression is evaluated by compiling it to native machine code, which is then executed. In Petit Larceny, the expression is evaluated by an interpreter because compiling to C, running the C compiler, and loading the compiled C code would take too long. Interpreted code behaves like compiled code, so most of what this manual says about the compiler is also true of Petit Larceny's interpreter.

3.2. R6RS mode

To execute a top-level R6RS program that is contained within a file named `pgm`, type:

```
    larceny -r6rs -program pgm
```

The `-program` option can be omitted, in which case Larceny will read the top-level program from standard input:

```
    larceny -r6rs < pgm
```

If you omit the `-program` option and do not redirect standard input, then Larceny will wait patiently for you to type a complete top-level program into standard input, terminating it with an end-of-file.

You probably don't want to do that. Had you wanted to type R6RS code at Larceny, you'd be using Larceny's R7RS read/eval/print loop instead.

3.3. R7RS mode

To execute a top-level R7RS/R6RS program that is contained within a file named `pgm`, type:

```
    larceny -r7rs -program pgm
```

To interact with Larceny's R7RS read/eval/print loop, omit the `-program` option:

```
    % larceny -r7rs
    Larceny v0.98 "General Ripper" (...)
```

The `(scheme base)` library has already been imported, but you may want to import other libraries as well. For example:

```
    > (import (scheme read)
              (scheme write)
              (scheme file)
              (scheme cxr)
              (scheme inexact)
              (scheme complex)
              (scheme char)
              (scheme load))
```

If you'd rather have Larceny import all of the standard R7RS and R6RS libraries at startup, along with a few Larceny-specific procedures, you can use the `-r7r6` option instead of `r7rs`:

```
    % larceny -r7r6
    Larceny v0.98 "General Ripper" (...)
```

Using the `-r7r6` option is equivalent to using the `-r7rs` option and then importing the `(larceny r7r6)` library.

Note

One name conflict could not be resolved by adding R7RS extensions to the conflicting R6RS procedure or syntax. When the `-r7r6` option is used, the `bytevector-copy!` procedure is imported with R7RS semantics, and the older R6RS version of that procedure is renamed to `r6rs:bytevector-copy!`.

The `features` procedure will return a list of all `cond-expand` features and libraries available to you. That procedure reads the source code for all library files found in your current Larceny library path, so don't be surprised if it takes a few seconds.

```
    > (features)
    (r7rs r6rs larceny larceny-0.98
     exact-closed ratios exact-complex complex ieee-float
     full-unicode full-unicode-strings unicode-7
     posix unix gnu-linux i386 ilp32 little-endian
     ...
     (rnrs arithmetic bitwise (6))
     (rnrs arithmetic fixnums (6))
     (rnrs arithmetic flonums (6))
     (rnrs bytevectors (6))
     ...
     (rnrs (6))
     (scheme base)
     (scheme case-lambda)
     (scheme char)
     (scheme complex)
     (scheme cxr)
     (scheme eval)
     (scheme file)
     (scheme inexact)
     (scheme lazy)
     (scheme load)
     (scheme process-context)
     (scheme r5rs)
     (scheme read)
     (scheme repl)
     (scheme time)
     (scheme write)
     (srfi 1)
     (srfi 1 lists)
     ...)
```

3.3.1. Automatic loading

As an extension to the R7RS and R6RS, Larceny attempts to load libraries automatically when they are first imported. Autoloading makes interactive development and separate compilation much more convenient.

All of Larceny's predefined libraries can be autoloaded.

To enable autoloading of other R7RS/R6RS libraries, including libraries you've written yourself, you can:

- use the `-path` command-line option
- use the `LARCENY_LIBPATH` environment variable
- use `current-require-path`

- edit `startup.sch` in Larceny's root directory
- add the libraries to Larceny's `lib` directory

3.3.2. Explicit loading

Larceny automatically loads R7RS/R6RS libraries when they are first imported. This is usually the most convenient way to load a library, but autoloading can't be used to load a top-level program. Explicit loading is needed for top-level programs, for libraries that don't reside in Larceny's `current-require-path`, and for libraries that are defined in files whose names do not follow Larceny's standard naming conventions.

In theory, explicit loading is the only portable way for R7RS programs to load a library. There is no portable way for R6RS programs to load or import libraries, so R6RS programs must rely upon implementation-specific mechanisms such as Larceny's autoloading.

For explicit loading of nonstandard libraries, top-level programs, or unadorned R5RS-style code from a file, you must first import a suitable load procedure:

```
    > (import (scheme load))
```

Loading a library does not automatically import it. To use the variables and syntax that are exported by a library, you must import that library explicitly:

```
    > (load "lib/R6RS/larceny/benchmarking.sls")
    > (import (larceny benchmarking))
    > (time (vector-for-each + (make-vector 1000000 0)))
    Words allocated: 3053286
    Elapsed time...: 25 ms (User: 25 ms; System: 0 ms)
    Elapsed GC time: 3 ms (CPU: 2 in 2 collections (1 minor).)
```

In Larceny, you may omit the call to `load` because the `(larceny benchmarking)` library will be autoloaded when it is imported. In other implementations of the R7RS, you may have to load all of the nonstandard libraries that will be imported by a top-level program or library before you load that top-level program or library.

3.3.3. Predefined libraries

Larceny predefines several nonstandard libraries in addition to the standard R7RS and R6RS libraries, and autoloads them for your convenience. The predefined, autoloadable libraries include:

Composite library:

```
(larceny r7r6)                         ; all R7RS/R6RS standard libraries
```

R7RS standard libraries:

```
(scheme base)
(scheme case-lambda)
(scheme char)
(scheme complex)
(scheme cxr)
(scheme eval)
(scheme file)
(scheme inexact)
```

```
(scheme lazy)
(scheme load)
(scheme process-context)
(scheme r5rs)
(scheme read)
(scheme repl)
(scheme time)
(scheme write)
```

R6RS standard libraries:

```
(rnrs base (6))                       ; R6RS chapter 9
(rnrs unicode (6))                    ; R6RS library chapter 1
(rnrs bytevectors (6))                ; R6RS library chapter 2
(rnrs lists (6))                      ; R6RS library chapter 3
(rnrs sorting (6))                    ; R6RS library chapter 4
(rnrs control (6))                    ; R6RS library chapter 5
(rnrs exceptions (6))                 ; R6RS library section 7.1
(rnrs conditions (6))                 ; R6RS library sections 7.2 and 7.3
(rnrs io ports (6))                   ; R6RS library sections 8.1 and 8.2
(rnrs io simple (6))                  ; R6RS library sections 8.1 and 8.3
(rnrs files (6))                      ; R6RS library chapter 9
(rnrs programs (6))                   ; R6RS library chapter 10
(rnrs arithmetic fixnums (6))         ; R6RS library section 11.2
(rnrs arithmetic flonums (6))         ; R6RS library section 11.3
(rnrs arithmetic bitwise (6))         ; R6RS library section 11.4
(rnrs syntax-case (6))                ; R6RS library chapter 12
(rnrs hashtables (6))                 ; R6RS library chapter 13
(rnrs enums)                          ; R6RS library chapter 14
(rnrs (6))                            ; R6RS library chapter 15
(rnrs eval (6))                       ; R6RS library chapter 16
(rnrs mutable-pairs (6))              ; R6RS library chapter 17
(rnrs mutable-strings (6))            ; R6RS library chapter 18
(rnrs r5rs (6))                       ; R6RS library chapter 19
```

R6RS standard libraries that are autoloadable but deprecated in Larceny because they have been superseded by the R7RS and SRFI 99 record facilities:

```
(rnrs records procedural (6))         ; R6RS library section 6.3
(rnrs records inspection (6))         ; R6RS library section 6.4
(rnrs records syntactic (6))          ; R6RS library section 6.2
```

SRFI libraries:

```
(srfi 1 lists)                        ; list library
(srfi 2 and-let*)                     ; extended `and` and `let*`
(srfi 5 let)                          ; extended version of `let`
(srfi 6 basic-string-ports)           ; basic string ports
(srfi 8 receive)                      ; binding to multiple values
(srfi 9 records)                      ; defining record types
(srfi 11 let-values)                  ; syntax for multiple values
(srfi 13 strings)                     ; string libraries
(srfi 14 char-sets)                   ; character-set library (default)
(srfi 14 unicode)                     ;   for all Unicode characters
(srfi 14 bmp)                         ;   for the Basic Multilingual Plane
(srfi 14 latin-1)                     ;   for ISO 8859-1 (Latin-1)
(srfi 16 case-lambda)                 ; syntax for variable arity
(srfi 17 generalized-set!)            ; generalized set!
(srfi 19 time)                        ; time data types and procedures
(srfi 23 error)                       ; error reporting mechanism
(srfi 25 multi-dimensional-arrays)    ; multi-dimensional array primitives
(srfi 26 cut)                         ; specializing without currying
(srfi 27 random-bits)                 ; sources of random bits
(srfi 28 basic-format-strings)        ; basic format strings
(srfi 29 localization)                ; localization
(srfi 38 with-shared-structure)       ; i/o for data with shared structure
```

```
(srfi 39 parameters)                   ; parameter objects
(srfi 41 streams)                      ; streams
(srfi 42 eager-comprehensions)         ; eager comprehensions
(srfi 43 vectors)                      ; vector library
(srfi 45 lazy)                         ; iterative lazy algorithms
(srfi 48 intermediate-format-strings)  ; format
(srfi 51 rest-values)                  ; rest values hackery
(srfi 54 cat)                          ; still more formatting
(srfi 59 vicinities)                   ; vicinity
(srfi 61 cond)                         ; a more general cond clause
(srfi 63 arrays)                       ; homogeneous, heterogeneous arrays
(srfi 64 testing)                      ; an API for test suites
(srfi 67 compare-procedures)           ; three-way comparison procedures
(srfi 78 lightweight-testing)          ; lightweight testing
(srfi 87 case)                         ; a more general case clause
(srfi 98 os-environment-variables)     ; environment variables
(srfi 99 records)                      ; (composite library)
(srfi 99 records procedural)           ; (procedural API)
(srfi 99 records inspection)           ; (inspection API)
(srfi 99 records syntactic)            ; (syntactic API)
(srfi 101 random-access-lists)         ; fast and purely functional lists
(srfi 111 boxes)                       ; boxes
(srfi 112)                             ; environment inquiry
(srfi 113 sets)                        ; sets and bags
(srfi 114 comparators)                 ; comparators
(srfi 115 regexp)                      ; regular expressions
(srfi 116 ilists)                      ; immutable lists
```

Note

For backward compatibility, `(srfi 1 lists)` through `(srfi 101 random-access-lists)` are also available with the SRFI 97 naming convention in which the number is preceded by a colon, as in `(srfi :1 lists)`. With the more liberal R7RS syntax, that SRFI 97 naming convention is now unnecessary. Larceny has extended the R6RS `library` syntax to allow R6RS libraries to import R7RS libraries that follow the R7RS naming convention shown in the list above.

SRFI libraries that are autoloadable but deprecated in Larceny, usually because they have been superseded in whole or in part by R6RS syntax or libraries:

```
(srfi 60 integer-bits)                 ; integers as bits
(srfi 66 octet-vectors)                ; octet vectors
(srfi 69 basic-hash-tables)            ; basic hash tables
(srfi 71 let)                          ; extensions of let, let*, letrec
(srfi 74 blobs)                        ; octet-addressed binary blocks
(srfi 95 sorting-and-merging)          ; sorting and merging
```

ERR5RS libraries that are autoloadable but deprecated in Larceny because they have been superseded by the R7RS and SRFI 99 record facilities:

```
(err5rs records procedural)            ; ERR5RS records (procedural API)
(err5rs records inspection)            ; ERR5RS records (inspection API)
(err5rs records syntactic)             ; ERR5RS records (syntactic API)
(err5rs load)                          ; ERR5RS load procedure
```

Other autoloadable libraries:

```
(larceny load)                         ; extension of (err5rs load)
(larceny compiler)                     ; separate compilation (R7RS/R6RS)
(larceny benchmarking)                 ; timing facilities
(larceny profiling)                    ; profiling of Scheme code
(larceny r7r6)                         ; all R7RS/R6RS standard libraries
(larceny records printer)              ; custom printing of records
(larceny shivers-syntax)               ; syntax favored by Olin Shivers
```

```
(r5rs)                                  ; approximates the R5RS top level
(explicit-renaming)                     ; macros with explicit renaming
```

3.3.4. Library path

Larceny's autoload feature locates R7RS/R6RS libraries by performing a depth-first search of the directories that belong to Larceny's `current-require-path`. Libraries will not be autoloaded unless they are defined in files whose names follow Larceny's standard conventions.

The `current-require-path` is initialized by the `startup.sch` file in Larceny's root directory.

Larceny's `-path` command-line option adds one or more directories to the directories in the `current-require-path`. On most systems, you can specify multiple directories by separating them with a colon; under Windows, use a semicolon as separator instead. The first directory listed will be searched first.

The `LARCENY_LIBPATH` environment variable can also be used to add one or more directories to the directories in the `current-require-path`. Multiple directories should be specified as with the `-path` option.

Tip

If you have a set of portable libraries that run under more than one implementation of the R7RS, and you want to have a special version of some of those libraries for Larceny, you can put all your portable versions in one directory and the Larceny-specific versions in another. When you run Larceny, use the `-path` option and specify the Larceny-specific directory first.

Note

The `-path` option cannot be used by Scheme scripts, because command-line options are passed along to the Scheme script without being interpreted by the `scheme-script` processor.

Warning

We emphasize that these extensions are non-portable. Other implementations of the R7RS or R6RS may not provide anything comparable to Larceny's `-path` option or `LARCENY_LIBPATH` environment variable. Even if they do, their mappings from library names to file names may be incompatible with Larceny's.

3.3.5. Defining libraries

As an extension to the R7RS and R6RS, Larceny allows a top-level program or Scheme script to define R7RS/R6RS libraries within the file that contains the top-level program or Scheme script, before the import form that begins the top-level program. These libraries must be arranged so that no library depends upon libraries that come later in the file.

Warning

We emphasize that this extension is non-portable.

3.3.6. Importing procedures from Larceny's underlying R5RS system

Any of Larceny's R5RS-mode top-level procedures can be imported into an R7RS or R6RS library or program by using an import declaration with a `primitives` clause that names the R5RS procedures to be imported. For example:

```
    (import (primitives random current-seconds
                        getenv setenv system
                        current-directory file-modification-time)
            (scheme time))
```

Warning

This feature is highly non-portable. Other implementations of the R7RS or R6RS may not even have an underlying implementation of the R5RS.

3.4. Scheme scripts

On most Unix systems (including Linux and Apple's OS X), Larceny's `scheme-script` will execute Scheme scripts as described in R6RS non-normative appendix D, with or without the optional script header. To make Scheme scripts executable in their own right, without executing `scheme-script` directly, add Larceny's root directory to your path as described in `doc/HOWTO-INSTALL`, or edit `scheme-script` to define `LARCENY_ROOT` and copy that edited `scheme-script` to a directory in your path.

Suppose, for example, that `/home/myself/hello` is an R7RS/R6RS Scheme script whose first line is the optional script header shown below:

```
#!/usr/bin/env scheme-script
```

If you do not have execute permission for this script, or Larceny's root directory is not in your path, you can still run the script from Larceny's root directory as follows:

```
    % ./scheme-script /home/myself/hello
```

If you have execute permission for the script, and Larceny's root directory is in your path, you can also run the script as follows:

```
    % /home/myself/hello
```

If, in addition, the directory that contains the script is in your path, you can run the script as follows:

```
    % hello
```

You may also pass command-line arguments to a Scheme script.

Warning

We emphasize that Scheme scripts are not portable. Scheme scripts are specified only by a non-binding appendix to the R6RS, not by the R6RS proper. Other implementations of the R7RS or R6RS may not support Scheme scripts at all, or may give them a semantics incompatible with Larceny's.

On Unix systems, standard input and output can be redirected in the usual way. In Larceny, standard input corresponds to the textual port initially returned by `current-input-port`, and standard output corresponds to the textual port initially returned by `current-output-port`.

Warning

We emphasize that redirection of standard input and output is non-portable. Other implementations of the R7RS or R6RS may not allow redirection, or may identify the standard input and output with ports other than those initially returned by `current-input-port` and `current-output-port`.

3.5. R5RS scripting

Suppose `hello.sch` contains the following R5RS code:

```
    (display "Hello world!")
    (newline)
    (exit)
```

You can run `hello.sch` as a script by executing Larceny as follows:

```
    % larceny -nobanner -- hello.sch
```

You can redirect Larceny's standard input, in which case you may want to eliminate the herald announcement and the read/eval/print loop's prompt:

```
    % larceny -nobanner -- -e "(begin (herald #f) (repl-prompt values))" \
              < hello.sch
```

For an explanation of why that works, which may suggest other creative uses of Larceny, ask for help:

```
    % larceny -help
```

3.6. Errors

In R6RS mode, which is batch-only, errors should result in an error message followed by a clean exit from the program.

If your program encounters an error in an interactive mode (R5RS or R75RS), it will enter the debugger; this is believed to be a feature.

Despite its crudity, and to some extent because of it, Larceny's debugger works at least as well with optimized compiled code as with interpreted code.

If you type a question mark at the debugger prompt, the debugger will print a help message. That message is more helpful if you understand the Twobit compiler and Larceny's internal representations and invariants, but this manual is not the place to explain those things.

The debugging context is saved so you can exit the debugger and re-enter it from the main read/eval/print loop's prompt:

```
    > (debug)
```

The debugger is pretty much a prototype; you don't need to tell us how bad it is.

3.7. Troubleshooting

3.7.1. Errors when starting Larceny

Although Larceny runs on x86-64 machines, it requires 32-bit libraries that are not always installed on Linux and MacOS X machines. If those libraries are absent, the operating system will probably give you a mysterious or misleading error message when you try to run Larceny. For example, the operating system's loader may tell you "larceny.bin not found" even though it's perfectly obvious that `larceny.bin` is present within Larceny's root directory. To install the necessary 32-bit libraries on Linux machines with x86-compatible processors, someone with superuser privileges must incant

```
    sudo apt-get install lib32z1
    sudo apt-get install libc6-i386
```

Warning

The names of those 32-bit packages have changed over time, and may change again.

For Macintosh machines, someone with administrative privileges must install the Apple Developer Command Line Tools [https://developer.apple.com/opensource/].

When attempting to run an R7RS/R6RS program, you may see a warning about "`loading source in favor of stale fasl file`", following by a long series of error messages about syntactic keywords used as a variable, ending with the kind of error you'd expect to see when a large R7RS/R6RS program is fed to a Scheme compiler that was expecting to see R5RS-compatible code. That means the R7RS/R6RS runtime and standard libraries were not installed correctly, or their source files have been touched or modified since they were last compiled. To fix the problem, recompile the R7RS standard libraries.

The precompiled binary forms of Larceny should run on most machines that use an appropriate processor and operating system, but the executable program "`larceny.bin`" may be incompatible with very old or with very new versions of the processor or operating system. If that appears to be the case, you should see whether a newer version of Larceny fixes the problem. If not, please report the problem to us at `larceny@ccs.neu.edu`. Please report success stories as well.

