Chez Scheme Version 9
User's Guide

Cisco Systems, Inc.
www.cisco.com

Revised June 2018 for Chez Scheme Version 9.5.1.

Contents

Preface

Chez Scheme is both a general-purpose programming language and an implementation of that language, with supporting tools and documentation. As a superset of the language described in the Revised[6] Report on Scheme (R6RS), *Chez Scheme* supports all standard features of Scheme, including first-class procedures, proper treatment of tail calls, continuations, user-defined records, libraries, exceptions, and hygienic macro expansion. *Chez Scheme* supports numerous non-R6RS features. A few of these are local and top-level modules, local import, foreign datatypes and procedures, nonblocking I/O, an interactive top-level, compile-time values and properties, pretty-printing, and formatted output.

The implementation includes a compiler that generates native code for each processor upon which it runs along with a run-time system that provides automatic storage management, foreign-language interfaces, source-level debugging, profiling support, and an extensive run-time library.

The threaded versions of *Chez Scheme* support native threads, allowing Scheme programs to take advantage of multiprocessor or multiple-core systems. Nonthreaded versions are also available and are faster for single-threaded applications. Both 32-bit and 64-bit versions are available for some platforms. The 64-bit versions support larger heaps, while the 32-bit versions are faster for some applications.

Chez Scheme's interactive programming system includes an expression editor that, like many shells, supports command-line editing, a history mechanism, and command completion. Unlike most shells that support command-line editing, the expression editor properly supports multiline expressions.

Chez Scheme is intended to be as reliable and efficient as possible, with reliability taking precedence over efficiency if necessary. Reliability means behaving as designed and documented. While a *Chez Scheme* program can always fail to work properly because of a bug in the program, it should never fail because of a bug in the *Chez Scheme* implementation. Efficiency means performing at a high level, consuming minimal CPU time and memory. Performance should be balanced across features, across run time and compile time, and across programs and data of different sizes. These principles guide *Chez Scheme* language and tool design as well as choice of implementation technique; for example, a language feature or debugging hook might not exist in *Chez Scheme* because its presence would reduce reliability, efficiency, or both.

The compiler has been rewritten for Version 9 and generates substantially faster code than the earlier compiler at the cost of greater compile time. This is the primary difference between Versions 8 and 9.

This book (CSUG) is a companion to *The Scheme Programming Language, 4th Edition* (TSPL4). TSPL4 serves as an introduction to and reference for R6RS, while CSUG describes *Chez Scheme* features and tools that are not part of R6RS. For the reader's convenience, the summary of forms and index at the back of this book contain entries from both books, with each entry from TSPL4 marked with a "t" in front of its page number. In the online version, the page numbers given in the summary of forms and index double as direct links into one of the documents or the other.

Additional documentation for *Chez Scheme* includes release notes, a manual page, and a number of published papers and articles that describe various aspects of the system's design and implementation.

Thank you for using *Chez Scheme*.

1. Introduction

This book describes *Chez Scheme* extensions to the Revised^6 Report on Scheme [28] (R6RS). It contains as well a concise summary of standard and *Chez Scheme* forms and procedures, which gives the syntax of each form and the number and types of arguments accepted by each procedure. Details on standard R6RS features can be found in *The Scheme Programming Language, 4th Edition* (TSPL4) [11] or the Revised^6 Report on Scheme. *The Scheme Programming Language, 4th Edition* also contains an extensive introduction to the Scheme language and numerous short and extended examples.

Most of this document also applies equally to *Petite Chez Scheme*, which is fully compatible with the complete *Chez Scheme* system but uses a high-speed interpreter in place of *Chez Scheme*'s incremental native-code compiler. Programs written for *Chez Scheme* run unchanged in *Petite Chez Scheme* as long as they do not require the compiler to be invoked. In fact, *Petite Chez Scheme* is built from the same sources as *Chez Scheme*, with all but the compiler sources included. A detailed discussion of the impact of this distinction appears in Section 2.8.

The remainder of this chapter covers *Chez Scheme* extensions to Scheme syntax (Section 1.1), notational conventions used in this book (Section 1.2), the use of parameters for system customization (Section 1.3), and where to look for more information on *Chez Scheme* (Section 1.4).

Chapter 2 describes how one uses *Chez Scheme* for program development, scripting, and application delivery, plus how to get the compiler to generate the most efficient code possible. Chapter 3 describes debugging and object inspection facilities. Chapter 4 documents facilities for interacting with separate processes or code written in other languages. Chapter 5 describes binding forms. Chapter 6 documents control structures. Chapter 7 documents operations on nonnumeric objects, while Chapter 8 documents various numeric operations, including efficient type-specific operations. Chapter 9 describes input/output operations and generic ports, which allow the definition of ports with arbitrary input/output semantics. Chapter 10 discusses how R6RS libraries and top-level programs are loaded into *Chez Scheme* along with various features for controlling and tracking the loading process. Chapter 11 describes syntactic extension and modules. Chapter 12 describes system operations, such as operations for interacting with the operating system and customizing *Chez Scheme*'s user interface. Chapter 13 describes how to invoke and control the storage management system and documents guardians and weak pairs. Chapter 14 describes *Chez Scheme*'s expression editor and how it can be customized. Chapter 15 documents the procedures and syntactic forms that comprise the interface to *Chez Scheme*'s native thread system. Finally, Chapter 16 describes various compatibility features.

The back of this book contains a bibliography, the summary of forms, and an index. The page numbers appearing in the summary of forms and the italicized page numbers appearing in the index indicate the locations in the text where forms and procedures are formally defined. The summary of forms and index includes entries from TSPL4, so that they cover the entire set of *Chez Scheme* features. A TSPL4 entry is marked by a "t" prefix on the page number.

Online versions and errata for this book and for TSPL4 can be found at www.scheme.com.

Acknowledgments: Michael Adams, Mike Ashley, Carl Bruggeman, Bob Burger, Sam Daniel, George Davidson, Matthew Flatt, Aziz Ghuloum, Bob Hieb, Andy Keep, and Oscar Waddell have contributed substantially to the development of *Chez Scheme*. *Chez Scheme*'s expression editor is based on a command-line editor for Scheme developed from 1989 through 1994 by C. David Boyer. File compression is performed with the use of the zlib compression library developed by Jean-loup Gailly and Mark Adler. Implementations of the list and vector sorting routines are based on Olin Shiver's opportunistic merge-sort algorithm and implementation. Michael Lenaghan provided a number of corrections for earlier drafts of this book. Many of the features documented in this book were suggested by current *Chez Scheme* users, and numerous comments from users have also led to improvements in the text. Additional suggestions for improvements to *Chez Scheme* and to this book are welcome.

1.1. Chez Scheme Syntax

Chez Scheme extends Scheme's syntax both at the object (datum) level and at the level of syntactic forms. At the object level, *Chez Scheme* supports additional representations for symbols that contain nonstandard characters, nondecimal numbers expressed in floating-point and scientific notation, vectors with explicit lengths, shared and cyclic structures, records, boxes, and more. These extensions are described below. Form-level extensions are described throughout the book and summarized in the Summary of Forms, which also appears in the back of this book.

Chez Scheme extends the syntax of identifiers in several ways. First, the sequence of characters making up an identifier's name may start with digits, periods, plus signs, and minus signs as long as the sequence cannot be parsed as a number. For example, `0abc`, `+++`, and `..` are all valid identifiers in *Chez Scheme*. Second, the single-character sequences `{` and `}` are identifiers. Third, identifiers containing arbitrary characters may be printed by escaping them with `\` or with `|`. `\` is used to escape a single character (except 'x', since `\x` marks the start of a hex scalar value), whereas `|` is used to escape the group of characters that follow it up through the matching `|`. For example, `\|\|` is an identifier with a two-character name consisting of the character `|` followed by the character `\`, and `|hit me!|` is an identifier whose name contains a space.

In addition, gensyms (page 7.9) are printed with `#{` and `}` brackets that enclose both the "pretty" and "unique" names, e.g., `#{g1426 e5g1c94g642dssw-a}`. They may also be printed using the pretty name only with the prefix `#:`, e.g., `#:g1426`.

Arbitrary radixes from two through 36 may be specified with the prefix `#`*n*`r`, where *n* is the radix. Case is not significant, so `#`*n*`R` may be used as well. Digit values from 10 through 35 are specified as either lower- or upper-case alphabetic characters, just as for hexadecimal numbers. For example, `#36rZZ` is $35 \times 36 + 35$, or 1295.

Chez Scheme also permits nondecimal numbers to be printed in floating-point or scientific notation. For example, `#o1.4` is equivalent to `1.5`, and `#b1e10` is equivalent to `4.0`. Digits take precedence over exponent specifiers, so that `#x1e20` is simply the four-digit hexadecimal number equivalent to `7712`.

In addition to the standard named characters `#\alarm`, `#\backspace`, `#\delete`, `#\esc`, `#\linefeed`, `#\newline`, `#\page`, `#\return`, `#\space`, and `#\tab`, *Chez Scheme* recognizes `#\bel`, `#\ls`, `#\nel`, `#\nul`, `#\rubout`, and `#\vt` (or `#\vtab`). Characters whose scalar values are less than 256 may also be printed with an octal syntax consisting of the prefix `#\` followed by a three octal-digit sequence. For example, `#\000` is equivalent to `#\nul`.

Chez Scheme's fxvectors, or fixnum vectors, are printed like vectors but with the prefix `#vfx(` in place of `#(`. Vectors, bytevectors, and fxvectors may be printed with an explicit length prefix, and when the explicit length prefix is specified, duplicate trailing elements may be omitted. For example, `#(a b c)` may be printed as `#3(a b c)`, and a vector of length 100 containing all zeros may be printed as `#100(0)`.

Chez Scheme's boxes are printed with a `#&` prefix, e.g., `#&17` is a box containing the integer `17`.

Records are printed with the syntax `#[`*type-name field* `...]`, where the symbol *type-name* is the name of the record type and *field* `...` are the printed representations for the contents of the fields of the record.