3.7.2. Errors when compiling the R7RS runtime

If something goes wrong while compiling the R7RS runtime, make sure you are running the copy of Larceny you think you are running and have read and write permission for `lib/R7RS`, `lib/R6RS`, `lib/SRFI`, and all their subdirectories and files. If you get an error message about something being "`expanded against a different build of this library`", then one or more of the compiled files in `lib/R7RS` or `lib/R6RS` or `lib/SRFI` or its subdirectories has gone stale. Removing all `.slfasl` files from `lib/R6RS` and `lib/SRFI` and their subdirectories will eliminate the stale file(s).

Warning

Don't remove the `.sch`, `.scm`, `.sls`, or `.sld` files.

3.7.3. Autoloading errors

If Larceny attempts to autoload an imported R7RS/R6RS library but cannot find the library, then the library may be defined in a file that doesn't follow Larceny's standard naming conventions. Another possibility is that the `-path` option was omitted or incorrect.

If an R7RS/R6RS library is recompiled, then all compiled libraries and top-level programs that depend upon it must also be recompiled. In particular, recompiling the standard R7RS runtime will invalidate all compiled libraries and top-level programs. Larceny's `compile-stale` script and the `compile-stale-libraries` procedure of `(larceny compiler)` make it convenient to recompile all of the libraries and top-level programs within any given directory in an order consistent with their dependencies.

3.7.4. Crashes

Please report all crashes with as much information is possible; a backtrace from a debugger or a core dump is ideal (but please do not mail the core dump without contacting us first). Larceny's run-time system is compiled with full debugging information by default and a debugger like GDB should be able to provide at least some clues.

3.8. Performance

By default, Larceny's Twobit compiler performs several optimizations that are fully compatible with the R7RS but may not be fully compatible with the older R6RS, R5RS, and IEEE-1178 standards.

When compiling R5RS code, Larceny's Twobit compiler normally makes several assumptions that allow it to generate faster code; for example, the compiler assumes Scheme's standard procedures will not be redefined.

To obtain strict conformance to R5RS semantics at the expense of slower code, use R5RS mode and evaluate the expression

```
    (compiler-switches 'standard)
```

To make the compiler generate faster code, you can promise not to redefine standard procedures *and* not to redefine any top-level procedure while it is running. To make this promise, evaluate

```
        (compiler-switches 'fast-safe)
```

To view the current settings of Twobit's numerous compiler switches, evaluate

```
        (compiler-switches)
```

All of Twobit's compiler switches are procedures whose setting can be changed by passing the new value of the switch as an argument.

For more information, evaluate

```
        (help)
```

Note

That `help` procedure is predefined only in R5RS mode, and some of the help information that will be printed may be irrelevant to the heap image you are using.

To alter the compiler switches from R7RS mode, or to disable certain compiler optimizations that are incompatible with the R6RS, see the section that describes the `(larceny compiler)` library.

4. Lexical syntax

Larceny's default lexical syntax extends the lexical syntax required by the R5RS, R6RS, and R7RS standards.

The R6RS forbids most lexical extensions, however, so Larceny provides several mechanisms for turning its lexical extensions on and off.

4.1. Flags

By default, Larceny recognizes several Larceny-specific flags of the form permitted by the R6RS. The flag you are most likely to encounter represents one of Larceny's unspecified values:

```
    #!unspecified
```

Certain other flags have special meanings to Larceny's `read` and `get-datum` procedures. They are described below.

4.2. Case-sensitivity

By default, Larceny is case-sensitive. This global default can be overridden by specifying `–foldcase` or `–nofoldcase` on Larceny's command line, or by changing the value of Larceny's `case-sensitive?` parameter.

The case-sensitivity of a particular textual input port is affected by reading one of the following flags from the port using the `read` or `get-datum` procedures:

```
    #!fold-case
    #!no-fold-case
```

The `#!fold-case` flag enables case-folding on data read from the port by the `read` and `get-datum` procedures, while the `#!no-fold-case` flag disables case-folding. The behavior established by one of these flags extends to the next such flag read from the port by `read` or `get-datum`.

Both `#!fold-case` and `#!no-fold-case` are treated as comments by Larceny's `read` and `get-datum` procedures. (This is a change from Larceny v0.97.)

4.3. Lexical extensions

When a port is first opened, the Larceny-specific lexical extensions that are accepted on the port are determined by Larceny's lexical parameters.

The following flags change the case-sensitivity and lexical extensions on the specific port from which they are read:

```
    #!r7rs          ; implies #!no-fold-case, enables R7RS syntax
    #!r6rs          ; implies #!no-fold-case, negates other flags
    #!r5rs          ; implies #!fold-case, enables R7RS syntax
    #!err5rs        ; enables R7RS/R6RS syntax with extensions
    #!larceny       ; implies #!no-fold-case and #!err5rs
```

All of those flags are treated as comments by Larceny's `read` and `get-datum` procedures. (This is a change from Larceny v0.97.)

Note

The `#!r6rs` flag is the only flag that *disables* lexical extensions. To disable R6RS lexical extensions when new ports are created, use the `read-r6rs-weirdness?` parameter described below.

4.4. Lexical parameters

When given no argument, these parameters return the current default for some aspects of the lexical syntax that will be accepted on newly created input ports or written to newly created output ports. When given an argument, these procedures change the default as specified by the argument.

The initial values of these parameters are determined by the `-r7r6`, `-r7rs`, `-r6rs`, or `-r5rs` options on Larceny's command line. The `-r6rs` option disables non-R6RS lexical syntax; the `-r7r6`, `-r7rs`, and `-r5rs` options allow both R7RS and R6RS syntax.

Procedure case-sensitive?

```
(case-sensitive? ) => boolean
```

```
(case-sensitive? boolean)
```

If true, newly created textual input ports behave as though they began with `!fold-case`. If false, newly created textual input ports behave as though they began with `!no-fold-case`.

Procedure read-r6rs-flags?

```
(read-r6rs-flags? ) => boolean
```

```
(read-r6rs-flags? boolean)
```

If true, allows flags other than `!r6rs` to be read from newly created ports. If false, flags other than `!r6rs` raise exceptions when read.

Procedure read-r7rs-weirdness?

```
(read-r7rs-weirdness? ) => boolean
```

```
(read-r7rs-weirdness? boolean)
```

If true, newly created textual input ports behave as though they began with `#!r7rs`, and R7RS lexical syntax will be used when writing external representations to newly created textual output ports. If false, R7RS-specific extensions to R5RS/R6RS lexical syntax may raise exceptions.

Procedure read-r6rs-weirdness?

```
(read-r6rs-weirdness? ) => boolean
```

```
(read-r6rs-weirdness? boolean)
```

If true, allows all R6RS lexical syntax on newly created ports without disabling other lexical syntax on those ports (so newly created textual input ports *do not* behave as though they began with `#!r6rs`). If false, R6RS-specific extensions to R5RS/R7RS lexical syntax may raise exceptions.

If `read-r6rs-weirdness?` is true and `read-r7rs-weirdness?` is false, then the R6RS bytevector syntax will be used when writing to newly opened textual output ports. If neither or both are true, then R7RS bytevector syntax will be used.

Procedure read-larceny-weirdness?

`(read-larceny-weirdness? ) =>` *`boolean`*

`(read-larceny-weirdness?` *`boolean`*`)`

Determines whether newly created textual ports allow Larceny's usual extensions to R5RS lexical syntax. In addition, this parameter determines whether newly created ports allow `#` as an insignificant digit, which is required by the R5RS but disallowed by the R6RS and not required by the R7RS.

Procedure read-traditional-weirdness?

`(read-traditional-weirdness? ) =>` *`boolean`*

`(read-traditional-weirdness?` *`boolean`*`)`

Determines whether newly created textual ports allow certain lexical extensions that are deprecated in Larceny.

> **Note**
>
> The semantics of `read-larceny-weirdness?` and `read-traditional-weirdness?` will change over time as deprecated misfeatures are added or dropped in response to popular demand or apathy. For the current semantics of these parameters, please consult the Larceny developers' web page that describes Larceny's lexical syntax [https://github.com/larcenists/larceny/wiki/LexicalConversion].

5. File naming conventions

5.1. Suffixes

In Larceny, file names generally follow Unix conventions, even on Windows. The following suffixes have special meanings to some components of Larceny.

`.sld`
: is the preferred suffix for files that contain libraries defined by the R7RS `define-library` syntax.

`.sls`
: is the preferred suffix for files that contain libraries defined by the R6RS `library` syntax.

`.sps`
: is the preferred suffix for files that contain R7RS/R6RS top-level programs (which consist of an `import` declaration followed by definitions and expressions).

`.scm`
: is the preferred suffix for files that contain R7RS/R5RS definitions and expressions but don't contain any `import` declarations and don't define any R7RS/R6RS libraries.

`.sch`
: is an alternative to `.scm` used by Larceny developers.

`.slfasl`
: is the suffix for files that contain the compiled form of a `.sld`, `.sls`, or `.sps` file.

`.fasl`
: is the suffix for files that contain the compiled form of R5RS source code (usually `.scm` or `.sch`).

`.mal`
: is the preferred suffix for files that contain MacScheme assembly language in symbolic form.

`.lap`
: is the suffix for files that contain MacScheme assembly language.

`.lop`
: is the suffix for files that contain machine code segments in the form expected by Larceny's heap linker.

`.heap`
: is the suffix for files that contain an executable heap image (must be combined with the `larceny.bin` runtime).

Note

In Larceny, R7RS `define-library` and R6RS `library` syntaxes are mostly interchangeable. The R6RS `for` and `meta` keywords may be needed when defining `syntax-case` macros, but the R7RS syntax is otherwise more versatile because of its `include` and `cond-expand` features. For new code, we recommend the R7RS `define-library` syntax.

Note

Although the R7RS `define-library` syntax allows `export` and `import` declarations to be placed anywhere at the top level of the syntax, it is standard practice to use only one `export` declaration per library, placed immediately following the name of the library, and to use only one `import` declaration per library, placed immediately following the `export` declaration.

Warning

Some of Larceny's compilation tools rely upon the convention described within the note above, and may not work if that convention is not followed.

Tip

An R7RS library definition may be split into two or more files, with the primary `.sld` file containing one or more `include` declarations that include `.scm` files. If `foo.sld` is the primary file, then the included file is ordinarily named `foo.body.scm` and placed within the same directory as `foo.sld`. If more than one `.scm` file is included, we recommend `foo.body1.scm`, `foo.body2.scm`, and so on. A Larceny-specific version of `foo.body2.scm` that's conditionally included using the `cond-expand` feature might be named `foo.body2.larceny.scm`.

Tip

Portable source code can be tailored to Larceny and other implementations of the R7RS by combining implementation-specific mechanisms such as Larceny's `-path` option with the `include` and `cond-expand` features of R7RS libraries.

5.2. Directories

Larceny's root directory should contain the following files:

```
    larceny
    scheme-script
    larceny.bin
    larceny.heap
    startup.sch
```

The following subdirectories are also essential for correct operation of some features of some modes in some varieties of Larceny:

```
    include
    lib
    lib/Base
    lib/Debugger
    lib/Ffi
    lib/MzScheme
    lib/R6RS
    lib/SRFI
    lib/Standard
    lib/TeachPacks
```

The `include` subdirectory is used when compiling files with Petit Larceny.

The `startup.sch` file tells Larceny's `require` procedure to search some of the `lib` subdirectories for libraries that are loaded dynamically.

5.3. Resolving references to libraries

The R7RS and R6RS standards do not specify any mapping from library names to files or other locations at which the code for a library might be found.

R6RS non-normative appendix E emphasizes the arbitrariness of such mappings:

> Implementations may take radically different approaches to storing source code for libraries, among them: files in the file system where each file contains an arbitrary number of library forms, files in the file system where each file contains exactly one library form, records in a database, and data structures in memory.
>
> Similarly, programs and scripts may be stored in a variety of formats. Platform constraints may restrict the choices available to an implementation, which is why the report neither mandates nor recommends a specific method for storage.
>
> Implementations may provide a means for importing libraries….
>
> Similarly, implementations may provide a means for executing a program represented as a UTF-8 text file containing its source code….
>
> —

To put it more starkly:

Warning

Although implementations of the R6RS *may* "provide a means for importing libraries" or "executing a program", they don't have to.

R7RS section 5.1 urges implementations to be reasonable:

> Implementations which store libraries in files should document the mapping from the name of a library to its location in the file system.
>
> —

Fortunately, *de facto* standards have been emerging. Larceny supports those *de facto* standards by providing these Larceny-specific mechanisms:

1. R7RS/R6RS standard libraries may be imported. Their code is located automagically.
2. Nonstandard libraries, such as `(larceny compiler)`, may be placed in one of the directories searched by Larceny's autoload feature, provided those libraries are located in files that follow Larceny's standard naming conventions as described in the next section.
3. R7RS/R6RS top-level programs may use Larceny's `-path` option to specify directories that contain other libraries the program may import, provided those libraries are located in files that follow Larceny's standard naming conventions as described in the next section.
4. R7RS/R6RS top-level programs may use Larceny's `LARCENY_LIBPATH` environment variable to specify directories that contain other libraries the program may import, provided those libraries are located in files that follow Larceny's standard naming conventions as described in the next section.
5. R7RS/R6RS top-level programs and Scheme scripts may define their own libraries in the same file that contains the top-level program or Scheme script.

R7RS programs may use any of those five mechanisms, and may also use a sixth mechanism: An R7RS program can be written as a little configuration program that loads the program's libraries from files before any libraries are imported. This sixth mechanism appears to be portable, but is not available to R6RS programs executing in Larceny's R6RS mode because it mixes execution with macro expansion, which is explicitly forbidden by one of the R6RS standard's "absolute requirements".

5.4. Mapping library names to files (R7RS/R6RS)

Suppose Larceny's `-path` option is used to specify a certain *directory*, and the program imports a nonstandard library whose name is of the form `(`*`name1 name2 … lastname`*`)`. Larceny will search for that library in the following files:

- *`directory/name1/name2/…/lastname`*`.larceny.slfasl`
- *`directory/name1/name2/…/lastname`*`.larceny.sld`
- *`directory/name1/name2/…/lastname`*`.larceny.sls`

- `*directory/name1/name2/…/lastname*.slfasl`
- `*directory/name1/name2/…/lastname*.sld`
- `*directory/name1/name2/…/lastname*.sls`
- …
- `*directory/name1/name2*.larceny.slfasl`
- `*directory/name1/name2*.larceny.sld`
- `*directory/name1/name2*.larceny.sls`
- `*directory/name1/name2*.slfasl`
- `*directory/name1/name2*.sld`
- `*directory/name1/name2*.sls`
- `*directory/name1*.larceny.slfasl`
- `*directory/name1*.larceny.sld`
- `*directory/name1*.larceny.sls`
- `*directory/name1*.slfasl`
- `*directory/name1*.sld`
- `*directory/name1*.sls`

The search starts with the first of those file names, continues with the following file names in order, and ends when a file with one of those names is found. The imported library *must* be one of the libraries defined within the first file found by this search, since the search is not continued after that first file is found (except as noted in the next paragraph).

If the search ends by finding a file whose name ends with `.slfasl`, then Larceny checks to see whether there is a file in the same directory with the same root name but ending with `.sld` or `.sls` instead of `.slfasl`. If the `.sld` or `.sls` file has been modified since the `.slfasl` file was last modified, then a warning is printed and the `.sld` or `.sls` file is loaded instead of the `.slfasl` file. Otherwise the `.slfasl` file is loaded.

Note

The R6RS allows arbitrary mappings from library names to library code. Larceny takes advantage of this by ignoring version numbers when mapping library names to files, and by (virtually) rewriting any version number that may be specified in the definition of a library so it matches any version specification that appears within the `import` form. Furthermore Larceny allows different versions of the same library to be imported, but Larceny's algorithm for resolving library references ensures that the different versions of a library will be identical except for their version numbers, which have no meaningful semantics. Although Larceny's treatment of versions conforms to the R6RS specification, it should be clear that version numbers serve no purpose in

Larceny. Since the R6RS version feature has no usefully portable semantics and has been ignored by most implementations of the R6RS, it is deprecated.

5.5. Mapping library names to files (R5RS)

In R5RS mode, Larceny's `-path` option and `LARCENY_LIBPATH` environment variable may be used to specify directories to be searched by the `require` procedure, which takes a single symbol *libname* as its argument. The `require` procedure will search for the following files in every directory that is part of the current require path, starting with the directories specified by LARCENY_LIBPATH and the `-path` option:

- *`libname`*`.fasl`
- *`libname`*`.sch`
- *`libname`*`.scm`

These files are expected to contain R5RS code, not library definitions. Otherwise the search proceeds much the same as when searching for an R7RS/R6RS library.

Note

The `require` path is specified by `startup.sch` in Larceny's root directory, but may be changed dynamically using the `current-require-path` parameter. Changing the `require` path is not recommended, however, because Larceny relies on the `require` path for dynamic loading of libraries used by several important features of Larceny, notably R7RS and R6RS modes.

Procedure require

`(require` *`libname`*`)`

libname must be a symbol that names an R5RS-compatible library within the current require path.

If the library has not already been loaded, then it is located and loaded. If the library is found and loaded successfully, then `require` returns true; otherwise an error is signalled.

If the library has already been loaded, then `require` returns false without loading the library a second time.

Procedure current-require-path

`(current-require-path ) =>` *`stringlist`*

`(current-require-path` *`stringlist`*`)`

The optional argument is a list of directory names (without slashes at the end) that should be searched by `require` and (in R7RS/R6RS modes) by Larceny's autoload feature. Returns the list of directory names that will be searched.

6. Compiling files and libraries

This chapter explains how you can use Larceny to compile Scheme source code to native machine code.

The native varieties of Larceny have a just-in-time compiler that compiles to native code automatically whenever you evaluate an expression, load a source file, or import a source library. Even so, files will load faster if they are compiled ahead of time.

Petit Larceny does not have a just-in-time compiler, so compiling ahead of time is the only way to enjoy the speed of native machine code in Petit Larceny.

The main disadvantage of compiling files and libraries is that compiled code goes *stale* when its original source code is changed or when a library on which the compiled code depends is changed or recompiled. Stale compiled code can be dangerously inconsistent with libraries on which it depends, so Larceny checks for staleness and refuses to execute a stale library or program.

6.1. Compiling R7RS/R6RS libraries

On Unix machines, the most convenient way to compile a group of R7RS/R6RS libraries and top-level programs is to use the `compile-stale` script in Larceny's root directory. If Larceny's root directory is in your execution path, then there are just two steps:

1. Use `cd` to change to the directory that contains the R7RS/R6RS files you want to compile. (Files that lie within subdirectories of that directory will be compiled also.)
2. Run the `compile-stale` script.

For example:

```
    % cd lib/R7RS
    % compile-stale
```

On non-Unix machines, you can accomplish the same thing using Larceny's R7RS mode and the `(larceny compiler)` library:

```
    % pushd lib\R7RS
    % ../../larceny -r7rs
    Larceny v0.98 "General Ripper"

    > (import (larceny compiler))

    > (compile-stale-libraries)
```

To compile individual files, use the `compile-file` or `compile-library` procedures that are exported by `(larceny compiler)`.

6.2. Compiling R5RS source files

Procedure compile-file

`(compile-file` *`sourcefile`*`)`

Compiles *sourcefile*, which must be a string naming a file that contains R5RS source code. If *faslfile* is supplied as a second argument, then it must be a string naming the file that will contain the compiled code; otherwise the name of the compiled file is obtained from *sourcefile* by replacing the "`.sch`" or "`.scm`" suffix with "`.fasl`".

For R7RS/R6RS libraries and top-level programs, see above.

7. R7RS standard libraries

The R7RS standard libraries are described by the R7RS (small) standard approved in 2013.

Larceny provides all of the R7RS standard libraries, supports the full numeric tower, and can represent all Unicode characters.

Binary releases of Larceny also support Unicode strings. (When built from source code, Larceny can be configured to use Latin-1 strings instead of Unicode.)

When Larceny is invoked with the `-r7r6` option on its command line, all of the standard R7RS and R6RS libraries are imported at startup. When invoked with the `-r7rs` option, only `(scheme base)` is imported at startup.

7.1. Known deviations from the R7RS standard

Larceny v0.98 does not implement these features of the R7RS standard:

- the second (arbitrary ellipsis) form of `syntax-rules` described in R7RS section 4.3.2
- `include` and `include-ci` at expression level
- `cond-expand` at expression level

`include`, `include-ci`, and `cond-expand` are fully supported at the top-level declaration and definition levels of R7RS libraries.

To simplify interoperability with R6RS libraries and programs, the `integer?`, `rational?`, and `real?` procedures exported by `(scheme base)` have R6RS semantics. It is not clear whether that is fully compatible with the R7RS (small) standard, because the R7RS specification of those procedures appears to contradict itself.

If any other R7RS feature is missing or incompatible with the R7RS (small) standard, it's a bug.

8. R6RS standard libraries

This chapter explains which features of the R6RS standard libraries are available in each of Larceny's major modes of execution.

Larceny was the first substantially complete implementation of the R6RS. Any features that are missing from R6RS modes are missing because of bugs or because the features are deprecated in Larceny.

Larceny is R6RS-compatible but not R6RS-conforming. When Larceny is said to support a feature of the R6RS, that means the feature is present and will behave as specified by the R6RS so long as no exception is raised or expected. Larceny does not always raise the specific conditions specified by the R6RS, and does not perform all of the checking for portability problems that is mandated by the R6RS. These deviations do not affect the execution of production code, and do not compromise Larceny's traditional safety.

For example, Larceny has extended the R6RS `library` syntax to allow R6RS libraries to import R7RS libraries even when the names of those imported libraries use the more liberal R7RS syntax.

Furthermore, Larceny has extended several R6RS procedures so they behave as specified by the newer R7RS (small) standard. In Larceny, for example, the `utf8->string` procedure accepts one, two, or three arguments, and the `finite?` procedure accepts any object as its argument. According to the R6RS, `utf8->string` *must* raise an exception when passed more than one argument, and `finite?` *must* raise an exception if it detects an argument that is not a real number. Although the R6RS says these exceptions are "absolute requirements", they interfere with interoperability between R6RS and R7RS code, and are best honored in the breach.

8.1. Base library

R7RS and R6RS modes support all procedures and syntaxes exported by the `(rnrs base)` library.

Larceny's R5RS mode does not support `library`, `import`, or `identifier-syntax`.

Note

The semantics of `quasiquote`, `let-syntax`, and `letrec-syntax` differ between the R5RS, R6RS, and R7RS. Larceny's R5RS mode still supports the R5RS semantics. R7RS and R6RS modes support the R6RS semantics.

8.2. Unicode

All of Larceny's modes support all features of the `(rnrs unicode)` library.

Larceny v0.98 tries to conform to The Unicode Standard, Version 7.0.

8.3. Bytevectors

R7RS and R6RS modes support all procedures and syntaxes exported by `(rnrs bytevectors)`, but the `endianness` syntax is deprecated because it is redundant with `quote`. Larceny's R5RS mode does not support `endianness`.

In Larceny, any symbol names a supported endianness. The symbols `big` and `little` have their expected meanings. All other symbols mean `(native-endianness)` with respect to integer operations, but mean the opposite of `(native-endianness)` with respect to IEEE-754 operations. For string operations, the endianness must be the symbol `big` or the symbol `little`. All of these extensions are permitted by the R6RS standard.

Larceny's `utf16->string` and `utf32->string` accept one, two, or three arguments. The R6RS specification of these procedures does not allow them to accept a single argument, but that is believed to be an error in the R6RS.

8.4. Lists

All of Larceny's modes support all features of the `(rnrs lists)` library.

8.5. Sorting

All of Larceny's modes support all features of the `(rnrs sorting)` library.

8.6. Control

All of Larceny's modes support all features of the `(rnrs control)` library.

8.7. Records

R7RS and R6RS modes support all procedures and syntaxes exported by `(rnrs records procedural)`, `(rnrs records inspection)`, and `(rnrs records syntactic)`.

Those libraries are deprecated, however; the `make-record-constructor-descriptor` procedure does not simplify unusually complex cases enough to justify the complexity it adds to typical cases, and the entire syntactic layer is gratuitously incompatible with the procedural layer.

Larceny extends the R7RS `define-record-type` syntax exported by `(scheme base)` to accept the deprecated R6RS syntax, and extends the deprecated `define-record-type` syntax exported by `(rnrs records syntactic)` to accept R7RS syntax. Larceny's unification of the two syntaxes within a single implementation of `define-record-type` allows libraries and programs to import both `(scheme base)` and `(rnrs)` without having to rename one version of `define-record-type`.

Larceny also extends its unified R7RS/R6RS `define-record-type` to support all features of `(srfi :99 records syntactic)`. So long as the deprecated R6RS syntax is not used, Larceny's `define-record-type` is fully compatible with the procedural layers defined by `(srfi :99 records procedural)` and by `(rnrs records procedural)`.

Larceny's R5RS mode supports all features of the deprecated `(rnrs records procedural)` and `(rnrs records inspection)` libraries. R5RS mode does not support `(rnrs records syntactic)`.

All of Larceny's modes support all features of the `(err5rs records procedural)` and `(err5rs records inspection)` libraries. R7RS and R6RS modes also support the `(err5rs records syntactic)` library. These libraries are equivalent to the `(srfi :99 records procedural)`, `(srfi :99 records inspection)`, and `(srfi :99 records syntactic)` libraries.

The record definition syntax of SRFI 9 [http://srfi.schemers.org/srfi-9/] is a proper subset of the syntax provided by the `(err5rs records syntactic)` library. In R5RS mode, SRFI 9 can be loaded dynamically using the `require` procedure:

```
    > (require 'srfi-9)
```

We recommend the R7RS and/or SRFI 9 libraries be used instead of the corresponding R6RS libraries.

Warning

The R6RS spouts some tendentious nonsense about procedural records being slower than syntactic records, but this is not true of Larceny's records, and is unlikely to be true of other implementations either.

Warning

Larceny continues to support its old-style records, which are almost but not quite compatible with R7RS and R6RS records. This can be confusing, since some of Larceny's procedures have the same names as R6RS procedures. That has made it necessary to overload those procedures to work with both old-style and R6RS records. We apologize for the mess.

8.8. Exceptions and conditions

All of Larceny's modes support all features of the `(rnrs exceptions)` and `(rnrs conditions)` libraries.

8.9. Input and output

R7RS and R6RS modes support all names exported by the `(rnrs io ports)`, `(rnrs io simple)`, and `(rnrs files)` libraries.

The `buffer-mode`, `eol-style`, and `error-handling-mode` syntaxes are deprecated because they are redundant with `quote`. Larceny may provide these deprecated syntaxes in the form of procedures rather than syntax, but this deviation from R6RS semantics cannot be detected by portable R6RS programs.

Larceny's R5RS mode supports all non-deprecated features of those libraries.

Larceny supports four distinct buffer modes: `none`, `line`, `datum`, and `block`. The R6RS requires the `buffer-mode` syntax to raise an exception for the `datum` buffer mode, which is the buffer mode Larceny uses for interactive output ports.

In Larceny, any symbol names a supported end-of-line style. All end-of-line and error-handling-mode symbols whose meanings are not described by the R6RS have locale-dependent meanings, which is an extension permitted by the R6RS standard.

Although Larceny supports the UTF-16 codec, it is not really useful on Windows machines (where it should be most useful) because Larceny's low-level file system mimics a byte-oriented Unix file system even on Windows. This problem should be addressed in some future version of Larceny.

The most up-to-date list of known deviations from R6RS io semantics can be found on the web page that describes the current status of Larceny's R6RS-compatible mode [https://github.com/larcenists/larceny/wiki/DargoMode].

8.10. Programs

R7RS and R6RS modes support the `(rnrs programs)` library.

Larceny's R5RS mode provides the `exit` procedure but not the `command-line` procedure of that library. Larceny's traditional `command-line-arguments` procedure can be used to implement an approximation to `command-line`. For a definition, see `lib/R6RS/rnrs/programs.sls`.

8.11. Arithmetic

All of Larceny's modes support all features of the `(rnrs arithmetic fixnums)`, `(rnrs arithmetic flonums)`, and `(rnrs arithmetic bitwise)` libraries.

Note

R6RS fixnum and flonum operations may be slower than the corresponding generic operations, since the fixnum and flonum operations are required to check their arguments and may also have to check their results. Isolated operations in small micro-benchmarks are likely to be slower than groups of similar operations in larger programs, however, because Larceny's compiler removes redundant checks and propagates type information.

8.12. Syntax-case

R7RS and R6RS modes support the `(rnrs syntax-case)` library. Larceny's R5RS mode does not.

8.13. Hashtables

All of Larceny's modes support all features of the `(rnrs hashtables)` library.

Note

Larceny's traditional `make-hashtable` procedure has been renamed to `make-oldstyle-hashtable`.

Note

When you use Larceny's R5RS or R7RS mode to dump a heap image that contains `eq?` or `eqv?` hashtables you have created, they are automatically reset so they will rehash themselves whenever you begin a new session with the dumped heap.

8.14. Enumeration sets

R7RS and R6RS modes support the `(rnrs enums)` library. Larceny's R5RS mode provides all of the procedures exported by `(rnrs enums)` but does not provide the `define-enumeration` syntax.

8.15. Eval

R7RS and R6RS modes support the `(rnrs eval)` library. Larceny's R5RS mode provides an R5RS-compatible eval procedure, not an R6RS-compatible eval procedure, and does not provide the `environment` procedure.

8.16. Mutable pairs and strings

All of Larceny's modes support all features of the `(rnrs mutable-pairs)` and `(rnrs mutable-strings)` libraries.

8.17. R5RS

All of Larceny's modes support all features of the `(rnrs r5rs)` library.

9. Larceny's R7RS/R6RS libraries

Larceny provides libraries for compiling R7RS/R6RS libraries and for timing benchmarks.

9.1. Load

The `(larceny load)` library exports both the `load` procedure of `(scheme load)` and Larceny's `r5rs:require` procedure, which is a renaming of the `require` procedure used by Larceny's R5RS mode.

In Larceny's R7RS mode, the `load` procedure can load R5RS libraries and programs as well as R7RS/R6RS libraries.

The `r5rs:require` procedure should be used only for dynamic loading of R5RS libraries into Larceny's underlying R5RS system. The variables defined by that library can be imported into an R7RS session or R7RS/R6RS library or program using a `primitives` clause in an `import` form.

Warning

These procedures should be used only at an interactive top level and in files that will be loaded into an interactive top level. Calls to these procedures have no effect at compile time, and should not appear in files that will be compiled separately; use the `define-library` and `import` syntaxes instead.

9.2. Compiler

The `(larceny compiler)` library exports the `load` and `r5rs:require` procedures of `(larceny load)`, the `current-require-path` procedure, the `compile-file`, `compile-library`, and `compile-stale-libraries` procedures described below, and the `compiler-switches` procedure.

These procedures can be used to compile R7RS/R6RS libraries and top-level programs before they are imported or executed. This is especially important for Petit Larceny, which would otherwise use an interpreter. For native Larceny, whose just-in-time compiler generates native machine code as source libraries and programs are loaded, imported, or executed, the main advantage of separate compilation is that compiled libraries and programs will load much faster than source libraries and programs.

The main disadvantage of separate compilation is that compiled libraries and programs go *stale* when their source code is changed or when a library on which they depend is changed or recompiled. Stale libraries and programs can be dangerously inconsistent with libraries on which they depend, so Larceny checks for staleness and refuses to execute a stale library or program. The `compile-stale-libraries` procedure provides a convenient way to recompile stale libraries and programs.

`(compile-file` *`sourcefile [slfaslfile]`*`)`

Compiles *sourcefile*, which must be a string naming a file that contains source code for one or more R7RS/R6RS libraries or a top-level program. If *slfaslfile* is supplied as a second argument, then it must be a string naming the file that will contain the compiled code; otherwise the name of the compiled file is obtained from *sourcefile* by replacing the "`.sld`" or "`.sls`" suffix with "`.slfasl`".

Procedure compile-library

`(compile-library` *`sourcefile [slfaslfile]`*`)`

Compiles *sourcefile*, which must be a string naming a file that contains source code for one or more R7RS/R6RS libraries. Apart from its unwillingness to compile top-level programs, `compile-library` behaves the same as `compile-file` above.

Procedure compile-stale-libraries

`(compile-stale-libraries )`

`(compile-stale-libraries` *`changedfile`*`)`

If no argument is supplied, then all "`.sld`" and "`.sls`" files that lie within the current directory or a subdirectory are recompiled.

If *changedfile* is supplied, then it must be a string giving the absolute pathname of a file. (In typical usage, *changedfile* is a source file that has been modified, making it necessary to recompile all files that depend upon it.) Compiles all R7RS/R6RS library files that lie within the same directory as *changedfile* or a subdirectory, and have not yet been compiled or whose compiled files are older than *changedfile*.

Note

In future versions of Larceny, `compile-stale-libraries` might compile only the source files that depend upon *changedfile*.

Procedure compiler-switches

`(compiler-switches )`

`(compiler-switches` *`mode`*`)`

If no argument is supplied, then the current settings of all compiler switches are displayed. Each of those switches is itself a parameter that is exported by the `(larceny compiler)` library. Calling any individual compiler switch with no arguments will return its current setting. Calling any individual compiler switch with an argument (usually a boolean) will change its setting to that argument.

The `compiler-switches` procedure may also be called with one of the following symbols as its argument:

`default` sets most compiler switches to their default settings.

`fast-safe` enables all optimizations but continues to generate code to perform all run-time type and range checks that are needed for safety (in the traditional sense, not the R6RS sense).

`fast-unsafe` enables all optimizations and also disables type and range checking. This setting is deprecated because it compromises safety (in the traditional sense).

`slow` turns off all optimizations.

`standard` sets compiler switches for maximal conformance to the R5RS and R6RS standards.

Warning

The `standard` setting is deprecated because it generates very slow code (because the R5RS makes it difficult to inline standard procedures), disables most compile-time checking (because the R6RS forbids rejection of programs with obvious errors unless the R6RS classifies the errors as syntactic), and may also compromise the portability or interoperability of R7RS/R6RS libraries and programs (because the R6RS outlaws several extensions that Larceny uses to improve its compatibility with other implementations of the R5RS, R6RS, and R7RS as well as interoperability between Larceny's own R5RS and R7RS/R6RS modes).

Tip

Selective toggling of compiler switches is almost always better than using the `standard` setting. To improve R5RS conformance without sacrificing too much performance, set the `benchmark-mode` switch to false and set the `integrate-procedures` switch to false only when compiling files that need to be sensitive to redefinitions of standard procedures. For R6RS libraries and programs, setting the `benchmark-mode` and `global-optimization` switches to false will eliminate a couple of minor conformance issues with only a small loss of performance and without sacrificing compile-time checking or portability. For R7RS libraries and programs, the compiler's default settings already conform to the R7RS.

9.3. Benchmarking

The `(larceny benchmarking)` library exports the `time` syntax and `run-benchmark` procedure described below.

Syntax time

`(time expression)`

Evaluates *expression* and returns its result after printing approximations to the storage allocated and time taken during evaluation of *expression*.

```
    > (time (fib 30))
    Words allocated: 0
    Words reclaimed: 0
    Elapsed time...: 49 ms (User: 48 ms; System: 0 ms)
    Elapsed GC time: 0 ms (CPU: 0 in 0 collections.)
    832040
```

`(run-benchmark` *`name iterations thunk predicate`*`)`

Given the *name* of a benchmark, the number of *iterations* to be performed, a zero-argument procedure *thunk* that runs the benchmark, and a unary *predicate* that checks the result of *thunk*, prints approximations to the storage allocated and time taken by *iterations* calls to *thunk*.

```
    > (run-benchmark "fib30"
                     100
                     (lambda () (fib 30))
                     (lambda (x) (= x 832040)))

    --------------------------------------------------------
    fib30
    Words allocated: 0
    Words reclaimed: 0
    Elapsed time...: 4828 ms (User: 4824 ms; System: 4 ms)
    Elapsed GC time: 0 ms (CPU: 0 in 0 collections.)
```

9.4. Records printer

The `(larceny records printer)` library exports the two procedures described below. These procedures can be used to override Larceny's usual printing of records and opaque types that were defined using the records libraries.

Procedure rtd-printer

```
(rtd-printer rtd) => maybe-procedure
```

Given a record type descriptor, returns its custom print procedure, or returns false if the rtd has no custom print procedure.

Procedure rtd-printer-set!

```
(rtd-printer-set! rtd printer)
```

Given a record type descriptor *rtd* and a *printer* for instances of that rtd, installs *printer* as a custom print procedure for *rtd*. The *printer* should be a procedure that, given an instance of the rtd and a textual output port, writes a representation of the instance to the port.

10. ERR5RS standard libraries

ERR5RS has been superseded by the R7RS, so the libraries described below are now deprecated.

10.1. Load

The `(err5rs load)` library has been superseded by the `(scheme load)` library, and continues to exist only for backward compatibility.

Procedure load

```
(load filename)
```

Loads ERR5RS code from *filename*, evaluating each form as though it had been entered at the interactive read/eval/print loop.

Warning

The `load` procedure should be used only at an interactive top level and in files that will be loaded into an interactive top level. Calls to the `load` procedure have no effect at compile time, and should not appear in files that will be compiled separately; use the `library` and `import` syntaxes instead.

10.2. Records

The ERR5RS record facility described below incorporates all optional features of SRFI 99 and is otherwise identical to the facilities described by SRFI 99. SRFI 99 is itself an extension of SRFI 9, whose `define-record-type` syntax is identical to that defined by the R7RS.