Shared and cyclic structure may be printed using the graph mark and reference prefixes `#`*n*`=` and `#`*n*`#`. `#`*n*`=` is used to mark an item in the input, and `#`*n*`#` is used to refer to the item marked *n*. For example, `'(#1=(a) . #1#)` is a pair whose car and cdr contain the same list, and `#0=(a . #0#)` is a cyclic list, i.e., its cdr is itself.

A `$primitive` form (see page 348) may be abbreviated in the same manner as a `quote` form, using the `#%` prefix. For example, `#%car` is equivalent to `($primitive car)`, `#2%car` to `($primitive 2 car)`, and `#3%car` to `($primitive 3 car)`.

Chez Scheme's end-of-file object is printed `#!eof`. If the end-of-file object appears outside of any datum within a file being loaded, `load` will treat it as if it were a true end of file and stop loading at that point. Inserting `#!eof` into the middle of a file can thus be handy when tracking down a load-time error.

Broken pointers in weak pairs (see page 395) are represented by the *broken weak pointer* object, which is printed `#!bwp`.

In addition to the standard delimiters (whitespace, open and close parentheses, open and close brackets, double quotes, semi-colon, and `#`), *Chez Scheme* also treats as delimiters open and close braces, single quote, backward quote, and comma.

The *Chez Scheme* lexical extensions described above are disabled in an input stream after an `#!r6rs` comment directive has been seen, unless a `#!chezscheme` comment directive has been seen since. Each library loaded implicitly via `import` and each RNRS top-level

program loaded via the `--program` command-line option, the `scheme-script` command, or the `load-program` procedure is treated as if it begins implicitly with an `#!r6rs` comment directive.

The case of symbol and character names is normally significant, as required by the Revised[6] Report. Names are folded, as if by `string-foldcase`, following a `#!fold-case` comment directive in the same input stream unless a `#!no-fold-case` has been seen since. Names are also folded if neither directive has been seen and the parameter `case-sensitive` has been set to `#f`.

The printer invoked by `write`, `put-datum`, `pretty-print`, and the `format ~s` option always prints standard Revised[6] Report objects using the standard syntax, unless a different behavior is requested via the setting of one of the print parameters. For example, it prints symbols in the extended identifier syntax of Chez Scheme described above using hex scalar value escapes, unless the parameter `print-extended-identifiers` is set to true. Similarly, it does not print the explicit length or suppress duplicate trailing elements unless the parameter `print-vector-length` is set to true.

1.2. Notational Conventions

This book follows essentially the same notational conventions as *The Scheme Programming Language, 4th Edition*. These conventions are repeated below, with notes specific to *Chez Scheme*.

When the value produced by a procedure or syntactic form is said to be *unspecified*, the form or procedure may return any number of values, each of which may be any Scheme object. *Chez Scheme* usually returns a single, unique *void* object (see `void`) whenever the result is unspecified; avoid counting on this behavior, however, especially if your program may be ported to another Scheme implementation. Printing of the void object is suppressed by *Chez Scheme*'s waiter (read-evaluate-print loop).

This book uses the words "must" and "should" to describe program requirements, such as the requirement to provide an index that is less than the length of the vector in a call to `vector-ref`. If the word "must" is used, it means that the requirement is enforced by the implementation, i.e., an exception is raised, usually with condition type `&assertion`. If the word "should" is used, an exception may or may not be raised, and if not, the behavior of the program is undefined. The phrase "syntax violation" is used to describe a situation in which a program is malformed. Syntax violations are detected prior to program execution. When a syntax violation is detected, an exception of type `&syntax` is raised and the program is not executed.

Scheme objects are displayed in a `typewriter` typeface just as they are to be typed at the keyboard. This includes identifiers, constant objects, parenthesized Scheme expressions, and whole programs. An *italic* typeface is used to set off syntax variables in the descriptions of syntactic forms and arguments in the descriptions of procedures. Italics are also used to set off technical terms the first time they appear. The first letter of an identifier that is not ordinarily capitalized is not capitalized when it appears at the beginning of a sentence. The same is true for syntax variables written in italics.

In the description of a syntactic form or procedure, a pattern shows the syntactic form or the application of the procedure. The syntax keyword or procedure name is given in typewriter font, as are parentheses. The remaining pieces of the syntax or arguments are shown in italics, using names that imply the types of the expressions or arguments expected by the syntactic form or procedure. Ellipses are used to specify zero or more occurrences of a subexpression or argument.

1.3. Parameters

All *Chez Scheme* system customization is done via *parameters*. A parameter is a procedure that encapsulates a hidden state variable. When invoked without arguments, a parameter returns the value of the encapsulated variable. When invoked with one argument, the parameter changes the value of the variable to the value of its argument. A parameter may raise an exception if its argument is not appropriate, or it may filter the argument in some way.

New parameters may be created and used by programs running in *Chez Scheme*. Parameters are used rather than global variables for program customization for two reasons: First, unintentional redefinition of a customization variable can cause unexpected problems, whereas unintentional redefinition of a parameter simply makes the parameter inaccessible. For example, a program that defines `*print-level*` for its own purposes in early releases of *Chez Scheme* would have unexpected effects on the printing of Scheme objects, whereas a program that defines `print-level` for its own purposes simply loses the ability to alter the printer's behavior. Of course, a program that invokes `print-level` by accident can still affect the system in unintended ways, but such an occurrence is less likely, and can only happen in an incorrect program.

Second, invalid values for parameters can be detected and rejected immediately when the "assignment" is made, rather than at the point where the first use occurs, when it is too late to recover and reinstate the old value. For example, an assignment of `*print-level*` to -1 would not have been caught until the first call to `write` or `pretty-print`, whereas an attempted assignment of -1 to the parameter `print-level`, i.e., `(print-level -1)`, is flagged as an error immediately, before the change is actually made.

Built-in system parameters are described in different sections throughout this book and are listed along with other syntactic forms and procedures in the Summary of Forms in the back of this book. Parameters marked "thread parameters" have per-thread values in threaded versions of *Chez Scheme*, while the values of parameters marked "global parameters" are shared by all threads. Nonthreaded versions of *Chez Scheme* do not distinguish between thread and global parameters. See Sections 12.13 and 15.6 for more information on creating and manipulating parameters.

1.4. More Information

The articles and technical reports listed below document various features of *Chez Scheme* and its implementation:

- syntactic abstraction [14, 8, 17],
- modules [32],
- libraries [21],
- storage management [12, 13],
- threads [10],
- multiple return values [2],
- optional arguments [16],
- continuations [7, 25, 3],
- eq? hashtables [20],
- internal definitions, `letrec`, and `letrec*` [33, 22],
- `equal?` [1],
- engines [15],
- floating-point printing [4],
- code generation [18],
- register allocation [6],
- procedure inlining [31],
- profiling [5], and
- history of the implementation [9].

Links to abstracts and electronic versions of these publications are available at the url http://www.cs.indiana.edu/chezscheme/pubs/.

2. Using Chez Scheme

Chez Scheme is often used interactively to support program development and debugging, yet it may also be used to create stand-alone applications with no interactive component. This chapter describes the various ways in which *Chez Scheme* is typically used and, more generally, how to get the most out of the system. Sections 2.1, 2.2, and 2.3 describe how one uses *Chez Scheme* interactively. Section 2.4 discusses how libraries and RNRS top-level programs are used in *Chez Scheme*. Section 2.5 covers support for writing and running Scheme scripts, including compiled scripts and compiled RNRS top-level programs. Section 2.6 describes how to structure and compile an application to get the most efficient code possible out of the compiler. Section 2.7 describes how one can customize the startup process, e.g., to alter or eliminate the command-line options, to preload Scheme or foreign code, or to run *Chez Scheme* as a subordinate program of another program. Section 2.8 describes how to build applications using *Chez Scheme* with *Petite Chez Scheme* for run-time support. Finally, Section 2.9 covers command-line options used when invoking *Chez Scheme*.

2.1. Interacting with Chez Scheme

One of the simplest and most effective ways to write and test Scheme programs is to compose them using a text editor, like `vi` or `emacs`, and test them interactively with *Chez Scheme* running in a shell window. When *Chez Scheme* is installed with default options, entering the command `scheme` at the shell's prompt starts an interactive Scheme session. The command `petite` does the same for *Petite Chez Scheme*. After entering this command, you should see a short greeting followed by an angle-bracket on a line by itself, like this:

```
Chez Scheme Version 9.5.1
Copyright 1984-2017 Cisco Systems, Inc.

>
```

You also should see that the cursor is sitting one space to the right of the angle-bracket. The angle-bracket is a prompt issued by the system's "REPL," which stands for "Read Eval Print Loop," so called because it reads, evaluates, and prints an expression, then loops back to read, evaluate, and print the next, and so on. (In *Chez Scheme*, the REPL is also called a waiter.)

In response to the prompt, you can type any Scheme expression. If the expression is well-formed, the REPL will run the expression and print the value. Here are a few examples:

```
> 3
3
> (+ 3 4)
7
> (cons 'a '(b c d))
(a b c d)
```

The reader used by the REPL is more sophisticated than an ordinary reader. In fact, it's a full-blown "expression editor" ("expeditor" for short) like a regular text editor but for just one expression at a time. One thing you might soon notice is that the system automatically indents the second and subsequent lines of an expression. For example, let's say we want to define `fact`, a procedure that implements the factorial function. If we type `(define fact` followed by the enter key, the cursor should be sitting under the first `e` in `define`, so that if we then type `(lambda (x)`, we should see:

```
> (define fact
    (lambda (x)
```

The expeditor also allows us to move around within the expression (even across lines) and edit the expression to correct mistakes. After typing:

```
> (define fact
    (lambda (x)
      (if (= n 0)
          0
          (* n (fact
```

we might notice that the procedure's argument is named `x` but we have been referencing it as `n`. We can move back to the second line using the arrow keys, remove the offending `x` with the backspace key, and replace it with `n`.

```
> (define fact
    (lambda (n)
      (if (= n 0)
          0
          (* n (fact
```

We can then return to the end of the expression with the arrow keys and complete the definition.

```
> (define fact
    (lambda (n)
      (if (= n 0)
          0
          (* n (fact (- n 1))))))
```

Now that we have a complete form with balanced parentheses, if we hit enter with the

cursor just after the final parenthesis, the expeditor will send it on to the evaluator. We'll know that it has accepted the definition when we get another right-angle prompt.