When a procedure is said to be equivalent to an R6RS procedure, the equivalence holds only when all arguments have the properties required of them by the R6RS specification. ERR5RS does not mandate R6RS exception semantics for programs that violate the specification.

10.2.1. Procedural layer

This section describes the `(err5rs records procedural)` library.

Procedure make-rtd

```
(make-rtd name fieldspecs)
```

```
(make-rtd name fieldspecs parent-rtd)
```

```
(make-rtd name fieldspecs parent-rtd option …)
```

name is a symbol, which matters only to the `rtd-name` procedure of the inspection layer. *fieldspecs* is a vector of field specifiers, where each field specifier is one of

- a symbol naming the (mutable) field;
- a list of the form `(mutable name)`, where *name* is a symbol naming the mutable field;
- a list of the form `(immutable name)`, where *name* is a symbol naming the immutable field.

The optional parent is an rtd or `#f`. It is an error for any of the symbols in fieldspecs to name more than one of the fields specified by fieldspecs, but the field names in fieldspecs may shadow field names in the parent rtd.

`make-rtd` returns an R6RS-compatible record-type descriptor.

Larceny allows the following optional arguments to follow the optional *parent-rtd* argument:

- the symbol `sealed` means the new rtd cannot be used as the parent of other rtds;
- the symbol `opaque` means the `record?` predicate will not recognize instances of the new rtd;
- the symbol `uid`, followed by another symbol *id*, means the new rtd is non-generative with uid *id*; the semantics of this extension is the same as in the R6RS.

These Larceny-specific options may be used in any combination, giving Larceny's ERR5RS records the same expressive power as R6RS records, with which they are fully interoperable.

Procedure rtd?

```
(rtd? obj)
```

This predicate returns true if and only if its argument is a record-type descriptor. `rtd?` is equivalent to the `record-type-descriptor?` procedure of the R6RS.

Procedure rtd-constructor

```
(rtd-constructor rtd)
```

```
(rtd-constructor rtd fieldspecs)
```

rtd is a record-type descriptor, and *fieldspecs* is an optional vector of symbols.

If no *fieldspecs* argument is supplied, then `rtd-constructor` returns a procedure that expects one argument for each field of the record-type described by *rtd* and returns an instance of that record-type with its fields initialized to the corresponding arguments. Arguments that correspond to the fields of the record-type's parent (if any) come first.

If *fieldspecs* is supplied, then `rtd-constructor` returns a procedure that expects one argument for each element of *fieldspecs* and returns an instance of the record-type described by *rtd* with the named fields initialized to the corresponding arguments.

It is an error if some symbol occurs more than once in *fieldspecs*. Fields of a derived record-type shadow fields of the same name in its parent; the *fieldspecs* argument cannot be used to initialize a shadowed field.

Procedure rtd-predicate

```
(rtd-predicate rtd)
```

Equivalent to the `record-predicate` procedure of the R6RS.

Procedure rtd-accessor

```
(rtd-accessor rtd field)
```

field is a symbol that names a field of the record-type described by the record-type descriptor *rtd*. Returns a unary procedure that accepts instances of *rtd* (or any record-type that inherits from *rtd*) and returns the current value of the named field.

Fields in derived record-types shadow fields of the same name in a parent record-type.

Procedure rtd-mutator

```
(rtd-mutator rtd field)
```

field is a symbol that names a field of the record-type described by the record-type descriptor *rtd*. Returns a binary procedure that accepts instances of *rtd* (or any record-type that inherits from *rtd*) and a new value to be stored into the named field, performs that side effect, and returns an unspecified value.

Fields in derived record-types shadow fields of the same name in a parent record-type.

10.2.2. Inspection layer

This section describes the `(err5rs records inspection)` library.

Procedure record?

```
(record? obj)
```

Equivalent to its R6RS namesake.

Procedure record-rtd

```
(record-rtd record)
```

Equivalent to its R6RS namesake.

Procedure rtd-name

`(rtd-name` *`rtd`*`)`

Equivalent to the `record-type-name` procedure of the R6RS.

Procedure rtd-parent

`(rtd-parent` *`rtd`*`)`

Equivalent to the `record-type-parent` procedure of the R6RS.

Procedure rtd-field-names

`(rtd-field-names` *`rtd`*`)`

Equivalent to the `record-type-field-names` procedure of the R6RS. (That is, it returns a vector of the symbols that name the fields of the record-type represented by *rtd*, excluding the fields of parent record-types.)

Procedure rtd-all-field-names

`(rtd-all-field-names` *`rtd`*`)`

Returns a vector of the symbols that name the fields of the record-type represented by *rtd*, including the fields of its parent record-types, if any, with the fields of parent record-types coming before the fields of its children, with each subsequence in the same order as in the vectors that would be returned by calling `rtd-field-names` on *rtd* and on all its ancestral record-type descriptors.

Procedure rtd-field-mutable?

`(rtd-field-mutable?` *`rtd field`*`)`

rtd is a record-type descriptor, and *field* is a symbol naming a field of the record-type described by *rtd*. Returns `#t` if the named field is mutable; otherwise returns `#f`.

10.2.3. Syntactic layer

This section describes the `(err5rs records syntactic)` library.

The syntactic layer consists of SRFI 9 [http://srfi.schemers.org/srfi-9/] extended with single inheritance and (optional) implicit naming.

All ERR5RS record-type definitions are generative (unless Larceny's optional `uid` feature is used), but ERR5RS drops the SRFI 9 restriction to top level, mainly because the R6RS allows generative definitions wherever a definition may appear.

The syntax of an ERR5RS record-type definition is

```
    <definition>
      -> <record type definition>           ; addition to 7.1.6 in R5RS

    <record type definition>
      -> (define-record-type <type spec>
           <constructor spec>
           <predicate spec>
           <field spec> ...)

    <type spec>  -> <type name>
```

```
                -> (<type name> <parent>)

    <constructor spec>
                -> #f
                -> #t
                -> <constructor name>
                -> (<constructor name> <field name> ...)

    <predicate spec>
                -> #f
                -> #t
                -> <predicate name>

    <field spec> -> <field name>
                -> (<field name>)
                -> (<field name> <accessor name>)
                -> (<field name> <accessor name> <mutator name>)

    <parent>           -> <expression>

    <type name>        -> <identifier>
    <constructor name> -> <identifier>
    <predicate name>   -> <identifier>
    <accessor name>    -> <identifier>
    <mutator name>     -> <identifier>
    <field name>       -> <identifier>
```

The semantics of a record type definition is the same as in SRFI 9: the record type definition macro-expands into a cluster of definitions that

- defines the `<type name>` as the record-type descriptor for the new record-type;
- defines a constructor for instances of the new record-type (unless the constructor spec is `#f`);
- defines a predicate that recognizes instances of the new record-type and its subtypes (unless the predicate spec is `#f`);
- defines an accessor for each field name;
- defines a mutator for each mutable field name.

An ERR5RS record type definition extends SRFI 9 with the following additional options:

- If a `<parent>` expression is specified, then it must evaluate to an rtd that serves as the parent record-type for the record-type being defined.
- If `#f` is specified for the constructor or predicate, then no constructor or predicate procedure is defined. (This is useful when the record-type being defined will be used as an abstract base class.)
- If `#t` is specified for the constructor or predicate, then the name of the constructor is the type name prefixed by `make-`, and the name of the predicate is the type name followed by a question mark (`?`).
- If the constructor name is specified as `#t` or as an identifier, then the constructor's arguments correspond to the fields of the parent (if any) followed by the new fields added by this record-type definition.
- If a field spec consists of a single identifier, then

    - the field is immutable;
    - the name of its accessor is the type name followed by a hyphen (`-`) followed by the field name.
- If a field spec consists of a list of one identifier, then
    - the field is mutable;
    - the name of its accessor is the type name followed by a hyphen (`-`) followed by the field name;
    - the name of its mutator is the type name followed by a hyphen (`-`) followed by the field name followed by `-set!`.

10.2.4. Record identity

Two ERR5RS records with fields are `eqv?` if and only if they were created by the same (dynamic) call to some record constructor. Two ERR5RS records are `eq?` if and only if they are `eqv?`.

Apart from the usual constraint that equivalence according to `eqv?` implies equivalence according to `equal?`, the behavior of `equal?` on ERR5RS records is unspecified. (This is compatible with the R6RS.)

A `define-record-type` form macro-expands into code that calls `make-rtd` each time the expanded record-type definition is executed. Two ERR5RS record-type descriptors are `eqv?` if and only if they were created by the same (dynamic) call to `make-rtd`.

11. Larceny's R5RS libraries

The procedures described in this chapter are nonstandard. Some are deprecated after being rendered obsolete by R7RS or R6RS standard libraries. Others still provide useful capabilities that the standard libraries don't.

11.1. Strings

Larceny provides Unicode strings with R6RS [http://www.r6rs.org/] semantics.

The `string-downcase` and `string-upcase` procedures perform Unicode-compatible case folding, which can result in a string whose length is different from that of the original.

Larceny may still provide `string-downcase!` and `string-upcase!` procedures, but they are deprecated.

11.2. Bytevectors

A *bytevector* is a data structure that stores bytes — exact 8-bit unsigned integers. Bytevectors are useful in constructing system interfaces and other low-level programming. In Larceny, many bytevector-like structures — bignums, for example — are implemented in terms of a lower-level *bytevector-like* data type. The operations on generic bytevector-like structures are particularly fast but useful largely in code that manipulates Larceny's data representations.

The `(rnrs bytevectors)` library now provides a large set of procedures that, in Larceny, are defined using the procedures described below.

Integrable procedure make-bytevector

`(make-bytevector` *`length`*`) =>` *`bytevector`*

`(make-bytevector` *`length fill`*`) =>` *`bytevector`*

Returns a bytevector of the desired length. If no second argument is given, then the bytevector has not been initialized and most likely contains garbage.

Operations on bytevector structures

`(bytevector?` *`obj`*`) =>` *`boolean`*

`(bytevector-length` *`bytevector`*`) =>` *`integer`*

`(bytevector-ref` *`bytevector offset`*`) =>` *`byte`*

`(bytevector-set!` *`bytevector offset byte`*`) =>` *`unspecified`*

`(bytevector-equal?` *`bytevector1 bytevector2`*`) =>` *`boolean`*

`(bytevector-fill!` *`bytevector byte`*`) =>` *`unspecified`*

`(bytevector-copy` *`bytevector`*`) =>` *`bytevector`*

These procedures do what you expect. All are integrable, except `bytevector-equal?` and `bytevector-copy`. The `bytevector-equal?` name is deprecated, since the R6RS calls it `bytevector=?`.

Operations on bytevector-like structures

`(bytevector-like?` *`obj`*`) =>` *`boolean`*

`(bytevector-like-length` *`bytevector`*`) =>` *`integer`*

`(bytevector-like-ref` *`bytevector offset`*`) =>` *`byte`*

`(bytevector-like-set!` *`bytevector offset byte`*`) =>` *`unspecified`*

`(bytevector-like-equal?` *`bytevector1 bytevector2`*`) =>` *`boolean`*

`(bytevector-like-copy` *`bytevector`*`) =>` *`bytevector`*

A bytevector-like structure is a low-level representation for indexed arrays of uninterpreted bytes. Bytevector-like structures are used to represent types such as bignums and flonums.

There is no way to construct a "generic" bytevector-like structure; use the constructors for specific bytevector-like types.

The bytevector-like operations operate on all bytevector-like structures. All are integrable, except `bytevector-like-equal?` and `bytevector-like-copy`. All are deprecated because they violate abstraction barriers and make your code representation-dependent; they are useful mainly to Larceny developers, who might otherwise be tempted to write some low-level operations in C or assembly language.

11.3. Vectors

Procedure vector-copy

(vector-copy *vector*) => *vector*

Returns a shallow copy of its argument.

Operations on vector-like structures

(vector-like? *object*) => *boolean*

(vector-like-length *vector-like*) => *fixnum*

(vector-like-ref *vector-like k*) => *object*

(vector-like-set! *vector-like k object*) => *unspecified*

A vector-like structure is a low-level representation for indexed arrays of Scheme objects. Vector-like structures are used to represent types such as vectors, records, symbols, and ports.

There is no way to construct a "generic" vector-like structure; use the constructors for specific data types.

The vector-like operations operate on all vector-like structures. All are integrable. All are deprecated because they violate abstraction barriers and make your code representation-dependent; they are useful mainly to Larceny developers, who might otherwise be tempted to write some low-level operations in C or assembly language.

11.4. Procedures

Operations on procedures

(make-procedure *length*) => *procedure*

(procedure-length *procedure*) => *fixnum*

(procedure-ref *procedure offset*) => *object*

(procedure-set! *procedure offset object*) => *unspecified*

These procedures operate on the representations of procedures and allow user programs to construct, inspect, and alter procedures.

Procedure procedure-copy

```
(procedure-copy procedure) => procedure
```

Returns a shallow copy of the procedure.

Warning

The procedures above are deprecated because they violate abstraction barriers and make your code representation-dependent; they are useful mainly to Larceny developers, who might otherwise be tempted to write some low-level operations in C or assembly language.

The rest of this section describes some procedures that reach through abstraction barriers in a more controlled way to extract heuristic information from procedures for debugging purposes.

Note

The following text is copied from a straw proposal authored by Will Clinger and sent to rrr-authors on 09 May 1996. The text has been edited lightly. See the end for notes about the Larceny implementation.

The procedures that extract heuristic information from procedures are permitted to return any result whatsoever. If the type of a result is not among those listed below, then the result represents an implementation-dependent extension to this interface, which may safely be interpreted as though no information were available from the procedure. Otherwise the result is to be interpreted as described below.

Procedure procedure-arity

```
(procedure-arity proc)
```

Returns information about the arity of *proc*. If the result is `#f`, then no information is available. If the result is an exact non-negative integer *k*, then *proc* requires exactly *k* arguments. If the result is an inexact non-negative integer *n*, then *proc* requires *n* or more arguments. If the result is a pair, then it is a list of non-negative integers, each of which indicates a number of arguments that will be accepted by *proc*; the list is not necessarily exhaustive.

Procedure procedure-documentation-string

```
(procedure-documentation-string proc)
```

Returns general information about *proc*. If the result is `#f`, then no information is available. If the result is a string, then it is to be interpreted as a "documentation string" (see Common Lisp).

Procedure procedure-name

```
(procedure-name proc)
```

Returns information about the name of *proc*. If the result is `#f`, then no information is available. If the result is a symbol or string, then it represents a name. If the result is a pair, then it is a list of symbols and/or strings representing a path of names; the first element represents an outer name and the last element represents an inner name.

Procedure procedure-source-file

`(procedure-source-file proc)`

Returns information about the name of a file that contains the source code for *proc*. If the result is `#f`, then no information is available. If the result is a string, then the string is the name of a file.

Procedure procedure-source-position

`(procedure-source-position proc)`

Returns information about the position of the source code for *proc* whithin the source file specified by procedure-source-file. If the result is `#f`, then no information is available. If the result is an exact integer *k*, then *k* characters precede the opening parenthesis of the source code for *proc* within that source file.

Procedure procedure-expression

`(procedure-expression proc)`

Returns information about the source code for *proc*. If the result is `#f`, then no information is available. If the result is a pair, then it is a lambda expression in the traditional representation of a list.

Procedure procedure-environment

`(procedure-environment proc)`

Returns information about the environment of *proc*. If the result is `#f`, then no information is available. In any case the result may be passed to any of the environment inquiry functions.

Notes on the Larceny implementation

Twobit does not yet produce data for all of these functions, so some of them always return `#f`.

11.5. Pairs and Lists

The `(rnrs lists)` library now provides a set of procedures that may supersede some of the procedures described below. If one of Larceny's procedures duplicates the semantics of an R6RS procedure whose name is different, then Larceny's name is deprecated.

Procedure append!

`(append! list1 list2 … obj) => object`

`append!` destructively appends its arguments, which must be lists, and returns the resulting list. The last argument can be any object. The argument lists are appended by changing the cdr of the last pair of each argument except the last to point to the next argument.

Procedure every?

`(every? procedure list1 list2 …) => object`

`every?` applies *procedure* to each element tuple of *list_s in first-to-last order, and returns `#f` as soon as _procedure* returns `#f`. If *procedure* does not return `#f` for any element tuple of *list_s, then the value* *returned by _procedure* for the last element tuple of _list_s is returned.

Procedure last-pair

`(last-pair` *`list-structure`*`) =>` *`pair`*

`last-pair` returns the last pair of the *list structure*, which must be a sequence of pairs linked through the cdr fields.

Procedure list-copy

`(list-copy` *`list-copy`*`) =>` *`list`*

`list-copy` makes a shallow copy of the *list* and returns that copy.

Procedure remove

`(remove` *`key list`*`) =>` *`list`*
Procedure remq

`(remq` *`key list`*`) =>` *`list`*
Procedure remv

`(remv` *`key list`*`) =>` *`list`*
Procedure remp

`(remp` *`pred? list`*`) =>` *`list`*

Each of these procedures returns a new list which contains all the elements of *list* in the original order, except that those elements of the original list that were equal to *key* (or that satisfy *pred?*) are not in the new list. Remove uses `equal?` as the equivalence predicate; `remq` uses `eq?`, and `remv` uses `eqv?`.

Procedure remove!

`(remove!` *`key list`*`) =>` *`list`*
Procedure remq!

`(remq!` *`key list`*`) =>` *`list`*
Procedure remv!

`(remv!` *`key list`*`) =>` *`list`*
Procedure remp!

`(remp!` *`pred? list`*`) =>` *`list`*

These procedures are like `remove`, `remq`, `remv`, and `remp`, except they modify *list* instead of returning a fresh list.

Procedure reverse!

`(reverse!` *`list`*`) =>` *`list`*

`reverse!` destructively reverses its argument and returns the reversed list.

Procedure some?

`(some? `*`procedure list1 list2 …`*`) => `*`object`*

`some?` applies *procedure* to each element tuple of *list_s in first-to-last order, and returns the first non-false value returned by _procedure.* If *procedure* does not return a true value for any element tuple of _list_s, then some? returns `#f`.

11.6. Sorting

The `(rnrs sorting)` library now provides a small set of procedures that supersede most of the procedures described below. All of the procedures described below are therefore deprecated.

Procedures sort and sort!

`(sort `*`list less?`*`) => `*`list`*

`(sort `*`vector less?`*`) => `*`vector`*

`(sort! `*`list less?`*`) => `*`list`*

`(sort! `*`vector less?`*`) => `*`vector`*

These procedures sort their argument (a list or a vector) according to the predicate *less?*, which must implement a total order on the elements in the data structures that are sorted.

`sort` returns a fresh data structure containing the sorted data; `sort!` sorts the data structure in-place.

11.7. Records

Note

Larceny's records have been extended to implement all SRFI 99 and R6RS [http://www.r6rs.org/] procedures from

```
(srfi :99 records procedural)
(srfi :99 records inspection)
(rnrs records procedural)
(rnrs records inspection)
```

We recommend that Larceny programmers use the SRFI 99 APIs instead of the R6RS APIs. This should entail no loss of portability, since the standard reference implementation of SRFI 99 records should run efficiently in any implementation of the R7RS/R6RS that permits new libraries to defined at all.