Now we can test our definition by entering, say, `(fact 6)` in response to the prompt:

```
> (fact 6)
0
```

The printed value isn't what we'd hoped for, since 6! is actually 720. The problem, of course, is that the base-case return-value `0` should have been `1`. Fortunately, we don't have to retype the definition to correct the mistake. Instead, we can use the expeditor's history mechanism to retrieve the earlier definition. The up-arrow key moves backward through the history. In this case, the first up-arrow retrieves `(fact 6)`, and the second retrieves the `fact` definition.

As we move back through the history, the expression editor shows us only the first line, so after two up arrows, this is all we see of the definition:

```
> (define fact
```

We can force the expeditor to show the entire expression by typing `^L` (control `L`, i.e., the control and `L` keys pressed together):

```
> (define fact
    (lambda (n)
      (if (= n 0)
          0
          (* n (fact (- n 1))))))
```

Now we can move to the fourth line and change the `0` to a `1`.

```
> (define fact
    (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
```

We're now ready to enter the corrected definition. If the cursor is on the fourth line and we hit enter, however, it will just open up a new line between the old fourth and fifth lines. This is useful in other circumstances, but not now. Of course, we can work around this by using the arrow keys to move to the end of the expression, but an easier way is to type `^J`, which forces the expression to be entered immediately no matter where the cursor is.

Finally, we can bring back `(fact 6)` with another two hits of the up-arrow key and try it again:

```
> (fact 6)
720
```

To exit from the REPL and return back to the shell, we can type `^D` or call the `exit` procedure.

The interaction described above uses just a few of the expeditor's features. The expeditor's remaining features are described in the following section.

Running programs may be interrupted by typing the interrupt character (typically `^C`). In response, the system enters a debug handler, which prompts for input with a `break>` prompt. One of several commands may be issued to the break handler (followed by a newline), including

"e" or end-of-file to exit from the handler and continue,

"r" to stop execution and reset to the current café,

"a" to abort *Chez Scheme*,

"n" to enter a new café (see below),

"i" to inspect the current continuation,

"s" to display statistics about the interrupted program, and

"?" to display a list of these options.

When an exception other than a warning occurs, the default exception handler prints a message that describes the exception to the console error port. If a REPL is running, the exception handler then returns to the REPL, where the programmer can call the `debug` procedure to start up the debug handler, if desired. The debug handler is similar to the break handler and allows the programmer to inspect the continuation (control stack) of the exception to help determine the cause of the problem. If no REPL is running, as is the case for a script or top-level program run via the `--script` or `--program` command-line options, the default exception handler exits from the script or program after printing the message. To allow scripts and top-level programs to be debugged, the default exception handler can be forced via the `debug-on-exception` parameter or the `--debug-on-exception` command-line option to invoke `debug` directly.

Developing a large program entirely in the REPL is unmanageable, and we usually even want to store smaller programs in a file for future use. (The expeditor's history is saved across Scheme sessions, but there is a limit on the number of items, so it is not a good idea to count on a program remaining in the history indefinitely.) Thus, a Scheme programmer typically creates a file containing Scheme source code using a text editor, such as `vi`, and loads the file into *Chez Scheme* to test them. The conventional filename extension for *Chez Scheme* source files is "`.ss`," but the file can have any extension or even no extension at all. A source file can be loaded during an interactive session by typing `(load "`*path*`")`. Files to be loaded can also be named on the command line when the system is started. Any form that can be typed interactively can be placed in a file to be loaded.

Chez Scheme compiles source forms as it sees them to machine code before evaluating them, i.e., "just in time." In order to speed loading of a large file or group of files, each file can be compiled ahead of time via `compile-file`, which puts the compiled code into a separate object file. For example, `(compile-file "`*path*`")` compiles the forms in the file *path*.ss and places the resulting object code in the file *path*.so. Loading a pre-compiled file is essentially no different from loading the source file, except that loading is faster since compilation has already been done.

When compiling a file or set of files, it is often more convenient to use a shell command than to enter *Chez Scheme* interactively to perform the compilation. This is easily accomplished by "piping" in the command to compile the file as shown below.

```
echo '(compile-file "filename")' | scheme -q
```

The `-q` option suppresses the system's greeting messages for more compact output, which is especially useful when compiling numerous files. The single-quote marks surrounding the `compile-file` call should be left off for Windows shells.

When running in this "batch" mode, especially from within "make" files, it is often desirable to force the default exception handler to exit immediately to the shell with a nonzero exit status. This may be accomplished by setting the `reset-handler` to `abort`.

```
echo '(reset-handler abort) (compile-file "filename")' | scheme -q
```

One can also redefine the `base-exception-handler` (Section 12.1) to achieve a similar effect while exercising more control over the format of the messages that are produced.

2.2. Expression Editor

When Chez Scheme is used interactively in a shell window, as described above, or when `new-cafe` is invoked explicitly from a top-level program or script run via `--program` or `--script`, the waiter's "prompt and read" procedure employs an expression editor that permits entry and editing of single- and multiple-line expressions, automatically indents expressions as they are entered, supports identifier completion outside string constants based on the identifiers defined in the interactive environment, and supports filename completion within string constants. The expression editor also maintains a history of expressions typed during and across sessions and supports tcsh-like history movement and search commands. Other editing commands include simple cursor movement via arrow keys, deletion of characters via backspace and delete, and movement, deletion, and other commands using mostly emacs key bindings.

The expression editor does not run if the TERM environment variable is not set (on Unix-based systems), if the standard input or output files have been redirected, or if the `--eedisable` command-line option (Section 2.9) has been used. The history is saved across sessions, by default, in the file ".chezscheme_history" in the user's home directory. The `--eehistory` command-line option (Section 2.9) can be used to specify a different location for the history file or to disable the saving and restoring of the history file.

Keys for nearly all printing characters (letters, digits, and special characters) are "self inserting" by default. The open parenthesis, close parenthesis, open bracket, and close bracket keys are self inserting as well, but also cause the editor to "flash" to the matching delimiter, if any. Furthermore, when a close parenthesis or close bracket is typed, it is automatically corrected to match the corresponding open delimiter, if any.

Key bindings for other keys and key sequences initially recognized by the expression editor are given below, organized into groups by function. Some keys or key sequences serve more

than one purpose depending upon context. For example, tab is used for identifier completion, filename completion, and indentation. Such bindings are shown in each applicable functional group.

Multiple-key sequences are displayed with hyphens between the keys of the sequences, but these hyphens should not be entered. When two or more key sequences perform the same operation, the sequences are shown separated by commas.

Detailed descriptions of the editing commands are given in Chapter 14, which also describes parameters that allow control over the expression editor, mechanisms for adding or changing key bindings, and mechanisms for creating new commands.

Newlines, acceptance, exiting, and redisplay:

enter, `^M`	accept balanced entry if used at end of entry; else add a newline before the cursor and indent
`^J`	accept entry unconditionally
`^O`	insert newline after the cursor and indent
`^D`	exit from the waiter if entry is empty; else delete character under cursor
`^Z`	suspend to shell if shell supports job control
`^L`	redisplay entry
`^L-^L`	clear screen and redisplay entry

Basic movement and deletion:

leftarrow, `^B`	move cursor left
rightarrow, `^F`	move cursor right
uparrow, `^P`	move cursor up; from top of unmodified entry, move to preceding history entry.
downarrow, `^N`	move cursor down; from bottom of unmodified entry, move to next history entry
`^D`	delete character under cursor if entry not empty, else exit from the waiter
backspace, `^H`	delete character before cursor
delete	delete character under cursor

Line movement and deletion:

home, `^A`	move cursor to beginning of line
end, `^E`	move cursor to end of line
`^K`, esc-k	delete to end of line or, if cursor is at the end of a line, join with next line
`^U`	delete contents of current line

When used on the first line of a multiline entry of which only the first line is displayed, i.e., immediately after history movement, `^U` deletes the contents of the entire entry, like `^G` (described below).

Expression movement and deletion:

esc-^F	move cursor to next expression
esc-^B	move cursor to preceding expression
esc-]	move cursor to matching delimiter
^]	flash cursor to matching delimiter
esc-^K, esc-delete	delete next expression
esc-backspace, esc-^H	delete preceding expression

Entry movement and deletion:

esc-<	move cursor to beginning of entry
esc->	move cursor to end of entry
^G	delete current entry contents
^C	delete current entry contents; reset to end of history

Indentation:

tab	re-indent current line if identifier/filename prefix not just entered; else insert completion
esc-tab	re-indent current line unconditionally
esc-q, esc-Q, esc-^Q	re-indent each line of entry

Identifier/filename completion:

tab	insert completion if identifier/filename prefix just entered; else re-indent current line
tab-tab	show possible identifier/filename completions at end of identifier/filename just typed, else re-indent
^R	insert next identifier/filename completion

Identifier completion is performed outside of a string constant, and filename completion is performed within a string constant. (In determining whether the cursor is within a string constant, the expression editor looks only at the current line and so can be fooled by string constants that span multiple lines.) If at end of existing identifier or filename, i.e., not one just typed, the first tab re-indents, the second tab inserts identifier completion, and the third shows possible completions.

History movement:

uparrow, ^P	move to preceding entry if at top of unmodified entry; else move up within entry
downarrow, ^N	move to next entry if at bottom of unmodified entry; else move down within entry
esc-uparrow, esc-^P	move to preceding entry from unmodified entry
esc-downarrow, esc-^N	move to next entry from unmodified entry
esc-p	search backward through history for given prefix
esc-n	search forward through history for given prefix
esc-P	search backward through history for given string
esc-N	search forward through history for given string

To search, enter a prefix or string followed by one of the search key sequences. Follow with additional search key sequences to search further backward or forward in the history. For

example, enter "(define" followed by one or more esc-p key sequences to search backward for entries that are definitions, or "(define" followed by one or more esc-P key sequences for entries that contain definitions.