Larceny now has two kinds of records: old-style and R7RS/R6RS/SRFI99/ERR5RS. Old-style records cannot be created in R6RS-conforming mode, so our extension of R6RS procedures to accept old-style records does not affect R6RS conformance.

Note

The following specification describes Larceny's old-style record API, which is now deprecated. It is based on a proposal posted by Pavel Curtis to rrrs-authors on 10 Sep 1989, and later re-posted by Norman Adams to comp.lang.scheme on 5 Feb 1992. The authorship and copyright status of the original text are unknown to me.

This document differs from the original proposal in that its record types are extensible, and that it specifies the type of record-type descriptors.

11.7.1. Specification

Procedure make-record-type

```
(make-record-type type-name field-names)
```

Returns a "record-type descriptor", a value representing a new data type, disjoint from all others. The *type-name* argument must be a string, but is only used for debugging purposes (such as the printed representation of a record of the new type). The *field-names* argument is a list of symbols naming the "fields" of a record of the new type. It is an error if the list contains any duplicates.

If the *parent-rtd* argument is provided, then the new type will be a subtype of the type represented by *parent-rtd*, and the field names of the new type will include all the field names of the parent type. It is an error if the complete list of field names contains any duplicates.

Record-type descriptors are themselves records. In particular, record-type descriptors have a field printer that is either `#f` or a procedure. If the value of the field is a procedure, then the procedure will be called to print records of the type represented by the record-type descriptor. The procedure must accept two arguments: the record object to be printed and an output port.

Procedure record-constructor

```
(record-constructor rtd)
```

Returns a procedure for constructing new members of the type represented by *rtd.* The returned procedure accepts exactly as many arguments as there are symbols in the given list, *field-names*; these are used, in order, as the initial values of those fields in a new record, which is returned by the constructor procedure. The values of any fields not named in that list are unspecified. The field-names argument defaults to the list of field-names in the call to make-record-type that created the type represented by *rtd*; if the *field-names* argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

Procedure record-predicate

```
(record-predicate rtd)
```

Returns a procedure for testing membership in the type represented by *rtd.* The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type or one of its subtypes; it returns a false value otherwise.

Procedure record-accessor

```
(record-accessor rtd field-name)
```

Returns a procedure for reading the value of a particular field of a member of the type represented by *rtd.* The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol *field-name* in that record. The symbol field-name must be a member of the list of field-names in the call to make-record-type that created the type represented by *rtd,* or a member of the field-names of the parent type of the type represented by *rtd.*

Procedure record-updater

```
(record-updater rtd field-name)
```

Returns a procedure for writing the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol *field-name* in that record to contain the given value. The returned value of the updater procedure is unspecified. The symbol *field-name* must be a member of the list of field-names in the call to make-record-type that created the type represented by *rtd*, or a member of the field-names of the parent type of the type represented by *rtd*.

```
(record? obj)
```

Returns a true value if *obj* is a record of any type and a false value otherwise. Note that `record?` may be true of any Scheme value; of course, if it returns true for some particular value, then `record-type-descriptor` is applicable to that value and returns an appropriate descriptor.

Procedure record-type-descriptor

```
(record-type-descriptor record)
```

Returns a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to record-predicate, the resulting predicate would return a true value when passed the given record. Note that it is not necessarily the case that the returned descriptor is the one that was passed to record-constructor in the call that created the constructor procedure that created the given record.

Procedure record-type-name

```
(record-type-name rtd)
```

Returns the type-name associated with the type represented by *rtd*. The returned value is eqv? to the type-name argument given in the call to make-record-type that created the type represented by rtd.

Procedure record-type-field-names

```
(record-type-field-names rtd)
```

Returns a list of the symbols naming the fields in members of the type represented by *rtd*.

Procedure record-type-parent

```
(record-type-parent rtd)
```

Returns a record-type descriptor for the parent type of the type represented by *rtd*, if that type has a parent type, or a false value otherwise. The type represented by *rtd* has a parent type if the call to make-record-type that created *rtd* provided the *parent-rtd* argument.

Procedure record-type-extends?

```
(record-type-extends? rtd1 rtd2)
```

Returns a true value if the type represented by *rtd1* is a subtype of the type represented by *rtd2* and a false value otherwise. A type *s* is a subtype of a type *t* if *s*=*t* or if the parent type of *s*, if it exists, is a subtype of *t.*

11.7.2. Implementation

The R6RS spouts some tendentious nonsense about procedural records being slower than syntactic records, but this is not true of Larceny's records, and is unlikely to be true of other implementations either. Larceny's procedural records are fairly efficient already, and will become even more efficient in future versions as interlibrary optimizations are added.

11.8. Input, Output, and Files

The `(scheme base)`, `(scheme file)`, `(rnrs io ports)`, and `(rnrs files)` libraries now provide a set of procedures that may supersede some of the procedures described below. If one of Larceny's procedures duplicates the semantics of an R7RS or R6RS procedure whose name is different, then Larceny's name is deprecated.

Procedure close-open-files

`(close-open-files )` => *unspecified*

Closes all open files.

Procedure console-input-port

`(console-input-port )` => *input-port*

Returns a character input port such that no read from the port has signalled an error or returned the end-of-file object.

Rationale: console-input-port and console-output-port are artifacts of Unix interactive I/O conventions, where an interactive end-of-file does not mean "quit" but rather "done here". Under these conventions the console port should be reset following an end-of-file. Resetting conflicts with the semantics of ports in Scheme, so console-input-port and console-output-port return a new port if the current port is already at end-of-file.

Since it is convenient to handle errors in the same manner as end-of-file, these procedures also return a new port if an error has been signalled during an I/O operation on the port.

Console-input-port and console-output-port simply call the port generators installed in the parameters console-input-port-factory and console-output-port-factory, which allow user programs to install their own console port generators.

Procedure console-output-port

`(console-output-port )` => *output-port*

Returns a character output port such that no write to the port has signalled an error.

See console-input-port for a full explanation.

Parameter console-input-port-factory

The value of this parameter is a procedure that returns a character input port such that no read from the port has signalled an error or returned the end-of-file object.

See console-input-port for a full explanation.

Parameter console-output-port-factory

The value of this parameter is a procedure that returns a character output port such that no write the port has signalled an error.

See console-input-port for a full explanation.

Parameter current-input-port

The value of this parameter is a character input port.

Parameter current-output-port

The value of this parameter is a character output port.

Procedure delete-file

```
(delete-file filename) => unspecified
```

Deletes the named file. No error is signalled if the file does not exist.

Procedure eof-object

```
(eof-object ) => end-of-file object
```

Eof-object returns an end-of-file object.

Procedure file-exists?

```
(file-exists? filename) => boolean
```

File-exists? returns #t if the named file exists at the time the procedure is called.

Procedure file-modification-time

```
(file-modification-time filename) => vector or #f
```

File-modification-time returns the time of last modification of the file as a vector, or #f if the file does not exist. The vector has six elements: year, month, day, hour, minute, second, all of which are exact nonnegative integers. The time returned is relative to the local timezone.

```
(file-modification-time "larceny") => #(1997 2 6 12 51 13)
```

```
(file-modification-time "geekdom") => #f
```

Procedure flush-output-port

```
(flush-output-port ) => unspecified
```

```
(flush-output-port port) => unspecified
```

Write any buffered data in the port to the underlying output medium.

Procedure get-output-string

`(get-output-string` *`string-output-port`*`) =>` *`string`*

Retrieve the output string from the given string output port.

Procedure open-input-string

`(open-input-string` *`string`*`) =>` *`input-port`*

Creates an input port that reads from *string*. The string may be shared with the caller. A string input port does not need to be closed, although closing it will prevent further reads from it.

Procedure open-output-string

`(open-output-string ) =>` *`output-port`*

Creates an output port where any output is written to a string. The accumulated string can be retrieved with get-output-string at any time.

Procedure port?

`(port?` *`object`*`) =>` *`boolean`*

Tests whether its argument is a port.

Procedure port-name

`(port-name` *`port`*`) =>` *`string`*

Returns the name associated with the port; for file ports, this is the file name.

Procedure port-position

`(port-position` *`port`*`) =>` *`fixnum`*

Returns the number of characters that have been read from or written to the port.

Procedure rename-file

`(rename-file` *`from to`*`) =>` *`unspecified`*

Renames the file *from* and gives it the name *to*. No error is signalled if *from* does not exist or *to* exists.

Procedure reset-output-string

`(reset-output-string` *`port`*`) =>` *`unspecified`*

Given a *port* created with *open-output-string*, deletes from the port all the characters that have been output so far.

Procedure with-input-from-port

`(with-input-from-port` *`input-port thunk`*`) =>` *`object`*

Calls *thunk* with current input bound to *input-port* in the dynamic extent of *thunk*. Returns whatever value was returned from *thunk*.

Procedure with-output-to-port

`(with-output-to-port` *`output-port thunk`*`) =>` *`object`*

Calls *thunk* with current output bound to *output-port* in the dynamic extent of *thunk*. Returns whatever value was returned from *thunk*.

11.9. Operating System Interface

Procedure command-line-arguments

`(command-line-arguments ) =>` *`vector`*

Returns a vector of strings: the arguments supplied to the program by the user or the operating system.

Procedure dump-heap

`(dump-heap` *`filename procedure`*`) =>` *`unspecified`*

Dump a heap image to the named file that will start up with the supplied procedure. Before *procedure* is called, command line arguments will be parsed and any init procedures registered with `add-init-procedure!` will be called.

Note: Currently, heap dumping is only available with the stop-and-copy collector (`-stopcopy` command line option), although the heap image can be used with all the other collectors.

Procedure dump-interactive-heap

`(dump-interactive-heap` *`filename`*`) =>` *`unspecified`*

Dump a heap image to the named file that will start up with the standard read-eval-print loop. Before the read-eval-print loop is called, command line arguments will be parsed and any init procedures registered with `add-init-procedure!` will be called.

Note: Currently, heap dumping is only available with the stop-and-copy collector (`-stopcopy` command line option), although the heap image can be used with all the other collectors.

Procedure getenv

`(getenv` *`key`*`) =>` *`string or #f`*

Returns the operating system environment mapping for the string *key*, or `#f` if there is no mapping for *key*.

Note

This is now a synonym for the `get-environment-variable` exported by the `(scheme process-context)` library.

Procedure setenv

`(setenv` *`key val`*`) =>` *`unspecified`*

Sets the operating system environment mapping for the string *key* to *val*.

Procedure system

```
(system command) => status
```

Send the *command* to the operating system's command processor and return the command's exit status, if any. On Unix, *command* is a string and *status* is an exact integer.

11.10. Fixnum primitives

Fixnums are small exact integers that are likely to be represented without heap allocation. Larceny never represents a number that can be represented as a fixnum any other way, so programs that can use fixnums will do so automatically. However, operations that work only on fixnums can sometimes be substantially faster than generic operations, and the following primitives are provided for use in those programs that need especially good performance.

The `(rnrs arithmetic fixnums)` library now provides a large set of procedures, some of them similar to the procedures described below. If one of Larceny's procedures duplicates the semantics of an R6RS procedure whose name is different, then Larceny's name is deprecated within R7RS/R6RS code.

All arguments to the following procedures must be fixnums.

Procedure fixnum?

```
(fixnum? obj) => boolean
```

Returns `#t` if its argument is a fixnum, and `#f` otherwise.

Procedure fx+

```
(fx+ fix1 fix2) => fixnum
```

Returns the fixnum sum of its arguments. If the result is not representable as a fixnum, then an error is signalled (unless error checking has been disabled).

Procedure fx-

Returns the fixnum difference of its arguments. If the result is not representable as a fixnum, then an error is signalled.

Procedure fx—

```
(fx— fix1) => fixnum
```

Returns the fixnum negative of its argument. If the result is not representable as a fixnum, then an error is signalled.

*Procedure fx**

```
(fx* fix1 fix2) => fixnum
```

Returns the fixnum product of its arguments. If the result is not representable as a fixnum, then an error is signalled.

Procedure fx=

`(fx= `*`fix1 fix2`*`) => `*`boolean`*

Returns `#t` if its arguments are equal, and `#f` otherwise.

Procedure fx<

`(fx< `*`fix1 fix2`*`) => `*`boolean`*

Returns `#t` if *fix1* is less than *fix2*, and `#f` otherwise.

Procedure fx<=

`(fx<= `*`fix1 fix2`*`) => `*`boolean`*

Returns `#t` if *fix1* is less than or equal to *fix2*, and `#f` otherwise.

Procedure fx>

`(fx> `*`fix1 fix2`*`) => `*`boolean`*

Returns `#t` if *fix1* is greater than *fix2*, and `#f` otherwise.

Procedure fx>=

`(fx>= `*`fix1 fix2`*`) => `*`boolean`*

Returns `#t` if *fix1* is greater than or equal to *fix2*, and `#f` otherwise.

Procedure fxnegative?

`(fxnegative? `*`fix`*`) => `*`boolean`*

Returns `#t` if its argument is less than zero, and `#f` otherwise.

Procedure fxpositive?

`(fxpositive? `*`fix`*`) => `*`boolean`*

Returns `#t` if its argument is greater than zero, and `#f` otherwise.

Procedure fxzero?

`(fxzero? `*`fix`*`) => `*`boolean`*

Returns `#t` if its argument is zero, and `#f` otherwise.

Procedure fxlogand

`(fxlogand `*`fix1 fix2`*`) => `*`fixnum`*

Returns the bitwise *and* of its arguments.

Procedure fxlogior

`(fxlogior `*`fix1 fix2`*`) => `*`fixnum`*

Returns the bitwise *inclusive or* of its arguments.

Procedure fxlognot

`(fxlognot fix) => fixnum`

Returns the bitwise *not* of its argument.

Procedure fxlogxor

`(fxlogxor fix1 fix2) => fixnum`

Returns the bitwise *exclusive or* of its arguments.

Procedure fxlsh

`(fxlsh fix1 fix2) => fixnum`

Returns *fix1* shifted left *fix2* places, shifting in zero bits at the low end. If the shift count exceeds the number of bits in the machine's word size, then the results are machine-dependent.

Procedure most-positive-fixnum

`(most-positive-fixnum ) => fixnum`

Returns the largest representable positive fixnum.

Procedure most-negative-fixnum

`(most-negative-fixnum ) => fixnum`

Returns the smallest representable negative fixnum.

Procedure fxrsha

`(fxrsha fix1 fix2) => fixnum`

Returns *fix1* shifted right *fix2* places, shifting in a copy of the sign bit at the left end. If the shift count exceeds the number of bits in the machine's word size, then the results are machine-dependent.

Procedure fxrshl

`(fxrshl fix1 fix2) => fixnum`

Returns *fix1* shifted right *fix2* places, shifting in zero bits at the high end. If the shift count exceeds the number of bits in the machine's word size, then the results are machine-dependent.

11.11. Numbers

Larceny has six representations for numbers: *fixnums* are small, exact integers; *bignums* are unlimited-precision exact integers; *ratnums* are exact rationals; *flonums* are inexact rationals; *rectnums* are exact complexes; and *compnums* are inexact complexes.

Number-representation predicates

`(fixnum? obj) => boolean`

`(bignum? obj) => boolean`

`(ratnum? obj) => boolean`

`(flonum? obj) => boolean`

`(rectnum? obj) => boolean`

`(compnum? obj) => boolean`

These predicates test whether an object is a number of a particular representation and return `#t` if so, `#f` if not.

Procedure random

`(random limit) => exact integer`

Returns a pseudorandom nonnegative exact integer in the range 0 through *limit*-1.

11.12. Hashtables and hash functions

Hashtables represent finite mappings from keys to values. If the hash function is a good one, then the value associated with a key may be looked up in constant time (on the average).

Note

The R6RS hashtables library are a big improvement over Larceny's traditional hash tables, and should be used instead of the API described below.

Note

To resolve a clash of names and semantics with the R6RS `make-hashtable` procedure, Larceny's traditional `make-hashtable` procedure has been renamed to `make-oldstyle-hashtable`.

11.12.1. Hash tables

Procedure make-oldstyle-hashtable

`(make-oldstyle-hashtable hash-function bucket-searcher size) => hashtable`

Returns a newly allocated mutable hash table using *hash-function* as the hash function and *bucket-searcher*, e.g. `assq`, `assv`, `assoc`, to search a bucket with *size* buckets at first, expanding the number of buckets as needed. The *hash-function* must accept a key and return a non-negative exact integer.

`(make-oldstyle-hashtable hash-function bucket-searcher) => hashtable`

Equivalent to `(make-oldstyle-hashtable hash-function bucket-searcher n)` for some value of *n* chosen by the implementation.

`(make-oldstyle-hashtable` *`hash-function`*`) =>` *`hashtable`*

Equivalent to `(make-oldstyle-hashtable` *`hash-function`* `assv)`.

`(make-oldstyle-hashtable ) =>` *`hashtable`*

Equivalent to `(make-oldstyle-hashtable object-hash assv)`.

Procedure hashtable-contains?

`(hashtable-contains?` *`hashtable key`*`) =>` *`bool`*

Returns true iff the *hashtable* contains an entry for *key*.

Procedure hashtable-fetch

`(hashtable-fetch` *`hashtable key flag`*`) =>` *`object`*

Returns the value associated with *key* in the *hashtable* if the *hashtable* contains *key*; otherwise returns *flag*.

Procedure hashtable-get

`(hashtable-get` *`hashtable key`*`) =>` *`object`*

Equivalent to `(hashtable-fetch #f)`.

Procedure hashtable-put!

`(hashtable-put!` *`hashtable key value`*`) =>` *`unspecified`*

Changes the *hashtable* to associate *key* with *value*, replacing any existing association for *key*.

Procedure hashtable-remove!

`(hashtable-remove!` *`hashtable key`*`) =>` *`unspecified`*

Removes any association for *key* within the *hashtable*.

Procedure hashtable-clear!

`(hashtable-clear!` *`hashtable`*`) =>` *`unspecified`*

Removes all associations from the *hashtable*.

Procedure hashtable-size

`(hashtable-size` *`hashtable`*`) =>` *`integer`*

Returns the number of keys contained within the *hashtable*.

Procedure hashtable-for-each

`(hashtable-for-each` *`procedure hashtable`*`) =>` *`unspecified`*

The *procedure* must accept two arguments, a key and the value associated with that key. Calls the *procedure* once for each key-value association in *hashtable*. The order of these calls is indeterminate.

Procedure hashtable-map

`(hashtable-map` *`procedure hashtable`*`)`

The *procedure* must accept two arguments, a key and the value associated with that key. Calls the *procedure* once for each key-value association in *hashtable*, and returns a list of the results. The order of the calls is indeterminate.

Procedure hashtable-copy

`(hashtable-copy` *`hashtable`*`) =>` *`hashtable`*

Returns a copy of the *hashtable*.

11.12.2. Hash functions

The *hash values* returned by these functions are nonnegative exact integer suitable as hash values for the hashtable functions.