Word and page movement:

esc-`f`, esc-`F`	move cursor to end of next word
esc-`b`, esc-`B`	move cursor to start of preceding word
`^X-[`	move cursor up one screen page
`^X-]`	move cursor down one screen page

Inserting saved text:

`^Y`	insert most recently deleted text
`^V`	insert contents of window selection/paste buffer

Mark operations:

`^@`, ^space, `^^`	set mark to current cursor position
`^X-^X`	move cursor to mark, leave mark at old cursor position
`^W`	delete between current cursor position and mark

Command repetition:

esc-`^U`	repeat next command four times
esc-`^U`-*n*	repeat next command *n* times

2.3. The Interaction Environment

In the language of the Revised[6] Report, code is structured into libraries and "top-level programs." The Revised[6] Report does not require an implementation to support interactive use, and it does not specify how an interactive top level should operate, leaving such details up to the implementation.

In *Chez Scheme*, when one enters definitions or expressions at the prompt or loads them from a file, they operate on an interaction environment, which is a mutable environment that initially holds bindings only for built-in keywords and primitives. It may be augmented by user-defined identifier bindings via top-level definitions. The interaction environment is also referred to as the top-level environment, because it is at the top level for purposes of scoping. Programs entered at the prompt or loaded from a file via `load` should not be confused with RNRS top-level programs, which are actually more similar to libraries in their behavior. In particular, while the same identifier can be defined multiple times in the interaction environment, to support incremental program development, an identifier can be defined at most once in an RNRS top-level program.

The default interaction environment used for any code that occurs outside of an RNRS top-level program or library (including such code typed at a prompt or loaded from a file) contains all of the bindings of the `(chezscheme)` library (or `scheme` module, which exports the same set of bindings). This set contains a number of bindings that are not in the RNRS libraries. It also contains a number of bindings that extend the RNRS counterparts in some way and are thus not strictly compatible with the RNRS bindings for the same

identifiers. To replace these with bindings strictly compatible with RNRS, simply import the `rnrs` libraries into the interaction environment by typing the following into the REPL or loading it from a file:

```
(import
  (rnrs)
  (rnrs eval)
  (rnrs mutable-pairs)
  (rnrs mutable-strings)
  (rnrs r5rs))
```

To obtain an interaction environment that contains all *and only* RNRS bindings, use the following.

```
(interaction-environment
  (copy-environment
    (environment
      '(rnrs)
      '(rnrs eval)
      '(rnrs mutable-pairs)
      '(rnrs mutable-strings)
      '(rnrs r5rs))
    #t))
```

To be useful for most purposes, `library` and `import` should probably also be included, from the `(chezscheme)` library.

```
(interaction-environment
  (copy-environment
    (environment
      '(rnrs)
      '(rnrs eval)
      '(rnrs mutable-pairs)
      '(rnrs mutable-strings)
      '(rnrs r5rs)
      '(only (chezscheme) library import))
    #t))
```

It might also be useful to include `debug` in the set of identifiers imported from `(chezscheme)` to allow the debugger to be entered after an exception is raised.

Most of the identifiers bound in the default interaction environment that are not strictly compatible with the Revised[6] Report are variables bound to procedures with extended interfaces, i.e., optional arguments or extended argument domains. The others are keywords bound to transformers that extend the Revised[6] Report syntax in some way. This should not be a problem except for programs that count on exceptions being raised in cases that coincide with the extensions. For example, if a program passes the `=` procedure a single numeric argument and expects an exception to be raised, it will fail in the initial interaction environment because `=` returns `#t` when passed a single numeric argument.

Within the default interaction environment and those created as described above, variables

that name built-in procedures are read-only, i.e., cannot be assigned, since they resolve to the read-only bindings exported from the `(chezscheme)` library or some other library:

```
(set! cons +)  ⇒  exception: cons is immutable
```

Before assigning a variable bound to the name of a built-in procedure, the programmer must first define the variable. For example,

```
(define cons-count 0)
(define original-cons cons)
(define cons
  (lambda (x y)
    (set! cons-count (+ cons-count 1))
    (original-cons x y)))
```

redefines `cons` to count the number of times it is called, and

```
(set! cons original-cons)
```

assigns `cons` to its original value. Once a variable has been defined in the interaction environment using `define`, a subsequent definition of the same variable is equivalent to a `set!`, so

```
(define cons original-cons)
```

has the same effect as the `set!` above. The expression

```
(import (only (chezscheme) cons))
```

also binds `cons` to its original value. It also returns it to its original read-only state.

The simpler redefinition

```
(define cons (let () (import scheme) cons))
```

turns `cons` into a mutable variable with the same value as it originally had. Doing so, however, prevents the compiler from generating efficient code for calls to `cons` or producing warning messages when `cons` is passed the wrong number of arguments.

All identifiers not bound in the initial interaction environment and not defined by the programmer are treated as "potentially bound" as variables to facilitate the definition of mutually recursive procedures. For example, assuming that `yin` and `yang` have not been defined,

```
(define yin (lambda () (- (yang) 1)))
```

defines `yin` at top level as a variable bound to a procedure that calls the value of the top-level variable `yang`, even though `yang` has not yet been defined. If this is followed by

```
(define yang (lambda () (+ (yin) 1)))
```

the result is a mutually recursive pair of procedures that, when called, will loop indefinitely

or until the system runs out of space to hold the recursion stack. If `yang` must be defined as anything other than a variable, its definition should precede the definition of `yin`, since the compiler assumes `yang` is a variable in the absence of any indication to the contrary when `yang` has not yet been defined.

A subtle consequence of this useful quirk of the interaction environment is that the procedure `free-identifier=?` (Section 8.3 of *The Scheme Programming Language, 4th Edition*) does not consider unbound library identifiers to be equivalent to (as yet) undefined top-level identifiers, even if they have the same name, because the latter are actually assumed to be valid variable bindings.

```
(library (A) (export a)
  (import (rnrs))
  (define-syntax a
    (lambda (x)
      (syntax-case x ()
        [(_ id) (free-identifier=? #'id #'undefined)]))))
(let () (import (A)) (a undefined))  ⇒  #f
```

If it is necessary that they have the same binding, as in the case where an identifier is used as an auxiliary keyword in a syntactic abstraction exported from a library and used at top level, the library should define and export a binding for the identifier.

```
(library (A) (export a aux-a)
  (import (rnrs) (only (chezscheme) syntax-error))
  (define-syntax aux-a
    (lambda (x)
      (syntax-error x "invalid context")))
  (define-syntax a
    (lambda (x)
      (syntax-case x (aux-a)
        [(_ aux-a) #''okay]
        [(_ _) #''oops]))))
(let () (import (A)) (a aux-a))  ⇒  okay
(let () (import (only (A) a)) (a aux-a))  ⇒  oops
```

This issue does not arise when libraries are used entirely within other libraries or within RNRS top-level programs, since the interaction environment does not come into play.

2.4. Using Libraries and Top-Level Programs

An R6RS library can be defined directly in the REPL, loaded explicitly from a file (using `load` or `load-library`), or loaded implicitly from a file via `import`. When defined directly in the REPL or loaded explicitly from a file, a library form can be used to redefine an existing library, but `import` never reloads a library once it has been defined.

A library to be loaded implicitly via `import` must reside in a file whose name reflects the name of the library. For example, if the library's name is `(tools sorting)`, the base name of the file must be `sorting` with a valid extension, and the file must be in a directory

named `tools` which itself resides in one of the directories searched by `import`. The set of directories searched by `import` is determined by the `library-directories` parameter, and the set of extensions is determined by the `library-extensions` parameter.

The values of both parameters are lists of pairs of strings. The first string in each `library-directories` pair identifies a source-file base directory, and the second identifies the corresponding object-file base directory. Similarly, the first string in each `library-extensions` pair identifies a source-file extension, and the second identifies the corresponding object-file extension. The full path of a library source or object file consists of the source or object base followed by the components of the library name, separated by slashes, with the library extension added on the end. For example, for base `/usr/lib/scheme`, library name `(app lib1)`, and extension `.sls`, the full path is `/usr/lib/scheme/app/lib1.sls`. So, if `(library-directories)` contains the pathnames `"/usr/lib/scheme/libraries"` and `"."`, and `(library-extensions)` contains the extensions `.ss` and `.sls`, the path of the `(tools sorting)` library must be one of the following.

```
/usr/lib/scheme/libraries/tools/sorting.ss
/usr/lib/scheme/libraries/tools/sorting.sls
./tools/sorting.ss
./tools/sorting.sls
```

When searching for a library, `import` first constructs a partial name from the list of components in the library name, e.g., `a/b` for library `(a b)`. It then searches for the partial name in each pair of base directories, in order, trying each of the source extensions then each of the object extensions in turn before moving onto the next pair of base directories. If the partial name is an absolute pathname, e.g., `~/.myappinit` for a library named `(~/.myappinit)`, only the specified absolute path is searched, first with each source extension, then with each object extension. If the expander finds both a source file and its corresponding object file, and the object file is not older than the source file, the expander loads the object file. If the object file does not exist, if the object file is older, or if after loading the object file, the expander determines it was built using a library or include file that has changed, the source file is loaded or compiled, depending on the value of the parameter `compile-imported-libraries`. If `compile-imported-libraries` is set to `#t`, the expander compiles the library via the value of the `compile-library-handler` parameter, which by default calls `compile-library` (which is described below). Otherwise, the expander loads the source file. (Loading the source file actually causes the code to be compiled, assuming the default value of `current-eval`, but the compiled code is not saved to an object file.) An exception is raised during this process if a source or object file exists but is not readable or if an object file cannot be created.

The search process used by the expander when processing an `import` for a library that has not yet been loaded can be monitored by setting the parameter `import-notify` to `#t`. This parameter can be set from the command line via the `--import-notify` command-line option.

Whenever the expander determines it must compile a library to a file or load one from source, it adds the directory in which the file resides to the front of the `source-directories` list while compiling or loading the library. This allows a library to include files stored in or relative to its own directory.