Procedure equal-hash

`(equal-hash` *`object`*`) =>` *`integer`*

Returns a hash value for *object* based on its contents.

Procedure object-hash

`(object-hash` *`object`*`) =>` *`integer`*

Returns a hash value for *object* based on its identity.

Warning

This hash function performs extremely poorly on pairs, vectors, strings, and bytevectors, which are the objects with which it is mostly likely to be used. For efficient hashing on object identity, create the hashtable with `make-eq-hashtable` or `make-eqv-hashtable` of the `(rnrs hashtables)` library.

Procedure string-hash

`(string-hash` *`string`*`) =>` *`fixnum`*

Returns a hash value for *string* based on its content.

Procedure symbol-hash

`(symbol-hash` *`symbol`*`) =>` *`fixnum`*

Returns a hash value for *symbol* based on its print name. The `symbol-hash` is very fast, because the hash code is cached in the symbol data structure.

11.13. Parameters

Parameters are procedures that serve as containers for values.

When called with no arguments, a parameter returns its current value. The value of a parameter can be changed temporarily using the *parameterize* syntax described below.

The effect of passing arguments to a parameter is implementation-dependent. In Larceny, passing one argument to a parameter changes the current value of the parameter to the result of applying a *converter* procedure to that argument, as described by SRFI 39.

Procedure make-parameter

```
(make-parameter init) => procedure
```

```
(make-parameter init converter) => procedure
```

```
(make-parameter name init predicate) => procedure
```

Creates a parameter.

When *make-parameter* is called with one argument *init*, the parameter's initial value is *init*, and the parameter's *converter* will be the identity function.

When *make-parameter* is called with two arguments, *converter* must be a procedure that accepts one argument, and the parameter's initial value is the result of calling *converter* on *init*.

Larceny extends SRFI 39 and the R7RS specification of *make-parameter* by allowing it to be called with three arguments. The first argument, *name*, must be a symbol or string giving the print name of the parameter. The second argument, *init*, will be the initial value of the parameter. The third argument is a *predicate* from which Larceny constructs a *converter* procedure that acts like the identity function on arguments that satisfy the *predicate* but raises an exception on arguments that don't.

```
(make-parameter name init) => procedure
```

Larceny's parameter objects predate SRFI 39. For backward compatibility, Larceny's *make-parameter* will accept two arguments even if the second is not a procedure, provided the first argument is a symbol or string. In that special case, the two arguments will be treated as the *name* and *init* arguments to Larceny's three-argument version, with the *predicate* defaulting to the identity function. *This extension is strongly deprecated.*

Syntax parameterize

```
(parameterize ((parameter0 value0) …) expr0 expr1 …)
```

Parameterize temporarily overrides the values of a set of parameters while the expressions in the body of the *parameterize* expression are evaluated. (It is like *fluid-let* for parameters instead of variables.)

11.13.1. Larceny parameters

The following is a partial list of Larceny's parameters. The first three are described by the R7RS standard.

Most of the others are intended for use by developers of Larceny; some are described in Wiki pages at Larceny's GitHub site, while others are described only by source code.

Parameter `current-input-port` [io.html#proc:current-input-port]

Parameter `current-output-port` [io.html#proc:current-output-port]

Parameter `current-error-port` [io.html#proc:current-error-port]

Parameter `console-input-port-factory` [io.html#proc:console-input-port-factory]

Parameter `console-output-port-factory` [io.html#proc:console-output-port-factory]

Parameter `herald` [repl.html#proc:herald]

Parameter `interaction-environment` [environ.html#proc:interaction-environment]

Parameter `evaluator` [control.html#proc:evaluator]

Parameter `load-evaluator` [control.html#proc:load-evaluator]

Parameter `repl-evaluator` [repl.html#proc:repl-evaluator]

Parameter `repl-level` [repl.html#proc:repl-level]

Parameter `repl-printer` [repl.html#proc:repl-printer]

Parameter `break-handler` [debugging.html#proc:break-handler]

Parameter `error-handler` [control.html#proc:error-handler]

Parameter `quit-handler` [control.html#proc:quit-handler]

Parameter `reset-handler` [control.html#proc:reset-handler]

Parameter `keyboard-interrupt-handler` [control.html#proc:keyboard-interrupt-handler]

Parameter `timer-interrupt-handler` [control.html#proc:timer-interrupt-handler]

Parameter `standard-timeslice` [control.html#proc:standard-timeslice]

Parameter `structure-comparator` [structures.html#proc:structure-comparator]

Parameter `structure-printer` [structures.html#proc:structure-printer]

11.14. Property Lists

The *property list* of a symbol is an association list that is attached to that symbol. The association list maps *properties*, which are themselves symbols, to arbitrary values.

Procedure putprop

```
(putprop symbol property obj) => unspecified
```

If an association exists for *property* on the property list of *symbol*, then its value is replaced by the new value *obj*. Otherwise, a new association is added to the property list of *symbol* that associates *property* with *obj*.

Procedure getprop

`(getprop `*`symbol property`*`) => `*`obj`*

If an association exists for *property* on the property list of *symbol*, then its value is returned. Otherwise, `#f` is returned.

Procedure remprop

`(remprop `*`symbol property`*`) => `*`unspecified`*

If an association exists for *property* on the property list of *symbol*, then that association is removed. Otherwise, this is a no-op.

11.15. Symbols

Procedure gensym

`(gensym `*`string`*`) => `*`symbol`*

Gensym returns a new uninterned symbol, the name of which contains the given *string*.

Procedure oblist

`(oblist ) => `*`list`*

Oblist returns the list of interned symbols.

Procedure oblist-set!

`(oblist-set! `*`list`*`) => `*`unspecified`*

`(oblist-set! `*`list table-size`*`) => `*`unspecified`*

`oblist-set!` sets the list of interned symbols to those in the given *list* by clearing the symbol hash table and storing the symbols in *list* in the hash table. If the optional *table-size* is given, it is taken to be the desired size of the new symbol table.

See also: symbol-hash.

11.16. System Control and Performance Measurement

Procedure collect

`(collect ) => `*`unspecified`*

`(collect `*`generation`*`) => `*`unspecified`* `(collect` *`generation method`*`) =>` *`unspecified`*

Collect initiates a garbage collection. If the system has multiple generations, then the optional arguments are interpreted as follows. The *generation* is the generation to collect, where 0 is the youngest generation. The *method* determines how the collection is performed. If *method* is the symbol collect, then a full collection is performed in that generation, whatever that means — in a normal multi-generational copying collector, it means that all live objects in the generation's current semispace and all live objects from all younger generations are copied into the generation's other semispace. If *method* is the symbol promote, then live objects are promoted from younger generations into the target generation — in our example collector, that means that the objects are copied into the target generation's current semispace.

The default value for *generation* is 0, and the default value for *method* is collect.

Note that the collector's internal policy settings may cause it to perform a more major type of collection than the one requested; for example, an attempt to collect generation 2 could cause the collector to promote all live data into generation 3.

Procedure gc-counter

`(gc-counter ) =>` *`fixnum`*

gc-counter returns the number of garbage collections performed since startup. On a 32-bit system, the counter wraps around every 1,073,741,824 collections.

gc-counter is a primitive and compiles to a single load instruction on the SPARC.

Procedure major-gc-counter

`(major-gc-counter ) =>` *`fixnum`*

major-gc-counter returns the number of major garbage collections performed since startup, where a major collection is defined as a collection that may change the address of objects that have already survived a previous collection. On a 32-bit system, the counter wraps around every 1,073,741,824 collections.

major-gc-counter is a primitive and compiles to a single load instruction on the SPARC. Its primary use to implement efficient hashtables that hash on object identity (make-eq-hashtable and make-eqv-hashtable).

Procedure gcctl

`(gcctl` *`heap-number operation operand`*`) =>` *`unspecified`*

[GCCTL is largely obsolete in the new garbage collector but may be resurrected in the future. It can still be used to control the non-predictive collector.]

gcctl controls garbage collection policy on a heap-wise basis. The *heap-number* is the heap to operate on, like for the command line switches: heap 1 is the youngest. If the given heap number does not correspond to a heap, gcctl fails silently.

The *operation* is a symbol that selects the operation to perform, and the *operand* is the operand to that operation, always a number. For the non-predictive garbage collector, the following operator/operand pairs are meaningful:

- j-fixed, *n*: after a collection, the collector parameter *j* should be set to the value *n*, if possible. (Non-predictive heaps only.)
- j-percent, *n*: after a collection, the collector parameter *j* should be set to be *n* percent of the number of free steps. (Non-predictive heaps only.)
- incr-fixed, *n*: when growing the heap, the growing should be done in increments of *n*. In the non-predictive heap, *n* is the number of steps. In other heaps, *n* denotes kilobytes.
- incr-percent, *n*: when growing the heap, the growing should be done in increments of *n* percent.

Example: if the non-predictive heap is heap number 2, then the expressions

```
(gcctl 2 'j-fixed 0)
(gcctl 2 'incr-fixed 1)
```

makes the non-predictive collector simulate a normal stop-and-copy collector (because *j* is always set to 0), and grows the heap only one step at a time as necessary. This may be useful for certain kinds of experiments.

Example: ditto, the expressions

```
(gcctl 2 'j-percent 50)
(gcctl 2 'incr-percent 20)
```

selects the default policy settings.

Note: The gcctl facility is experimental. A more developed facility will allow controlling heap contraction policy, as well as setting all the watermarks. Certainly one can envision other uses, too. Finally, it needs to be possible to get current values.

Note: Currently the non-predictive heap (np-sc-heap.c) and the standard stop-and-copy "old" heap (old-heap.c) are supported, but not the standard "young" heap (young-heap.c), nor the stop-and-copy collector (sc-heap.c).

Procedure sro

(sro *pointer-tag type-tag limit*) => *vector*

SRO ("standing room only") is a system primitive that traverses the entire heap and returns a vector that contains all live objects in the heap that satisfy the constraints imposed by its parameters:

- If *pointer-tag* is -1, then object type is unconstrained; otherwise, the object type is constrained to have a pointer tag that matches *pointer-tag*. You can read all about pointer tags here, but the short story is that 1=pair, 3=vector-like, 5=bytevector-like, and 7=procedure-like.
- If *type-tag* is -1, then object type is unconstrained by type-tag; otherwise, only objects with a matching type-tag are selected (after selection by pointer tag). Pairs don't have type-tags, but other objects do. You can read all about type-tags here.
- *Limit* constrains the selected objects by the number of references. If *limit* is -1, then no constraints are imposed; otherwise, only objects (selected by pointer-tag and type-tag) with no more than *limit*

references to them are selected.

For example, (sro -1 -1 -1) returns a vector that contains all live objects (not including the vector), and (sro 5 2 3) returns a vector containing all live flonums (bytevector-like, with typetag 2) that are referred to in no more than 3 places.

Procedure stats-dump-on

```
(stats-dump-on filename) => unspecified
```

Stats-dump-on turns on garbage collection statistics dumping. After each collection, a complete RTS statistics dump is appended to the file named by *filename*.

The file format and contents are documented in a banner written at the top of the output file. In addition, accessor procedures for the output structure are defined in the program Util/process-stats.sch.

Stats-dump-on does not perform an initial dump when the file is first opened; only at the first collection is the first set of statistics dumped. The user might therefore want to initiate a minor collection just after turning on dumping in order to have a baseline set of data.

Procedure stats-dump-off

```
(stats-dump-off ) => unspecified
```

Stats-dump-off turns off garbage collection statistics dumping (which was turned on with stats-dump-on). It does not dump a final set of statistics before closing the file; therefore, the user may wish to initiate a minor collection before calling this procedure.

Procedure system-features

```
(system-features ) => alist
```

System-features returns an association lists of system features. Most entries are self-explanatory. The following are a more subtle:

- The value of architecture-name is Larceny's notion of the architecture for which it was compiled, not the architecture the program is currently running on. For example, the value of this feature is "Standard-C" if you're running Petit Larceny.
- The value of heap-area-info is a vector of vectors, one subvector for each heap area in the running system. The subvector has four entries: the generation number, the area type, the current size, and additional information.

Procedure display-memstats

```
(display-memstats vector) => unspecified

(display-memstats vector minimal) => unspecified

(display-memstats vector minimal full) => unspecified
```

Display-memstats takes as its argument a vector as returned by memstats and displays the contents of the vector in human-readable form on the current output port. By default, not all of the values in the vector are displayed.

If the symbol minimal is passed as the second argument, then only a small number of statistics generally relevant to running benchmarks are displayed.

If the symbol full is passed as the second argument, then all statistics are displayed.

Procedure memstats

`(memstats )` => *vector*

Memstats returns a freshly allocated vector containing run-time-system resource usage statistics. Many of these will make no sense whatsoever to you unless you also study the RTS sources. A listing of the contents of the vector is available here.

Procedure run-with-stats

`(run-with-stats` *thunk*`)` => *obj*

Run-with-stats evaluates *thunk*, then prints a short summary of run-time statistics, as with

```
(display-memstats ... 'minimal),
```

and then returns the result of evaluating *thunk*.

Procedure run-benchmark

`(run-benchmark` *name k thunk ok?*`)` => *obj*

Run-benchmark prints a short banner (including the identifying *name*) to identify the benchmark, then runs *thunk k* times, and finally tests the value returned from the last call to *thunk* by applying the predicate *ok?* to it. If the predicate returns true, then run-benchmark prints summary statistics, as with

```
([display-memstats][5] ... 'minimal).
```

If the predicate returns false, an error is signalled.

11.17. SRFI Support

SRFIs (Scheme Requests For Implementations) describe and implement additional Scheme libraries. The SRFI effort is open to anyone, and is described at http://srfi.schemers.org.

SRFIs are numbered. Importing SRFIs into an R7RS library or program is straightforward:

```
(import (srfi 19)
        (srfi 27))
```

The R6RS forbids numbers within library names, so R6RS libraries and programs must import SRFI libraries using the SRFI 97 naming convention in which a colon precedes the number:

```
(import (srfi :19)
        (srfi :27))
```

To test whether particular SRFIs are available, use the R7RS `cond-expand` feature:

```
(cond-expand
 ((and (library (srfi 19))
       (library (srfi 27)))
  (import (srfi 19))
  (import (srfi 27))))
```

`cond-expand` is not available to R6RS libraries or programs.

R5RS programs can use `cond-expand` as implemented by SRFI 0, "Feature-based conditional expansion construct". (SRFI 0 must be loaded into Larceny before it can be used; see below.) Larceny provides the following nonstandard key for use in SRFI 0:

```
    larceny
```

Larceny currently supports many SRFIs, though not as many as it should. Some SRFIs are built into Larceny's R5RS mode, but most must be loaded dynamically using Larceny's `require` procedure:

```
    > (require 'srfi-0)
```

The design documents for SRFI 0 and other SRFIs are available at http://srfi.schemers.org.

11.18. SLIB support

SLIB [http://www-swiss.ai.mit.edu/~jaffer/SLIB.html] is a large collection of useful libraries that have been written or collected by Aubrey Jaffer.

Larceny supports SLIB via SRFI 96 [http://srfi.schemers.org/srfi-96/], but SLIB itself is not shipped with Larceny; it must be downloaded separately and then installed. For the most up-to-date information on installing and using SLIB with Larceny, see `doc/HOWTO-SLIB`.

11.19. Foreign-Function Interface to C

Larceny provides a general foreign-function interface (FFI) substrate on which other FFIs can be built; see Larceny Note #7 [LarcenyNotes/note7-ffi.html]. The FFI described in this manual section is a simple example of a derived FFI. It is not yet fully evolved, but it is useful.

Warning

This section has undergone signficant revision, but not all of the material has been properly vetted. Some of the information in this section may be out of date.

Note

Some of the text below is adapted from the 2008 Scheme Workshop paper, “The Layers of Larceny's Foreign Function Interface,” by Felix S Klock II. That paper may provide additional insight for those searching for implementation details and motivations.

11.19.1. Introducing the FFI

There are a number of different potential ways to use the FFI. One client may want to develop code in C and load it into Larceny. Another client may want to load native libraries provided by the host operating system, enabling invocation of foreign code from Scheme expressions without developing any C code or even running a C compiler. Larceny's FFI can be used for both of these cases, but many of its facilities target a third client in between the two extremes: a client with a C compiler and the header files and object code for the foreign libraries, but who wishes to avoid writing glue code in C to interface with the libraries.

There are four main steps to interacting with foreign code:

1. identifying the space of values manipulated by the foreign code that will also be manipulated in Scheme,
2. describing how to marshal values between foreign and Scheme code,
3. loading library file(s) holding foreign object code, and
4. linking procedures from the loaded library.

Step 1 is conceptual, while steps 2 through 4 yield artifacts in Scheme source code.

11.19.2. The space of foreign values

At the machine code level, foreign values are uninterpreted sequences of bits. Often foreign object code is oriented around manipulating word-sized bit-sequences (*words*) or arrays and tuples of words.

Many libraries are written with a particular interpretation of such values. In C code, explicit types are often used hints to guide such interpretation; for example, a `0` of type `bool` is usually interpreted as *false*, while a `1` (or other non-zero value) of type `bool` is usually interpreted as *true*. Another example are C enumerations (or *enums*). An enum declaration defines a set of named integral constants. After the C declaration:

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

a `JAN` in C code now denotes `1`, `FEB` is `2`, and so on. Furthermore, tools like debuggers may render a variable `x` dynamically assigned the value `2` (and of static type `enum months`) as `FEB`. Thus the enum declaration intoduces a new interpretation for a finite set of integers.

This leads to questions for a client of an FFI; we explore some below.

- Should foreign words be passed over to the Scheme world as uninterpreted numbers (and thus be converted into Scheme integers, usually fixnums), or should they be marshaled into interpreted values, such as `#f` and `#t` for the `bool` type, or the Scheme symbols {`JAN`, `FEB`, `MAR`, `APR`, `MAY`, `JUN`, `JUL`, `AUG`, `SEP`, `OCT`, `NOV`, `DEC`} for the `enum months` type?
- Similarly, how should Scheme values be marshaled into foreign words?
- A foreign library might leave the mapping of names like `FEB` to words like `2` *unspecified* in the library interface. That is, while the C compiler will know `FEB` maps to `2` according to a particular version of the library's header file, the library designer may intend to change this mapping in the future, and clients writing C code should *only* use the names to refer to a `enum months` value, and *not* integer expressions.

  - How should this constraint be handled in the FFI; should the library client revise their code in reaction to such changes to the mapping?
  - Or should the system derive the mapping from the header files, in the same manner that the C compiler does?
- Foreign libraries often manipulate mutable entities, like arrays of words where modifications can be observed (often by design).
  - How should such values be marshaled?
  - Is it sound to copy such values to the Scheme heap? If so, is a shallow copy sufficient?
- Will the foreign code hold references to heap-allocated objects? Heap-allocated objects that *leak* out to foreign memory must be treated with care; garbage collection presents two main problems.
  - First, such objects must not move during a garbage collection; Larceny supports this via special-purpose allocation routines: `cons-nonrelocatable`, `make-nonrelocatable-bytevector`, and `make-nonrelocatable-vector`.
  - Second, the garbage collector must know to hold on to (i.e. trace) such values as long as they are needed by foreign code; otherwise the objects or their referents may be collected without the knowledge of the foreign code.

Answering these questions may require deep knowledge of the intended usage of the foreign library.

The Larceny FFI attempts to ease interfacing with foreign code in the presence of the above concerns, but the nature of the header files included with most foreign libraries means that the FFI cannot infer the answers unassisted.

Note

Foreign C code developed to work in concert with Larceny could hypothetically be written to cope with holding handles for objects managed by the the garbage collector, but there is currently no significant support for this use-case.

Note

One class of foreign values is not addressed by the Larceny FFI: structures passed by value (as opposed to by reference, ie pointers to structures). There is no way to describe the interface to a foreign procedure that accepts or produces a C `struct` (at least not properly nor portably).

This tends to not matter for many foreign libraries (since many C programmers eschew passing structures by value), but it can arise.

If the foreign library of interest has procedures that accept or produce a C `struct`, we currently recommend either avoiding such procedures, or writing adapter code in C that marshals between values handled by the FFI and the C `struct`.

The conclusion is: when designing an interface to a foreign library, you should analyze the values manipulated on the foreign side and identify their relationship with values on the Scheme side. After you have identified the domains of interest, you then describe how the values will be marshaled back and forth between the two domains.

11.19.3. Marshalling via ffi-attributes

This section describes the marshalling protocol defined in `lib/Base/std-ffi.sch`.

Foreign functions automatically marshal their inputs and outputs according to type-descriptors attached to each foreign function.

Type-descriptors are S-expressons formed according to the following grammar:

```
TypeDesc ::= CoreAttr | ArrowT | MaybeT | OneOfT

CoreAttr ::= PrimAttr | VoidStar | ---

PrimAttr ::= CurrentPrimAttr | DeprecatedPrimAttr

CurrentPrimAttr
         ::= int | uint | byte | short | ushort | char | uchar
          |  long | ulong | longlong | ulonglong
          |  size_t | float | double |  bool | string | void

DeprecatedPrimAttr
         ::= unsigned | boxed

VoidStar ::= void* | ---

ArrowT   ::= (-> (TypeDesc ...) TypeDesc)

MaybeT   ::= (maybe TypeDesc)

OneOfT   ::= (oneof (Any Fixnum) ... TypeDesc)
```

where `---` represents a user-extensible part of the grammar (see below), `Any` represents any Scheme value, and `Fixnum` represents any word-sized integer.

A central registry maps `CoreAttr`'s to a foreign representation and two conversion routines: one to convert a Scheme value to a foreign argument, and another to convert a foreign result back back to a Scheme value. The denoted components are collectively referred to as a *type* within the FFI documentation. The registry is extensible; the `ffi-add-attribute-core-entry!` procedure adds new `CoreAttr's` to the registry, and one can alternatively add short-hands for type-descriptors via the `ffi-add-alias-of-attribute-entry!` procedure. Finally, one can add new `VoidStar` productions (subtypes of the `void*` type-descriptor) via the `ffi-install-void*-subtype` procedure (defined in the `lib/Standard/foreign-stdlib.sch` library).

11.19.3.1. Primitive Attribute Types

The following is a list of the accepted types and their conversions at the boundary between Scheme and foreign code:

`int`
: Exact integer values in the range [-2^{31},2^{31}-1]. Scheme integers in that range are converted to and from C "`int`".

`uint`
: Exact integer values in the range [0,2^{32}-1]. Scheme integers in that ranges are converted to and from C "`unsigned int`".

`byte`
: Synonymous with `int` in the current implementation.

`short`
: Synonymous with `int` in the current implementation.

`ushort`
: Synonymous with `unsigned` in the current implementation.

`char`
: Scheme ASCII characters are converted to and from C "`char`".

`uchar`
: Scheme ASCII characters are converted to and from C "`unsigned char`".

`long`
: Synonymous with `int` in the current implementation.

`ulong`
: Synonymous with `unsigned` in the current implementation.

`longlong`
: Exact integer values in the range [-2^{63},2^{63}-1]. Scheme integers in that range are converted to and from C "`long long`".

`ulonglong`
: Exact integer values in the range [0,2^{64}-1]. Scheme integers in that range are converted to and from C "`unsigned long long`".

`size_t`
: Synonymous with `uint` in the current implementation.

`float`
: Scheme flonums are converted to and from C "`float`". The conversion to `float` is performed via a C `(float)` cast from a C `double`.

`double`
: Scheme flonums are converted to and from C "double".

`bool`
: Scheme objects are converted to C "`int`"; `#f` is converted to 0, and all other objects to 1. In the reverse direction, 0 is converted to `#f` and all other integers to `#t`.

`string`
: A Scheme string holding ASCII characters is *copied* into a NUL-terminated bytevector, passing a pointer to its first byte to the foreign procedure; `#f` is converted to a C "`(char*)0`" value. In the reverse direction, a pointer to a NUL-terminated sequence of bytes interpreted as ASCII characters is copied into a freshly allocated Scheme string; a NULL pointer is converted to `#f`.

`void`
: No return value. (Only used in return position for foreign functions; all Scheme procedures passed to the FFI are invoked in a context expecting one value.)

`unsigned`

Synonymous with `uint`; deprecated.

`boxed`

Any heap-allocated data structure (pair, bytevector-like, vector-like, procedure) is converted to a C "`void*`" to the first element of the structure. The value `#f` is also acceptable. It is converted to a C "`(void*)0`" value. (Only used in argument position for foreign functions; foreign functions are not expected to return direct references to heap-allocated values.)

11.19.3.2. Extending the Core Attribute Registry

The public interface to many foreign libraries is written in terms of types defined within that foreign library. One can introduce new types to the Larceny FFI by extending the core attribute entry table.

Procedure ffi-add-attribute-core-entry!

`(ffi-add-attribute-core-entry! `*`entry-name rep-sym marshal unmarshal`*`) => `*`unspecified`*

ffi-add-attribute-core-entry! extends the internal registry with the new entry specified by its arguments.

- *entry-name* is a symbol (the symbolic type name being introduced to the ffi).
- *rep-name* is a low-level type descriptor symbol, one of `signed32`, `unsigned32`, `signed64`, `unsigned64` (representing varieties of fixed width integers), `ieee32` (representing “floats”), `ieee64` (representing “doubles”), or `pointer` (representing “`(void*)`” in C).
- *marshal* is a marshaling function that accepts a Scheme object and a symbol (the name of the invoking procedure); it is responsible for checking the Scheme object's validity and then producing a corresponding instance of the low-level representation.
- *unmarshal* is either `#f` or an unmarshalling function that accepts an instance of the low-level representation and produces a corresponding Scheme object.

11.19.3.3. Attribute Type Constructors

Core attributes suffice for linking to simple functions. Constructured FFI attributes express more complex marshaling protocols

Arrow Type Constructors. A structured FFI attribute of the form `(-> (`*`s_1 … s_n`*`) `*`s_r`*`)` (called an *arrow type*) allows passing functions from Scheme to C and back again. Each of the *s_1*, …, *s_n*, *s_r* is an FFI attribute. When an arrow type describes an input to a foreign function, it marshals a Scheme procedure to a C function pointer by generating glue code to hook the two together and marshal values as described by the FFI attributes within the arrow type. Likewise, when an arrow type describes an output from a foreign function, it marshals a C function pointer to a Scheme procedure, again by generating glue code. These two mappings naturally generalize to arbitrary nesting of arrow types, so one can create callbacks that consume callouts, return callouts that consume callbacks, and so on.

Warning

The current implementation of arrow types introduces an unnecessary space leak, because none of Larceny's current garbage collectors attempt to reclaim some of the structure allocated (in particular, the so-called trampolines) when functions are marshaled via arrow types.

> The FFI could be revised to reduce the leak (e.g. it could keep a cache of generated trampolines and reuse them, but currently do not do so).
>
> Many foreign libraries have a structure where one only sets up a fixed set of callbacks, and then all further computation does not require arrow type marshaling. This is one reason why fixing this problem has been a low priority item for the Larceny development team.

Maybe Type Constructor. `(maybe *t*)` captures the pattern of passing `NULL` in C and `#f` in Scheme to represent the absence of information. The FFI attribute *t* within the maybe type describes the typical information passed; the constructed maybe type marshals `#f` to the foreign null pointer or `0` (as appropriate), and otherwise applies the marshaling of *t*. Likewise, it unmarshals the foreign null pointer and `0` to `#f`, and otherwise applies the unmarshaling of *t*.

(There are a few other built-in type constructors, such as the `oneof` type constructor, but they are not as fully-developed as the two above, and are intended for use only for internal development for now.)

11.19.3.4. void* Type Hierarchies

Using the `void*` attribute wraps foreign addresses up in a Larceny record, so that standard numeric operations cannot be directly applied by accident. The FFI uses two features of Larceny's record system: the record type descriptor is a first class value with an inspectable name, and record types are extensible via single-inheritance.

Basic Operations on `void*`. The FFI provides `void*-rt`, a record type descriptor with a single field (a wrapped address). There is also a family of functions for dereferencing the pointer within a `void*-rt` and manipulating the state it references.

Procedure void->address*

`(void*->address x) => number`
Extracts the underlying address held in a `void*`.

Procedure void?*

`(void*? x) => boolean`
Distinquishes `void*`'s from other Scheme values.

Procedure void-byte-ref*

`(void*-byte-ref x idx) => number`
Extracts byte at offset from address within *x*.

Procedure void-byte-set!*

`(void*-byte-set! x idx val) => unspecified`
Modifies byte at offset from address within *x*.

Procedure void-word-ref*

`(void*-word-ref x idx) => number`
Extracts word-sized integer at offset from address within *x*.

Procedure void-word-set!*

`(void*-word-set!` *`x idx val`*`) =>` *`unspecified`*
Modifies word-sized integer at offset from address within *x*.

Procedure void-void*-ref*

`(void*-void*-ref` *`x idx`*`) =>` *`void*`*
Extracts address (and wraps it in a `void*`) at offset from address within *x*.

Procedure void-void*-set!*

`(void*-void*-set!` *`x idx val`*`) =>` *`unspecified`*
Modifies address at offset from address within *x*.

Procedure void-double-ref*

`(void*-double-ref` *`x idx`*`) =>` *`number`*
Extracts 64-bit flonum at offset from address within *x*.

Procedure void-double-set!*

`(void*-double-set!` *`x idx val`*`) =>` *`unspecified`*
Modifies 64-bit flonum at offset from address within *x*.

Type Hierarchies. Procedures for establishing type hierarchies are provided by the `lib/Standard/foreign-stdlib.sch` library; see ffi-install-void*-subtype and establish-void*-subhierarchy!.

11.19.4. Creating loadable modules

You must first compile your C code and create one or more loadable object modules. These object modules may then be loaded into Larceny, and Scheme foreign functions may link to specific functions in the loaded module. Defining foreign functions in Scheme is covered in a later section.

The method for creating a loadable object module varies from platform to platform. In the following, assume you have to C source files file1.c and file2.c that define functions that you want to make available as foreign functions in Larceny.

11.19.4.1. SunOS 4

Compile your source files and create a shared library. Using GCC, the command line might look like this:

```
gcc -fPIC -shared file1.c file2.c -o my-library.so
```

The command creates my-library.so in the current directory. This library can now be loaded into Larceny using foreign-file. Any other shared libraries used by your library files should also be loaded into Larceny using foreign-file before any procedures are linked using foreign-procedure.

By default, /lib/libc.so is made available to the dynamic linker and to the foreign function interface, so there is no need for you to load that library explicitly.

11.19.4.2. SunOS 5

Compile your source files and create a shared library, linking with all the necessary libraries. Using GCC, the command line might look like this:

```
gcc -fPIC -shared file1.c file2.c -lc -lm -lsocket -o my-library.so
```

Now you can use foreign-file to load my-library.so into Larceny.

By default, /lib/libc.so is made available to the foreign function interface, so there is no need for you to load that library explicitly.

11.19.5. The Interface

11.19.5.1. Procedures

Procedure foreign-file

`(foreign-file *filename*) => *unspecified*`

foreign-file loads the named object file into Larceny and makes it available for dynamic linking.

Larceny uses the operating system provided dynamic linker to do dynamic linking. The operation of the dynamic linker varies from platform to platform:

- On some versions of SunOS 4, if the linker is given a file that does not exist, it will terminate the process. (Most likely this is a bug.) This means you should never call foreign-file with the name of a file that does not exist.
- On SunOS 5, if a foreign file is given to foreign-file without a directory specification, then the dynamic linker will search its load path (the `LD_LIBRARY_PATH` environment variable) for the file. Hence, a foreign file in the current directory should be "./file.so", not "file.so".

Procedure foreign-procedure

`(foreign-procedure *name (arg-type …) return-type*) => *unspecified*`

FIXME: The interface to this function has been extended to support hooking into Windows procedures that use the Pascal calling convention instead of the C one. The way to select which convention to use should be documented.

Returns a Scheme procedure *p* that calls the foreign procedure whose name is *name*. When *p* is called, it will convert its parameters to representations indicated by the *arg-type*s and invoke the foreign procedure, passing the converted values as parameters. When the foreign procedure returns, its return value is converted to a Scheme value according to *return-type*.

Types are described below.

The address of the foreign procedure is obtained by searching for *name* in the symbol tables of the foreign files that have been loaded with *foreign-file*.

Procedure foreign-null-pointer

`(foreign-null-pointer ) => *integer*`

Returns a foreign null pointer.

Procedure foreign-null-pointer?

```
(foreign-null-pointer? integer) => boolean
```

Tests whether its argument is a foreign null pointer.

11.19.6. Foreign Data Access

11.19.6.1. Raw memory access

The two primitives *peek-bytes* and *poke-bytes* are provided for reading and writing memory at specific addresses. These procedures are typically used for copying data from foreign data structures into Scheme bytevectors for subsequent decoding.

(The use of *peek-bytes* and *poke-bytes* can often be avoided by keeping foreign data in a Scheme bytevector and passing the bytevector to a call-out using the **boxed** parameter type. However, this technique is inappropriate if the foreign code retains a pointer to the Scheme datum, which may be moved by the garbage collector.)

Procedure peek-bytes

```
(peek-bytes addr bytevector count) => unspecified
```

Addr must be an exact nonnegative integer. *Count* must be a fixnum. The bytes in the range from *addr* through *addr+count-1* are copied into *bytevector*, which must be long enough to hold that many bytes.

If any address in the range is not an address accessible to the process, unpredictable things may happen. Typically, you'll get a segmentation fault. Larceny does not yet catch segmentation faults.

Procedure poke-bytes

```
(poke-bytes addr bytevector count) => unspecified
```

Addr must be an exact nonnegative integer. *Count* must be a fixnum. The *count* first bytes from *bytevector* are copied into memory in the range from *addr* through *addr+count-1*.

If any address in the range is not an address accessible to the process, unpredictable things may happen. Typically, you'll get a segmentation fault. Larceny does not yet catch segmentation faults.

Also, it's possible to corrupt memory with *poke-bytes*. Don't do that.

11.19.6.2. Foreign data sizes

The following variables constants define the sizes of basic C data types:

- **sizeof:short** The size of a "short int".
- **sizeof:int** The size of an "int".
- **sizeof:long** The size of a "long int".

- **sizeof:pointer** The size of any pointer type.

11.19.6.3. Decoding foreign data

Foreign data is visible to a Scheme program either as an object pointed to by a memory address (which is itself represented as an integer), or as a bytevector that contains the bytes of the foreign datum.

A number of utility procedures that make reading and writing data of common C primitive types have been written for both these kinds of foreign objects.

Bytevector accessor procedures

`(%get16 `*`bv i`*`) => `*`integer`*

`(%get16u `*`bv i`*`) => `*`integer`*

`(%get32 `*`bv i`*`) => `*`integer`*

`(%get32u `*`bv i`*`) => `*`integer`*

`(%get-int `*`bv i`*`) => `*`integer`*

`(%get-unsigned `*`bv i`*`) => `*`integer`*

`(%get-short `*`bv i`*`) => `*`integer`*

`(%get-ushort `*`bv i`*`) => `*`integer`*

`(%get-long `*`bv i`*`) => `*`integer`*

`(%get-ulong `*`bv i`*`) => `*`integer`*

`(%get-pointer `*`bv i`*`) => `*`integer`*

These procedures decode bytevectors that contain the bytes of foreign objects. In each case, *bv* is a bytevector and *i* is the offset of the first byte of a field in that bytevector. The field is fetched and returned as an integer (signed or unsigned as appropriate).

Bytevector updater procedures

`(%set16 `*`bv i val`*`) => `*`unspecified`*

`(%set16u `*`bv i val`*`) => `*`unspecified`*

`(%set32 `*`bv i val`*`) => `*`unspecified`*

`(%set32u `*`bv i val`*`) => `*`unspecified`*

`(%set-int `*`bv i val`*`) => `*`unspecified`*

`(%set-unsigned `*`bv i val`*`) => `*`unspecified`*

`(%set-short` *`bv i val`*`) =>` *`unspecified`*

`(%set-ushort` *`bv i val`*`) =>` *`unspecified`*

`(%set-long` *`bv i val`*`) =>` *`unspecified`*

`(%set-ulong` *`bv i val`*`) =>` *`unspecified`*

`(%set-pointer` *`bv i val`*`) =>` *`unspecified`*

These procedures update bytevectors that contain the bytes of foreign objects. In each case, *bv* is a bytevector, *i* is an offset of the first byte of a field in that bytevector, and *val* is a value to be stored in that field. The values must be exact integers in a range implied by the data type.

Foreign-pointer accessor procedures

`(%peek8` *`addr`*`) =>` *`integer`*

`(%peek8u` *`addr`*`) =>` *`integer`*

`(%peek16` *`addr`*`) =>` *`integer`*

`(%peek16u` *`addr`*`) =>` *`integer`*

`(%peek32` *`addr`*`) =>` *`integer`*

`(%peek32u` *`addr`*`) =>` *`integer`*

`(%peek-int` *`addr`*`) =>` *`integer`*

`(%peek-long` *`addr`*`) =>` *`integer`*

`(%peek-unsigned` *`addr`*`) =>` *`integer`*

`(%peek-ulong` *`addr`*`) =>` *`integer`*

`(%peek-short` *`addr`*`) =>` *`integer`*

`(%peek-ushort` *`addr`*`) =>` *`integer`*

`(%peek-pointer` *`addr`*`) =>` *`integer`*

`(%peek-string` *`addr`*`) =>` *`integer`*

These procedures read raw memory. In each case, *addr* is an address, and the value stored at that address (the size of which is indicated by the name of the procedure) is fetched and returned as an integer.

%Peek-string expects to find a NUL-terminated string of 8-bit bytes at the given address. It is returned as a Scheme string.