When `import` compiles a library as described above, it does not also load the compiled library, because this would cause portions of library to be reevaluated. Because of this, run-time expressions in the file outside of a `library` form will not be evaluated. If such expressions are present and should be evaluated, the library should be compiled ahead of time or loaded explicitly.

A file containing a library may be compiled with `compile-file` or `compile-library`. The only difference between the two is that the latter treats the source file as if it were prefixed by an implicit `#!r6rs`, which disables *Chez Scheme* lexical extensions unless an explicit `#!chezscheme` marker appears in the file. Any libraries upon which the library depends must be compiled first. If one of the libraries imported by the library is subsequently recompiled (say because it was modified), the importing library must also be recompiled. Compilation and recompilation of imported libraries must be done explicitly by default but is done automatically when the parameter `compile-imported-libraries` is set to `#t` before compiling the importing library.

As with `compile-file`, `compile-library` can be used in "batch" mode via a shell command:

```
echo '(compile-library "filename")' | scheme -q
```

with single-quote marks surrounding the `compile-library` call omitted for Windows shells.

An RNRS top-level-program usually resides in a file, but one can also enter one directly into the REPL using the `top-level-program` forms, e.g.:

```
(top-level-program
  (import (rnrs))
  (display "What's up?\n"))
```

A top-level program stored in a file does not have the `top-level-program` wrapper, so the same top-level program in a file is just:

```
(import (rnrs))
(display "What's up?\n")
```

A top-level program stored in a file can be loaded from the file via the `load-program` procedure. A top-level program can also be loaded via `load`, but not without affecting the semantics. A program loaded via `load` is scoped at top level, where it can see all top-level bindings, whereas a top-level program loaded via `load-program` is self-contained, i.e., it can see only the bindings made visible by the leading `import` form. Also, the variable bindings in a program loaded via `load` also become top-level bindings, whereas they are local to the program when the program is loaded via `load-program`. Moreover, `load-program`, like `load-library`, treats the source file as if it were prefixed by an implicit `#!r6rs`, which disables *Chez Scheme* lexical extensions unless an explicit `#!chezscheme` marker appears in the file. A program loaded via `load` is also likely to be less efficient. Since the program's variables are not local to the program, the compiler must assume they could change at any time, which inhibits many of its optimizations.

Top-level programs may be compiled using `compile-program`, which is like `compile-file` but, as with `load-program`, properly implements the semantics and lexical restrictions of top-level programs. `compile-program` also copies the leading `#!` line, if any, from the

source file to the object file, resulting in an executable object file. Any libraries upon which the top-level program depends, other than built-in libraries, must be compiled first. The program must be recompiled if any of the libraries upon which it depends are recompiled. Compilation and recompilation of imported libraries must be done explicitly by default but is done automatically when the parameter `compile-imported-libraries` is set to `#t` before compiling the importing library.

As with `compile-file` and `compile-library`, `compile-program` can be used in "batch" mode via a shell command:

```
echo '(compile-program "filename")' | scheme -q
```

with single-quote marks surrounding the `compile-program` call omitted for Windows shells.

`compile-program` returns a list of libraries directly invoked by the compiled top-level program. When combined with the `library-requirements` and `library-object-filename` procedures, the list of libraries returned by `compile-program` can be used to determine the set of files that must be distributed with the compiled program file.

When run, a compiled program automatically loads the run-time code for each library upon which it depends, as if via `revisit`. If the program also imports one of the same libraries at run time, e.g., via the `environment` procedure, the system will attempt to load the compile-time information from the same file. The compile-time information can also be loaded explicitly from the same or a different file via `load` or `visit`.

2.5. Scheme Shell Scripts

When the `--script` command-line option is present, the named file is treated as a Scheme shell script, and the command-line is made available via the parameter `command-line`. This is primarily useful on Unix-based systems, where the script file itself may be made executable. To support executable shell scripts, the system ignores the first line of a loaded script if it begins with `#!` followed by a space or forward slash. For example, assuming that the *Chez Scheme* executable has been installed as /usr/bin/scheme, the following script prints its command-line arguments.

```
#! /usr/bin/scheme --script
(for-each
  (lambda (x) (display x) (newline))
  (cdr (command-line)))
```

The following script implements the traditional Unix `echo` command.

```
#! /usr/bin/scheme --script
(let ([args (cdr (command-line))])
  (unless (null? args)
    (let-values ([(newline? args)
                  (if (equal? (car args) "-n")
                      (values #f (cdr args))
                      (values #t args))])
```

```
(do ([args args (cdr args)] [sep "" " "])
    ((null? args))
  (printf "~a~a" sep (car args)))
(when newline? (newline)))))
```

Scripts may be compiled using `compile-script`, which is like `compile-file` but differs in two ways: (1) it copies the leading `#!` line from the source-file script into the object file, and (2) when the `#!` line is present, it disables the default compression of the resulting file, which would otherwise prevent it from being recognized as a script file.

If *Petite Chez Scheme* is installed, but not *Chez Scheme*, `/usr/bin/scheme` may be replaced with `/usr/bin/petite`.

The `--program` command-line option is like `--script` except that the script file is treated as an RNRS top-level program (Chapter 10). The following RNRS top-level program implements the traditional Unix `echo` command, as with the script above.

```
#! /usr/bin/scheme --program
(import (rnrs))
(let ([args (cdr (command-line))])
  (unless (null? args)
    (let-values ([(newline? args)
                  (if (equal? (car args) "-n")
                      (values #f (cdr args))
                      (values #t args))])
      (do ([args args (cdr args)] [sep "" " "])
          ((null? args))
        (display sep)
        (display (car args)))
      (when newline? (newline)))))
```

Again, if only *Petite Chez Scheme* is installed, `/usr/bin/scheme` may be replaced with `/usr/bin/petite`.

`scheme-script` may be used in place of `scheme --program` or `petite --program`, i.e.,

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

`scheme-script` runs *Chez Scheme*, if available, otherwise *Petite Chez Scheme*.

It is also possible to use `/usr/bin/env`, as recommended in the Revised[6] Report nonnormative appendices, which allows `scheme-script` to appear anywhere in the user's path.

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

If a top-level program depends on libraries other than those built into *Chez Scheme*, the `--libdirs` option can be used to specify which source and object directories to search. Similarly, if a library upon which a top-level program depends has an extension other than one of the standard extensions, the `--libexts` option can be used to specify additional extensions to search.

These options set the corresponding *Chez Scheme* parameters `library-directories` and `library-extensions`, which are described in Section 2.4. The format of the arguments

to `--libdirs` and `--libexts` is the same: a sequence of substrings separated by a single separator character. The separator character is a colon (:), except under Windows where it is a semi-colon (;). Between single separators, the source and object strings, if both are specified, are separated by two separator characters. If a single separator character appears at the end of the string, the specified pairs are added to the front of the existing list; otherwise, the specified pairs replace the existing list.

For example, where the separator is a colon,

```
scheme --libdirs "/home/moi/lib:"
```

adds the source/object directory pair

```
("/home/moi/lib" . "/home/moi/lib")
```

to the front of the default set of library directories, and

```
scheme --libdirs "/home/moi/libsrc::/home/moi/libobj:"
```

adds the source/object directory pair

```
("/home/moi/libsrc" . "/home/moi/libobj")
```

to the front of the default set of library directories. The parameters are set after all boot files have been loaded.

If no `--libdirs` option appears and the CHEZSCHEMELIBDIRS environment variable is set, the string value of CHEZSCHEMELIBDIRS is treated as if it were specified by a `--libdirs` option. Similarly, if no `--libexts` option appears and the CHEZSCHEMELIBEXTS environment variable is set, the string value of CHEZSCHEMELIBEXTS is treated as if it were specified by a `--libexts` option.

2.6. Optimization

To get the most out of the *Chez Scheme* compiler, it is necessary to give it a little bit of help. The most important assistance is to avoid the use of top-level (interaction-environment) bindings. Top-level bindings are convenient and appropriate during program development, since they simplify testing, redefinition, and tracing (Section 3.1) of individual procedures and syntactic forms. This convenience comes at a sizable price, however.

The compiler can propagate copies (of one variable to another or of a constant to a variable) and inline procedures bound to local, unassigned variables within a single top-level expression. For the procedures it does not inline, it can avoid constructing and passing unneeded closures, bypass argument-count checks, branch to the proper entry point in a case-lambda, and build rest arguments (more efficiently) on the caller side, where the length of the rest list is known at compile time. It can also discard the definitions of unreferenced variables, so there's no penalty for including a large library of routines, only a few of which are actually used.

It cannot do any of this with top-level variable bindings, since the top-level bindings can change at any time and new references to those bindings can be introduced at any time.

Fortunately, it is easy to restructure a program to avoid top-level bindings. This is naturally accomplished for portable code by placing the code into a single RNRS top-level program or by placing a portion of the code in a top-level program and the remainder in one or more separate libraries. Although not portable, one can also put all of the code into a single top-level `module` form or `let` expression, perhaps using `include` to bring in portions of the code from separate files. The compiler performs some optimization even across library boundaries, so the penalty for breaking a program up in this manner is generally acceptable. The compiler also supports whole-program optimization (via `compile-whole-program`), which can be used to eliminate all overhead for placing portions of a program into separate libraries.

Once an application's code has been placed into a single top-level program or into a top-level program and one or more libraries, the code can be loaded from source via `load-program` or compiled via `compile-program` and `compile-library`, as described in Section 2.4. Be sure not to use `compile-file` for the top-level program since this does not preserve the semantics nor result in code that is as efficient.

With an application structured as a single top-level program or as a top-level program and one or more libraries that do not interact frequently, we have done most of what can be done to help the compiler, but there are still a few more things we can do.

First, we can allow the compiler to generate "unsafe" code, i.e., allow the compiler to generate code in which the usual run-time type checks have been disabled. We do this by using the compiler's "optimize level 3" when compiling the program and library files. This can be accomplished by setting the parameter `optimize-level` to 3 while compiling the library or program, e.g.:

`(parameterize ([optimize-level 3]) (compile-program "`*filename*`"))`

or in batch mode via the `--optimize-level` command-line option:

`echo '(compile-program "`*filename*`")' | scheme -q --optimize-level 3`

It may also be useful to experiment with some of the other compiler control parameters and also with the storage manager's run-time operation. The compiler-control parameters, including `optimize-level`, are described in Section 12.6, and the storage manager control parameters are described in Section 13.1.