Foreign-pointer updater procedures

`(%poke8` *`addr val`*`) =>` *`unspecified`*

`(%poke8u` *`addr val`*`) =>` *`unspecified`*

`(%poke16` *`addr val`*`) =>` *`unspecified`*

`(%poke16u` *`addr val`*`) =>` *`unspecified`*

`(%poke32` *`addr val`*`) =>` *`unspecified`*

`(%poke32u` *`addr val`*`) =>` *`unspecified`*

`(%poke-int` *`addr val`*`) =>` *`unspecified`*

`(%poke-long` *`addr val`*`) =>` *`unspecified`*

`(%poke-unsigned` *`addr val`*`) =>` *`unspecified`*

`(%poke-ulong` *`addr val`*`) =>` *`unspecified`*

`(%poke-short` *`addr val`*`) =>` *`unspecified`*

`(%poke-ushort` *`addr val`*`) =>` *`unspecified`*

`(%poke-pointer` *`addr val`*`) =>` *`unspecified`*

These procedures update raw memory. In each case, *addr* is an address, and *val* is a value to be stored at that address.

11.19.7. Heap dumping and the FFI

If foreign functions are linked into Larceny using the FFI, and a Larceny heap image is subsequently dumped (with dump-interactive-heap or dump-heap), then the foreign functions are not saved as part of the heap image. When the heap image is subsequently loaded into Larceny at startup, the FFI will attempt to re-link all the foreign functions in the heap image.

During the relinking phase, foreign files will again be loaded into Larceny, and Larceny's FFI will use the file names *as they were originally given to the FFI* when it tries to load the files. In particular, if relative pathnames were used, Larceny will not have converted them to absolute pathnames.

An error during relinking will result in Larceny aborting with an error message and returning to the operating system. This is considered a feature.

11.19.8. Examples

11.19.8.1. Change directory

This procedure uses the chdir() system call to set the process's current working directory. The string parameter type is used to pass a Scheme string to the C procedure.

```
(define cd
  (let ((chdir (foreign-procedure "chdir" '(string) 'int)))
    (lambda (newdir)
      (if (not (zero? (chdir newdir)))
      (error "cd: " newdir " is not a valid directory name."))
      (unspecified))))
```

11.19.8.2. Print Working Directory

This procedure uses the getcwd() (get current working directory) system call to retrieve the name of the process's current working directory. A bytevector is created and passed in as a buffer in which to store the return value — a 0-terminated ASCII string. Then the FFI utility function ffi/asciiz->string is called to convert the bytevector to a string.

```
(define pwd
  (let ((getcwd (foreign-procedure "getcwd" '(boxed int) 'int)))
    (lambda ()
      (let ((s (make-bytevector 1024)))
    (getcwd s 1024)
    (ffi/asciiz->string s)))))
```

11.19.8.3. Quicksort

Warning

this example is bogus. It is not safe to pass a collectable object into a C procedure when the callback invocation might cause a garbage collection, thus moving the object and invalidating the address stored in the C machine context.

This demonstrates how to use a callback such as the comparator argument to qsort. It is specified in the type signature using -> as a type constructor. (Note that one should probably use the built-in sort routines rather than call out like this; this example is for demonstrating callbacks, not how to sort.)

```
(define qsort!
  (foreign-procedure "qsort" '(boxed ushort ushort (-> (void* void*) int)) 'void))
```

```
(let ((bv (list->vector '(40 10 30 20 1 2 3 4))))
  (qsort! bv 8 4
          (lambda (x y)
            (let ((x (/ (void*-word-ref x 0) 4))
                  (y (/ (void*-word-ref y 0) 4)))
              (- x y))))
  bv)
```

```
(let ((bv (list->bytevector '(40 10 30 20 1 2 3 4))))
  (qsort! bv 8 1
          (lambda (x y)
            (let ((x (void*-byte-ref x 0))
                  (y (void*-byte-ref y 0)))
              (- x y))))
  bv)
```

11.19.8.4. Other examples

The Experimental directory contains several examples of use of the FFI. See in particular the files unix.sch (Unix system calls) and socket.sch (procedures for communicating over sockets).

11.19.9. Higher level layers

The general foreign-function interface functionality described above is powerful but awkward to use in practice. A user might be tempted to hard code values of offsets or constants that are compiler dependent. Also, the FFI will marshall some low-level values such as strings or integers, but other values such as enumerations which could be naturally mapped to sets of symbols are not marshalled since the host environment does not provide the necessary type information to the FFI.

This section documents a collection of libraries to mitigate these and other problems.

11.19.9.1. foreign-ctools

Foreign data access is performed by peeking at manually calculated addresses, but in practice one often needs to inspect fields of C structures, whose offsets are dependant on the application binary interface (ABI) of the host environment. Similarly, C programs often use refer to values via constant macro definitions; since the values of such names are not provided by the object code and Scheme programs do not have a C preprocessor run on them prior to execution, it is difficult to refer to the same value without encoding "magic numbers" into the Scheme source code.

The foreign-ctools library is meant to mitigate problems like the two described above. It provides special forms for introducing global definitions of values typically available at compile-time for a C program. The library assumes the presence of a C compiler (such as *cc* on Unix systems or *cl.exe* on Windows systems). The special forms work by dynamically generating, compiling, and running C code at expansion time to determine the desired values of structure offsets or macro constants.

Here is a grammar for the `define-c-info` form provided by the `foreign-ctools` library.

```
<exp>     ::= (define-c-info <c-decl> ... <c-defn> ...)

<c-decl>  ::= (compiler <cc-spec>)
           |  (path <include-path>)
           |  (include <header>)
           |  (include<> <header>)

<cc-spec> ::= cc | cl

<c-defn>  ::= (const <id> <c-type> <c-expr>)
           |  (sizeof <id> <c-type-expr>)
           |  (struct <c-name> <field-clause> ...)
           |  (fields <c-name> <field-clause> ...)
           |  (ifdefconst <id> <c-type> <c-name>)

<c-type>  ::= int | uint | long | ulong

<include-path>
          ::= <string-literal>

<header>  ::= <string-literal>

<field-clause>
          ::= (<offset-id> <c-field>)
           |  (<offset-id> <c-field> <size-id>)

<c-expr>  ::= <string-literal>

<c-type-expr>
          ::= <string-literal>

<c-name>  ::= <string-literal>

<c-field> ::= <string-literal>
```

Syntax define-c-info

```
(define-c-info <c-decl> … <c-defn> …)
```

The `<c-decl>` clauses of `define-c-info` control how header files are processed. The `compiler` clause selects between `cc` (the default UNIX system compiler) and `cl` (the compiler included with Microsoft's Windows SDK). The `path` clause adds a directory to search when looking for header files. The `include` and `include<>` clauses indicate header files to include when executing the `<c-defn>` clauses; the two variants correspond to the quoted and bracketed forms of the C preprocessor's `#include` directive.

The `<c-defn>` clauses bind identifiers. A `(const` *`x t`* `"`*`ae`*`")` clause binds *x* to the integer value of *ae* according to the C language; *ae* can be any C arithmetic expression that evaluates to a value of type *t*. (The expected usage is for *ae* to be an expression that the C preprocessor expands to an arithmetic expression.)

The remaining clauses provide similar functionality:

- `(sizeof` *`x`* `"`*`te`*`")` binds *x* to the size occupied by values of type *te*, where *te* is any C type expression.
- `(struct "`*`cn`*`" … (`*`x`* `"`*`cf`*`"` *`y`*`) …)` binds *x* to the offset from the start of a structure of type `struct` *`cn`* to its *cf* field, and binds *y*, if present, to the field's size. A `fields` clause is similar, but it applies to structures of type *`cn`* rather than `struct` *`cn`*.
- `(ifdefconst` *`x t`* `"`*`cn`*`")` binds *x* to the value of *`cn`* if *`cn`* is defined; *x* is otherwise bound to Larceny's unspecified value.

11.19.9.2. foreign-sugar

The foreign-procedure function is sufficient to link in dynamically loaded C procedures, but it can be annoying to use when there are many procedures to define that all follow a regular pattern where one could infer a mapping between Scheme identifiers and C function names.

For example, some libraries follow a naming convention where a words within a name are separated by underscores; such functions could be immediately mapped to Scheme names where the underscores have been replaced by dashes.

The foreign-sugar library provides a special form, `define-foreign`, which gives the user a syntax for defining foreign functions using a syntax where one provides only the Scheme name, the argument types, and the return type. The `define-foreign` form then attempts to infer what C function the name was meant to refer to.

Syntax define-foreign

```
(define-foreign (name arg-type …) result-type)
```

Note

There is other functionality provided allowing the user to introduce new rules for inferring C function names, but they are undocumented because they will probably have to change when we switch to an R6RS macro expander.

11.19.9.3. foreign-stdlib

Procedure stdlib/malloc

`(stdlib/malloc` *`rtd [ctor]`*`) =>` *`procedure`*

Given a record extension of *void*-rt*, returns an allocator that uses the C `malloc` procedure to allocate instances of such an object. Note that the client is responsible for eventually freeing such objects with stdlib/free.

Procedure stdlib/free

`(stdlib/free` *`void*-obj`*`)`

Frees objects produced by allocators returned from stdlib/malloc.

Procedure ffi-install-void-subtype*

`(ffi-install-void*-subtype` *`rtd`*`) =>` *`rtd`*

`(ffi-install-void*-subtype` *`string [parent-rtd]`*`) =>` *`rtd`*

`(ffi-install-void*-subtype` *`symbol [parent-rtd]`*`) =>` *`rtd`*

ffi-install-void*-subtype extends the core attribute registry with a new primitive entry for *subtype*. The *parent-rtd* argument should be a subtype of `void*-rt` and defaults to `void*-rt`. In the case of the *symbol* or *string* inputs, the procedure constructs a new record type subtyping the *parent* argument. In the case of the *rtd* input, the *rtd* record type must extend `void*-rt`. ffi-install-void*-subtype returns the subtype record type.

The returned record type represents a tagged wrapped C pointer, allowing one to encode type hierarchies.

Procedure establish-void-subhierarchy!*

`(establish-void*-subhierarchy!` *`symbol-tree`*`) =>` *`unspecified`*

establish-void*-subhierarchy! is a convenience function for constructing large object hierarchies. It descends the *symbol-tree*, creates a record type descriptor for each symbol (where the root of the tree has the parent `void*-rt`), and invokes ffi-install-void*-subtype on all of the introduced types.

*Type char** extends *void* Procedure string->char**

`(string->char*` *`string`*`) =>` *`char*`*
Procedure char-strlen*

`(char*-strlen` *`char*`*`) =>` *`fixnum`*
Procedure char->string*

`(char*->string` *`char*`*`) =>` *`string`*

`(char*->string` *`char* len`*`) =>` *`string`*
*Procedure call-with-char**

`(call-with-char*` *`string string-function`*`) =>` *`value`*
*Type char*** extends *void* Procedure call-with-char***

`(call-with-char**` *`string-vector function`*`) =>` *`value`*
*Type int** extends *void* Procedure call-with-int**

`(call-with-int* *fixnum-vector function*) => *value*`
*Type short** extends *void* Procedure call-with-short**

`(call-with-short* *fixnum-vector function*) => *value*`
*Type double** extends *void* Procedure call-with-double**

`(call-with-double* *num-vector function*) => *value*`

FIXME: (There are other functions, but I want to test and document the ones above first…)

11.19.9.4. foreign-cstructs

The `foreign-cstructs` library provides a more direct interface to C structures. It provides the `define-c-struct` special form. This form is layered on top of `define-c-info`; the latter provides the structure field offsets and sizes used to generate constructors (which produce appropriately sized bytevectors, not record instances). The `define-c-struct` form combines these with marshaling and unmarshaling procedures to provide high-level access to a structure.

The grammar for the `define-c-struct` form is presented below.

```
<exp>    ::= (define-c-struct (<struct-type> <ctor-id> <c-decl> ...)
               <field-clause> ...)

<field-clause>
         ::= (<c-field> <getter>) | (<c-field> <getter> <setter>)

<getter> ::= (<id>) | (<id> <unmarshal>)

<setter> ::= (<id>) | (<id> <marshal>)

<marshal> ::= <ffi-attr-symbol> | <marshal-proc-exp>

<unmarshal> ::= <ffi-attr-symbol> | <unmarshal-proc-exp>

<struct-type> ::= <string-literal>
```

11.19.9.5. foreign-cenums

This library provides the special forms `define-c-enum` and `define-c-enum-set`, which associate the identifiers of a C `enum` type declaration with the integer values they denote.

The `define-c-enum` form describes enums encoding a discriminated sum; `define-c-enum-set` describes bitmasks, mapping them to R^{6}RS enum-sets in Scheme.

The `(define-c-enum *en* (<c-decl> …) (*x* "*cn*") …)` form adds the *en* FFI attribute. The attribute marshals each symbol *x* to the integer value that *cn* denotes in C; unmarshaling does the inverse translation.

The `(define-c-enum-set *ens* (<c-decl> …) (*x* "*cn*") …)` form binds *ens* to an R^{6}RS enum-set constructor with universe resulting from `(make-enumeration '(*x* …))`; it also adds the *ens* FFI attribute. The attribute marshals an enum-set *s* constructed by *ens* to the corresponding bitmask in C (that is, the integer one would get by logically or'ing all *cn* such that the corresponding *x* is in *s*). Unmarshaling attempts to do the inverse translation.

The grammar for the two forms is presented below.

```
<exp> ::= (define-c-enum <enum-id> (<c-decl> ...)
            (<id> <c-name>) ...)

<exp> ::= (define-c-enum-set <enum-id> (<c-decl> ...)
            (<id> <c-name>) ...)

<enum-id> ::= <id>
```

12. Debugging

Larceny's debugging functionality is implemented in Scheme, using some of Larceny's extensions for catching exceptions and inspecting the continuation structure.

12.1. Entering the debugger

When Larceny detects an error or a keyboard interrupt, or when it hits a breakpoint, it signals the condition by printing a message on the console. Larceny then enters the debugger, which signals its presence with a short banner and the debugger prompt:

```
Entering debugger; type "?" for help.
debug>
```

You can also re-enter the debugger by evaluating (debug).

12.2. Debugger commands

The debugger is still in an immature state. The following commands are available (commands can be typed in upper or lower case):

B Print backtrace of continuation.

C Print source code of procedure, if available.

D Move down to previous (earlier) activation record.

E n expr *Expr* is evaluated in the current interaction environment and must evaluate to a procedure. It is passed the contents of slot *n* from the current activation record, and the result, if not unspecified, is printed.

E (n1 … nk) expr *Expr* is evaluated in the current interaction environment and must evaluate to a procedure. It is passed the contents of slots *n1* through *nk* from the current activation record, and the result, if not unspecified, is printed.

I n Inspect the procedure in slot *n* of the current activation record.

I @_ Inspect the active procedure.

Q Quit the debugger and abort the computation.

R Return from the debugger and continue the computation.

S Summarize the contents of the current activation record.

U Up to the next (later) activation record.

X Examine the contents of the current activation record.

The **B**, **D**, and **U** commands can be prefixed with a count, for example, 5 U moves up five activation records, and 10 B displays the next 10 activation records. The default for **B** is to display all the activations; the default count for **D** and **U** is 1.

12.3. Breakpoints

You can set breakpoints either in program text with the break primitive or interactively at the start of a procedure with the break-entry procedure. When Larceny reaches a breakpoint during execution, the program is suspended and the debugger is entered to allow you to inspect the program.

Procedure larceny-break

```
(larceny-break )
```

Invokes the breakpoint handler.

Procedure break-entry

```
(break-entry procedure)
```

Set a breakpoint at the start of the *procedure*.

Procedure unbreak

```
(unbreak procedure …)


(unbreak )
```

In the first form, remove any breakpoint set by break-entry at the start of the *procedure_s. In the second form, remove all breakpoints set by _break-entry*.

12.4. Tracing

Procedure trace-entry

```
(trace-entry procedure)
```

Set a trace point on entry to the *procedure*, removing any other trace points on the procedure. When the *procedure* is entered, information about the call is printed on the console: the name of the procedure and the actual arguments.

Procedure trace-exit

```
(trace-exit procedure)
```

Set a trace point on exit from the *procedure*, removing any other trace points on the procedure. When the *procedure* returns, information about the return is printed on the console: the name of the procedure and the returned values.

Note that trace-exit destroys the tail recursion properties of the instrumented procedure. Where the *procedure* would normally "return" by tail-calling another procedure, the instrumented procedure will call the other procedure by a non-tail call and then return, at which point the procedure name and return values will be printed. Thus use of trace-exit may destroy the space properties of the program.

Procedure trace

```
(trace procedure)
```

Set trace points on *procedure* both at entry and exit.

Procedure untrace

```
(untrace procedure …)


(untrace )
```

The first form removes any trace points from the specified procedures. The second form removes all untrace points.

12.5. Other functionality

Parameter break-handler

The value of break-handler is a procedure that is called when a breakpoint or tracepoint is encountered. The procedure takes two arguments: the procedure in which the breakpoint was set, and the byte offset within the procedure's code vector of the breakpoint.

13. Standards

13.1. Scheme standards

IEEE Standard 1178-1990, "IEEE Standard for the Scheme Programming Language", IEEE, 1991. ISBN 1-55937-125-0. May be ordered from IEEE by calling 1-800-678-IEEE or 908-981-1393 or writing IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, and using order number SH14209.

Richard Kelsey, William Clinger, and Jonathan Rees [editors]. Revised^5 Report on the Algorithmic Language Scheme [http://www.brics.dk/~hosc/11-1/]. *Journal of Higher Order and Symbolic Computation*, 11(1), 1998, pages 7-105. Also appears in *ACM SIGPLAN Notices* 33(9), September 1998. Available online in various formats [http://www.schemers.org/Documents/Standards/R5RS/].

Michael Sperber, R Kent Dybvig, Matthew Flatt, and Anton van Straaten [editors]. Revised^6 Report on the Algorithmic Language Scheme [http://www.r6rs.org/], 2007.

Alex Shinn, John Cowan, and Arthur A Gleckler [editors]. Revised^7 Report on the Algorithmic Language Scheme [http://www.scheme-reports.org/], 2013.

13.2. Other relevant standards

IEEE Standard 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic", IEEE, 1985.

IEEE Standard 754-2008, "IEEE Standard for Floating-Point Arithmetic", IEEE, 2008. (Revision of IEEE Std 754-1985 [http://en.wikipedia.org/wiki/IEEE_754r] began in 2000. The IEEE Microprocessor Standards Committee (MSC) accepted a candidate draft on 9 October 2006. The candidate draft 1.2.6 was approved by 79% of 70 votes, which exceeded the required supermajority of 75%. Because there were negative votes and several hundred comments, however, a revised draft 1.3.0 was prepared and approved by 84% of 73 votes. Since there were over a hundred comments on the second candidate draft as well, a third candidate draft 1.4.0 was prepared and another vote taken in April 2007. After a total of eight ballots, with the last four being approved by more than 90% of the voters, the Ballot Review Committee decided in May 2008 that maximum possible timely consensus has been obtained [http://www.validlab.com/754R/], and the consensus draft was submitted to IEEE-SA RevCom. IEEE-754-2008 was approved on 12 June 2008.)

The Unicode Consortium. The Unicode Standard [http://www.unicode.org/].

Index