Finally, it is often useful to "profile" your code to determine that parts of the code that are executed most frequently. While this will not help the system optimize your code, it can help you identify "hot spots" where you need to concentrate your own hand-optimization efforts. In these hot spots, consider using more efficient operators, like fixnum or flonum operators in place of generic arithmetic operators, and using explicit loops rather than nested combinations of linear list-processing operators like `append`, `reverse`, and `map`. These operators can make code more readable when used judiciously, but they can slow down time-critical code.

Section 12.7 describes how to use the compiler's support for automatic profiling. Be sure

that profiling is not enabled when you compile your production code, since the code introduced into the generated code to perform the profiling adds significant run-time overhead.

2.7. Customization

Chez Scheme and *Petite Chez Scheme* are built from several subsystems: a "kernel" encapsulated in a static or shared library (dynamic link library) that contains operating-system interface and low-level storage management code, an executable that parses command-line arguments and calls into the kernel to initialize and run the system, a base boot file (petite.boot) that contains the bulk of the run-time library code, and an additional boot file (scheme.boot), for *Chez Scheme* only, that contains the compiler.

While the kernel and base boot file are essential to the operation of all programs, the executable may be replaced or even eliminated, and the compiler boot file need be loaded only if the compiler is actually used. In fact, the compiler is typically not loaded for distributed applications unless the application creates and executes code at run time.

The kernel exports a set of entry points that are used to initialize the Scheme system, load boot or heap files, run an interactive Scheme session, run script files, and deinitialize the system. In the threaded versions of the system, the kernel also exports entry points for activating, deactivating, and destroying threads. These entry points may be used to create your own executable image that has different (or no) command-line options or to run Scheme as a subordinate program within another program, i.e., for use as an extension language.

These entry points are described in Section 4.8, along with other entry points for accessing and modifying Scheme data structures and calling Scheme procedures.

The file main.c in the 'c' subdirectory contains the "main" routine for the distributed executable image; look at this file to gain an understanding of how the system startup entry points are used.

2.8. Building and Distributing Applications

Although useful as a stand-alone Scheme system, *Petite Chez Scheme* was conceived as a run-time system for compiled *Chez Scheme* applications. The remainder of this section describes how to create and distribute such applications using *Petite Chez Scheme*. It begins with a discussion of the characteristics of *Petite Chez Scheme* and how it compares with *Chez Scheme*, then describes how to prepare application source code, how to build and run applications, and how to distribute them.

Petite Chez Scheme Characteristics. Although interpreter-based, *Petite Chez Scheme* evaluates Scheme source code faster than might be expected. Some of the reasons for this are listed below.

- The run-time system is fully compiled, so library implementations of primitives ranging from `+` and `car` to `sort` and `printf` are just as efficient as in *Chez Scheme*, although they cannot be open-coded as in code compiled by *Chez Scheme*.
- The interpreter is itself a compiled Scheme application. Because it is written in Scheme, it directly benefits from various characteristics of Scheme that would have to be dealt with explicitly and with additional overhead in most other languages, including proper treatment of tail calls, first-class procedures, automatic storage management, and continuations.
- The interpreter employs a preprocessor that converts the code into a form that can be interpreted efficiently. In fact, the preprocessor shares its front end with the compiler, and this front end performs a variety of source-level optimizations.

Nevertheless, compiled code is still more efficient for most applications. The difference between the speed of interpreted and compiled code varies significantly from one application to another, but often amounts to a factor of five and sometimes to a factor of ten or more.

Several additional limitations result from the fact that *Petite Chez Scheme* does not include the compiler:

- The compiler must be present to process `foreign-procedure` and `foreign-callable` expressions, even when these forms are evaluated by the interpreter. These forms cannot be processed by the interpreter alone, so they cannot appear in source code to be processed by *Petite Chez Scheme*. Compiled versions of `foreign-procedure` and `foreign-callable` forms may, however, be included in compiled code loaded into *Petite Chez Scheme*.
- Inspector information is attached to code objects, which are generated only by the compiler, so source information and variable names are not available for interpreted procedures or continuations into interpreted procedures. This makes the inspector less effective for debugging interpreted code than it is for debugging compiled code.
- Procedure names are also attached to code objects, so while the compiler associates a name with each procedure when an appropriate name can be determined, the interpreter does not do so. This mostly impacts the quality of error messages, e.g., an error message might read "incorrect number of arguments to `#<procedure>`" rather than the likely more useful "incorrect number of arguments to `#<procedure` *name*`>`."
- The compiler detects, at compile time, some potential errors that the interpreter does not detect and reports them via compile-time warnings that identify the expression or the location in the source file, if any, where the expression appears.
- Automatic profiling cannot be enabled for interpreted code as it is for compiled code when `compile-profile` is set to `#t`.

Except as noted above, *Petite Chez Scheme* does not restrict what programs can do, and like *Chez Scheme*, it places essentially no limits on the size of programs or the memory images they create, beyond the inherent limitations of the underlying hardware or operating system.

Compiled scripts and programs.

One simple mechanism for distributing an application is to structure it as a script or RNRS top-level program, use `compile-script` or `compile-program`, as appropriate to compile it as described in Section 2.5, and distribute the resulting object file along with a complete distribution of *Petite Chez Scheme*. When this mechanism is used on Unix-based systems, if the source file begins with `#!` and the path that follows is the path to the *Chez Scheme* executable, e.g., `/usr/bin/scheme`, the one at the front of the object file should be replaced with the path to the *Petite Chez Scheme* executable, e.g., `/usr/bin/petite`. The path may have to be adjusted by the application's installation program based on where *Petite Chez Scheme* is installed on the target system. When used under Windows, the application's installation program should set up an appropriate shortcut that starts *Petite Chez Scheme* with the `--script` or `--program` option, as appropriate, followed by the path to the object file.

The remainder of this section describes how to distribute applications that do not require *Petite Chez Scheme* to be installed as a stand-alone system on the target machine.

Preparing Application Code. While it is possible to distribute applications in source-code form, i.e., as a set of Scheme source files to be loaded into *Petite Chez Scheme* by the end user, distributing compiled code has two major advantages over distributing source code. First, compiled code is usually much more efficient, as discussed in the preceding section, and second, compiled code is in binary form and thus provides more protection for proprietary application code.

Application source code generally consists of a set of Scheme source files possibly augmented by foreign code developed specifically for the application and packaged in shared libraries (also known as shared objects or, on Windows, dynamic link libraries). The following assumes that any shared-library source code has been converted into object form; how to do this varies by platform. (Some hints are given in Section 4.6.) The result is a set of one or more shared libraries that are loaded explicitly by the Scheme source code during program initialization.

Once the shared libraries have been created, the next step is to compile the Scheme source files into a set of Scheme object files. Doing so typically involves simply invoking `compile-file`, `compile-library`, or `compile-program`, as appropriate, on each source file to produce the corresponding object file. This may be done within a build script or "make" file via a command line such as the following:

```
echo '(compile-file "filename")' | scheme
```

which produces the object file `filename.so` from the source file `filename.ss`.

If the application code has been developed interactively or is usually loaded directly from source, it may be necessary to make some adjustments to a file to be compiled if the file contains expressions or definitions that affect the compilation of subsequent forms in the file. This can be accomplished via `eval-when` (Section 12.4). This is not typically necessary or desirable if the application consists of a set of RNRS libraries and programs.

You may also wish to disable generation of inspector information both to reduce the size of the compiled application code and to prevent others from having access to the expanded source code that is retained as part of the inspector information. To do so, set the parameter

`generate-inspector-information` to `#f` while compiling each file The downside of disabling inspector information is that the information will not be present if you need to debug your application, so it is usually desirable to disable inspector information only for production builds of your application. An alternative is to compile the code with inspector information enabled and strip out the debugging information later with `strip-fasl-file`.

The Scheme startup procedure determines what the system does when it is started. The default startup procedure loads the files listed on the command line (via `load`) and starts up a new café, like this.

```
(lambda fns (for-each load fns) (new-cafe))
```

The startup procedure may be changed via the parameter `scheme-start`. The following example demonstrates the installation of a variant of the default startup procedure that prints the name of each file before loading it.

```
(scheme-start
  (lambda fns
    (for-each
      (lambda (fn)
        (printf "loading ~a ..." fn)
        (load fn)
        (printf "~%"))
      fns)
    (new-cafe)))
```

A typical application startup procedure would first invoke the application's initialization procedure(s) and then start the application itself:

```
(scheme-start
  (lambda fns
    (initialize-application)
    (start-application fns)))
```

Any shared libraries that must be present during the running of an application must be loaded during initialization. In addition, all foreign procedure expressions must be executed after the shared libraries are loaded so that the addresses of foreign routines are available to be recorded with the resulting foreign procedures. The following demonstrates one way in which initialization might be accomplished for an application that links to a foreign procedure `show_state` in the Windows shared library `state.dll`:

```
(define show-state)

(define app-init
  (lambda ()
    (load-shared-object "state.dll")
    (set! show-state
      (foreign-procedure "show_state" (integer-32)
        integer-32))))

(scheme-start
```

```
(lambda fns
  (app-init)
  (app-run fns)))
```

Building and Running the Application. Building and running an application is straightforward once all shared libraries have been built and Scheme source files have been compiled to object code.

Although not strictly necessary, we suggest that you concatenate your object files, if you have more than one, into a single object file. This may be done on Unix systems simply via the `cat` program or on Windows via `copy`. Placing all of the object code into a single file simplifies both building and distribution of applications.

For top-level programs with separate libraries, `compile-whole-program` can be used to produce a single, fully optimized object file. Otherwise, when concatenating object files, put each library after the libraries it depends upon, with the program last.

With the Scheme object code contained within a single composite object file, it is possible to run the application simply by loading the composite object file into *Petite Chez Scheme*, e.g.:

```
petite app.so
```

where `app.so` is the name of the composite object file, and invoking the startup procedure to restart the system:

```
> ((scheme-start))
```

The point of setting `scheme-start`, however, is to allow the set of object files to be converted into a *boot file*. Boot files are loaded during the process of building the initial heap. Because of this, boot files have the following advantages over ordinary object files.

- Any code and data structures contained in the boot file or created while it is loaded is automatically compacted along with the base run-time library code and made static. Static code and data are never collected by the storage manager, so garbage collection overhead is reduced. (It is also possible to make code and data static explicitly at any time via the `collect` procedure.)
- The system looks for boot files automatically in a set of standard directories based on the name of the executable image, so you can install a copy of the *Petite Chez Scheme* executable image under your application's name and spare your users from supplying any command-line arguments or running a separate script to load the application code.

A boot file is simply an object file, possibly containing the code for more than one source file, prefixed by a boot header. The boot header identifies a base boot file upon which the application directly depends, or possibly two or more alternatives upon which the application can be run. In most cases, petite.boot will be identified as the base boot file, but in a layered application it may be another boot file of your creation that in turn depends upon petite.boot. The base boot file, and its base boot file, if any, are loaded automatically when your application boot file is loaded.

Boot files are created with `make-boot-file`. This procedure accepts two or more arguments. The first is a string naming the file into which the boot header and object code should be placed, the second is a list of strings naming base boot files, and the remainder are strings naming input files. For example, the call:

```
(make-boot-file "app.boot" '("petite") "app1.so" "app2.ss" "app3.so")
```

creates the boot file app.boot that identifies a dependency upon petite.boot and contains the object code for app1.so, the object code resulting from compiling app2.ss, and the object code for app3.so. The call:

```
(make-boot-file "app.boot" '("scheme" "petite") "app.so")
```

creates a header file that identifies a dependency upon either scheme.boot or petite.boot, with the object code from app.so. In the former case, the system will automatically load petite.boot when the application boot file is loaded, and in the latter it will load scheme.boot if it can find it, otherwise petite.boot. This would allow your application to run on top of the full *Chez Scheme* if present, otherwise *Petite Chez Scheme*.

In most cases, you can construct your application so it does not depend upon features of *Chez Scheme* (specifically, the compiler) by specifying only `"petite"` in the call to `make-boot-file`. If your application calls `eval`, however, and you wish to allow users to be able to take advantage of the faster execution speed of compiled code, then specifying both `"scheme"` and `"petite"` is appropriate.

Distributing the Application. Distributing an application involves can be as simple as creating a distribution package that includes the following items:

- the *Petite Chez Scheme* distribution,
- the application boot file,
- any application-specific shared libraries,
- an application installation script.

The application installation script should install *Petite Chez Scheme* if not already installed on the target system. It should install the application boot file in the same directory as the *Petite Chez Scheme* boot file petite.boot is installed, and it should install the application shared libraries, if any, either in the same location or in a standard location for shared libraries on the target system. It should also create a link to or copy of the *Petite Chez Scheme* executable under the name of your application, i.e., the name given to your application boot file. Where appropriate, it should also install desktop and start-menu shortcuts to run the executable.

2.9. Command-Line Options

Chez Scheme recognizes the following command-line options.

`-q`, `--quiet`	suppress greeting and prompt
`--script` *path*	run as shell script
`--program` *path*	run rnrs top-level program as shell script
`--libdirs` *dir*`:...`	set library directories
`--libexts` *ext*`:...`	set library extensions
`--compile-imported-libraries`	compile libraries before loading
`--import-notify`	enable import search messages
`--optimize-level 0 | 1 | 2 | 3`	set initial optimize level
`--debug-on-exception`	on uncaught exception, call `debug`
`--eedisable`	disable expression editor
`--eehistory off |` *path*	expression-editor history file
`--enable-object-counts`	have collector maintain object counts
`--retain-static-relocation`	keep reloc info for compute-size, etc.
`-b` *path*, `--boot` *path*	load boot file
`--verbose`	trace boot-file search process
`--version`	print version and exit
`--help`	print help and exit
`--`	pass through remaining args

The following options are recognized but cause the system to print an error message and exit because saved heaps are no longer supported.

`-h` *path*, `--heap` *path*	load heap file
`-s[`*n*`]` *path*, `--saveheap[`*n*`]` *path*	save heap file
`-c`, `--compact`	toggle compaction flag

With the default `scheme-start` procedure (Section 2.8), any remaining command-line arguments are treated as the names of files to be loaded before *Chez Scheme* begins interacting with the user, unless the `--script` or `--program` is present, in which case the remaining arguments are made available to the script via the `command-line` parameter (Section 2.1).

Most of the options are described elsewhere in this chapter, and a few are self-explanatory. The remainder pertain to the loading of boot files at system start-up time and are described below.

When *Chez Scheme* is run, it looks for one or more boot files to load. Boot files contain the compiled Scheme code that implements most of the Scheme system, including the interpreter, compiler, and most libraries. Boot files may be specified explicitly on the command line via `-b` options or implicitly. In the simplest case, no `-b` options are given and the necessary boot files are loaded automatically based on the name of the executable.

For example, if the executable name is "frob", the system looks for "frob.boot" in a set of standard directories. It also looks for and loads any subordinate boot files required by "frob.boot".

Subordinate boot files are also loaded automatically for the first boot file explicitly specified via the command line. Each boot file must be listed before those that depend upon it.

The `--verbose` option may be used to trace the file searching process and must appear before any boot arguments for which search tracing is desired.

Ordinarily, the search for boot files is limited to a set of installation directories, but this

may be overridden by setting the environment variable `SCHEMEHEAPDIRS`. `SCHEMEHEAPDIRS` should be a colon-separated list of directories, listed in the order in which they should be searched. Within each directory, the two-character escape sequence "`%v`" is replaced by the current version, and the two-character escape sequence "`%m`" is replaced by the machine type. A percent followed by any other character is replaced by the second character; in particular, "`%%`" is replaced by "`%`", and "`%:`" is replaced by "`:`". If `SCHEMEHEAPDIRS` ends in a non-escaped colon, the default directories are searched after those in `SCHEMEHEAPDIRS`; otherwise, only those listed in `SCHEMEHEAPDIRS` are searched.

Under Windows, semi-colons are used in place of colons, and one additional escape is recognized: "`%x`," which is replaced by the directory in which the executable file resides. The default search path under Windows consists of "`%x`" and "`%x\..\..\boot\%m`." The registry key `HeapSearchPath` in `HKLM\SOFTWARE\Chez Scheme\csv`*version*, where *version* is the *Chez Scheme* version number, e.g., `7.9.4`, can be set to override the default search path, and the `SCHEMEHEAPDIRS` environment variable overrides both the default and the registry setting, if any.

Boot files consist of ordinary compiled code and consist of a boot header and the compiled code for one or more source files. See Section 2.8 for instructions on how to create boot files.

3. Debugging

Chez Scheme has several features that support debugging. In addition to providing error messages when fully type-checked code is run, *Chez Scheme* also permits tracing of procedure calls, interruption of any computation, redefinition of exception and interrupt handlers, and inspection of any object, including the continuations of exceptions and interrupts.

Programmers new to Scheme or *Chez Scheme*, and even more experienced Scheme programmers, might want to consult the tutorial "How to Debug Chez Scheme Programs." HTML and PDF versions are available at http://www.cs.indiana.edu/chezscheme/debug/.

3.1. Tracing

Tracing is one of the most useful mechanisms for debugging Scheme programs. *Chez Scheme* permits any primitive or user-defined procedure to be traced. The trace package prints the arguments and return values for each traced procedure with a compact indentation mechanism that shows the nesting depth of calls. The distinction between tail calls and nontail calls is reflected properly by an increase in indentation for nontail calls only. For nesting depths of 10 or greater, a number in brackets is used in place of indentation to signify nesting depth.

This section covers the mechanisms for tracing procedures and controlling trace output.

`(trace-lambda` *name formals* $body_1$ $body_2$ `...)` **syntax**
returns: a traced procedure
libraries: `(chezscheme)`

A `trace-lambda` expression is equivalent to a `lambda` expression with the same formals and body except that trace information is printed to the trace output port whenever the procedure is invoked, using *name* to identify the procedure. The trace information shows the value of the arguments passed to the procedure and the values returned by the procedure, with indentation to show the nesting of calls.

The traced procedure `half` defined below returns the integer quotient of its argument and 2.

```
(define half
  (trace-lambda half (x)
    (cond
      [(zero? x) 0]
      [(odd? x) (half (- x 1))]
      [(even? x) (+ (half (- x 1)) 1)])))
```

A trace of the call `(half 5)`, which returns 2, is shown below.

```
|(half 5)
|(half 4)
| (half 3)
| (half 2)
| |(half 1)
| |(half 0)
| |0
| 1
|2
```

This example highlights the proper treatment of tail and nontail calls by the trace package. Since `half` tail calls itself when its argument is odd, the call `(half 4)` appears at the same level of indentation as the call `(half 5)`. Furthermore, since the return values of `(half 5)` and `(half 4)` are necessarily the same, only one return value is shown for both calls.

`(trace-case-lambda` *name clause* `...)` **syntax**
returns: a traced procedure
libraries: `(chezscheme)`

A `trace-case-lambda` expression is equivalent to a `case-lambda` expression with the same clauses except that trace information is printed to the trace output port whenever the procedure is invoked, using *name* to identify the procedure. The trace information shows the value of the arguments passed to the procedure and the values returned by the procedure, with indentation to show the nesting of calls.

`(trace-let` *name* `((`*var expr*`) ...)` $body_1$ $body_2$ `...)` **syntax**
returns: the values of the body $body_1$ $body_2$ `...`
libraries: `(chezscheme)`

A `trace-let` expression is equivalent to a named `let` expression with the same name, bindings, and body except that trace information is printed to the trace output port on entry or reentry (via invocation of the procedure bound to `name`) into the `trace-let` expression.

A `trace-let` expression of the form

```
(trace-let name ([var expr] ...)
  body1 body2 ...)
```

can be rewritten in terms of `trace-lambda` as follows:

```
((letrec ([name
           (trace-lambda name (var ...)
             body1 body2 ...)])
   name)
 expr ...)
```

`trace-let` may be used to trace ordinary `let` expressions as well as `let` expressions as long as the name inserted along with the `trace-let` keyword in place of `let` does not appear free within the body of the `let` expression. It is also sometimes useful to insert a `trace-let` expression into a program simply to display the value of an arbitrary expression at the current trace indentation. For example, a call to the following variant of `half`

```
(define half
  (trace-lambda half (x)
    (cond
      [(zero? x) 0]
      [(odd? x) (half (trace-let decr-value () (- x 1)))]
      [(even? x) (+ (half (- x 1)) 1)])))
```

with argument 5 results in the trace:

```
|(half 5)
| (decr-value)
| 4
|(half 4)
| (half 3)
| |(decr-value)
| |2
| (half 2)
| |(half 1)
| | (decr-value)
| | 0
| |(half 0)
| 1
|2
```

`(trace-do ((`*var init update*`) ...) (`*test result ...*`) `*expr ...*`)` **syntax**
returns: the values of the last *result* expression
libraries: `(chezscheme)`

A `trace-do` expression is equivalent to a `do` expression with the same subforms, except that trace information is printed to the trace output port, showing the values of *var* `...` and each iteration and the final value of the loop on termination. For example, the expression

```
(trace-do ([old '(a b c) (cdr old)]
           [new '() (cons (car old) new)])
  ((null? old) new))
```

produces the trace

```
|(do (a b c) ())
|(do (b c) (a))
|(do (c) (b a))
|(do () (c b a))
|(c b a)
```

and returns `(c b a)`.

`(trace` *var*$_1$ *var*$_2$ `...)` **syntax**
returns: a list of *var*$_1$ *var*$_2$ ...
`(trace)` **syntax**
returns: a list of all currently traced top-level variables
libraries: `(chezscheme)`

In the first form, `trace` reassigns the top-level values of *var*$_1$ *var*$_2$ ..., whose values must be procedures, to equivalent procedures that display trace information in the manner of `trace-lambda`.

`trace` works by encapsulating the old value of each var in a traced procedure. It could be defined approximately as follows. (The actual version records and returns information about traced variables.)

```
(define-syntax trace
  (syntax-rules ()
    [(_ var ...)
     (begin
       (set-top-level-value! 'var
         (let ([p (top-level-value 'var)])
           (trace-lambda var args (apply p args))))
       ...)]))
```

Tracing for a procedure traced in this manner may be disabled via `untrace` (see below), an assignment of the corresponding variable to a different, untraced value, or a subsequent use of `trace` for the same variable. Because the value is traced and not the binding, however, a traced value obtained before tracing is disabled and retained after tracing is disabled will remain traced.

`trace` without subexpressions evaluates to a list of all currently traced variables. A variable is currently traced if it has been traced and not subsequently untraced or assigned to a different value.

The following transcript demonstrates the use of `trace` in an interactive session.

```
> (define half
    (lambda (x)
      (cond
        [(zero? x) 0]
        [(odd? x) (half (- x 1))]
        [(even? x) (+ (half (- x 1)) 1)])))
> (half 5)
2
```

```
> (trace half)
(half)
> (half 5)
|(half 5)
|(half 4)
| (half 3)
| (half 2)
| |(half 1)
| |(half 0)
| |0
| 1
|2
2
> (define traced-half half)
> (untrace half)
(half)
> (half 2)
1
> (traced-half 2)
|(half 2)
|1
1
```

(untrace *var*$_1$ *var*$_2$ ...**)** **syntax**
(untrace) **syntax**
returns: a list of untraced variables
libraries: `(chezscheme)`

`untrace` restores the original (pre-`trace`) top-level values of each currently traced variable in *var*$_1$ *var*$_2$ ..., effectively disabling the tracing of the values of these variables. Any variable in *var*$_1$ *var*$_2$ ... that is not currently traced is ignored. If `untrace` is called without arguments, the values of all currently traced variables are restored.

The following transcript demonstrates the use of `trace` and `untrace` in an interactive session to debug an incorrect procedure definition.

```
> (define square-minus-one
    (lambda (x)
      (- (* x x) 2)))
> (square-minus-one 3)
7
> (trace square-minus-one * -)
(square-minus-one * -)
> (square-minus-one 3)
|(square-minus-one 3)
| (* 3 3)
| 9
|(- 9 2)
|7
```

```
7
> (define square-minus-one
    (lambda (x)
      (- (* x x) 1))) ; change the 2 to 1
> (trace)
(- *)
> (square-minus-one 3)
|(* 3 3)
|9
|(- 9 1)
|8
8
> (untrace square-minus-one)
()
> (untrace * -)
(- *)
> (square-minus-one 3)
8
```

The first call to `square-minus-one` indicates there is an error, the second (traced) call indicates the step at which the error occurs, the third call demonstrates that the fix works, and the fourth call demonstrates that `untrace` does not wipe out the fix.

`trace-output-port` **thread parameter**
libraries: `(chezscheme)`

`trace-output-port` is a parameter that determines the output port to which tracing information is sent. When called with no arguments, `trace-output-port` returns the current trace output port. When called with one argument, which must be a textual output port, `trace-output-port` changes the value of the current trace output port.

`trace-print` **thread parameter**
libraries: `(chezscheme)`

The value of `trace-print` must be a procedure of two arguments, an object and an output port. The trace package uses the value of `trace-print` to print the arguments and return values for each call to a traced procedure. `trace-print` is set to `pretty-print` by default.

The trace package sets `pretty-initial-indent` to an appropriate value for the current nesting level before calling the value of `trace-print` so that multiline output can be indented properly.

`(trace-define` *var expr*`)` **syntax**
`(trace-define (`*var . idspec*`)` $body_1$ $body_2$ `...)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

`trace-define` is a convenient shorthand for defining variables bound to traced procedures

of the same name. The first form is equivalent to

```
(define var
  (let ([x expr])
    (trace-lambda var args
      (apply x args))))
```

and the second is equivalent to

```
(define var
  (trace-lambda var idspec
    body1 body2 ...))
```

In the former case, *expr* must evaluate to a procedure.

```
> (let ()
    (trace-define plus
      (lambda (x y)
        (+ x y)))
    (list (plus 3 4) (+ 5 6)))
|(plus 3 4)
|7
(7 11)
```

(trace-define-syntax *keyword expr***)** **syntax**
returns: unspecified
libraries: `(chezscheme)`

`trace-define-syntax` traces the input and output to the transformer value of *expr*, stripped of the contextual information used by the expander to maintain lexical scoping.

```
> (trace-define-syntax let*
    (syntax-rules ()
      [(_ () b1 b2 ...)
       (let () b1 b2 ...)]
      [(_ ((x e) m ...) b1 b2 ...)
       (let ((x e))
         (let* (m ...) b1 b2 ...))]))
> (let* ([x 3] [y (+ x x)]) (list x y))
|(let* (let* [(x 3) (y (+ x x))] [list x y]))
|(let ([x 3]) (let* ([y (+ x x)]) (list x y)))
|(let* (let* [(y (+ x x))] [list x y]))
|(let ([y (+ x x)]) (let* () (list x y)))
|(let* (let* () [list x y]))
|(let () (list x y))
(3 6)
```

Without contextual information, the displayed forms are more readable but less precise, since different identifiers with the same name are indistinguishable, as shown in the example below.

```
> (let ([x 0])
    (trace-define-syntax a
      (syntax-rules ()
        [(_ y) (eq? x y)]))
    (let ([x 1])
      (a x)))
|(a (a x))
|(eq? x x)
#f
```

3.2. The Interactive Debugger

The interactive debugger is entered as a result of a call to the procedure `debug` after an exception is handled by the default exception handler. It can also be entered directly from the default exception handler, for serious or non-warning conditions, if the parameter `debug-on-exception` is true.

Within the debugger, the command "?" lists the debugger command options. These include commands to:

- inspect the raise continuation,
- display the condition,
- inspect the condition, and
- exit the debugger.

The raise continuation is the continuation encapsulated within the condition, if any. The standard exception reporting procedures and forms `assert`, `assertion-violation`, and `error` as well as the *Chez Scheme* procedures `assertion-violationf`, `errorf`, and `syntax-error` all raise exceptions with conditions that encapsulate the continuations of their calls, allowing the programmer to inspect the frames of pending calls at the point of a violation, error, or failed assertion.

A variant of the interactive debugger, the break handler, is entered as the result of a keyboard interrupt handled by the default keyboard-interrupt handler or an explicit call to the procedure `break` handled by the default break handler. Again, the command "?" lists the command options. These include commands to:

- exit the break handler and continue,
- reset to the current café,
- abort the entire Scheme session,
- enter a new café,
- inspect the current continuation, and
- display program statistics (run time and memory usage).

It is also usually possible to exit from the debugger or break handler by typing the end-of-file character ("control-D" under Unix, "control-Z" under Windows).

`(debug)` **procedure**
returns: does not return
libraries: `(chezscheme)`

When the default exception handler receives a serious or non-warning condition, it displays the condition and resets to the current café. Before it resets, it saves the condition in the parameter `debug-condition`. The `debug` procedure may be used to inspect the condition. Whenever one of the built-in error-reporting mechanisms is used to raise an exception, the continuation at the point where the exception was raised can be inspected as well. More generally, `debug` allows the continuation contained within any continuation condition created by `make-continuation-condition` to be inspected.

If the parameter `debug-on-exception` is set to `#t`, the default exception handler enters the debugger directly for all serious and non-warning conditions, delaying its reset until after the debugger exits. The `--debug-on-exception` command-line option may be used to set `debug-on-exception` to `#t` from the command line, which is particularly useful when debugging scripts or top-level programs run via the `--script` or `--program` command-line options.

3.3. The Interactive Inspector

The inspector may be called directly via the procedure `inspect` or indirectly from the debugger. It allows the programmer to examine circular objects, objects such as ports and procedures that do not have a reader syntax, and objects such as continuations and variables that are not directly accessible by the programmer, as well as ordinary printable Scheme objects.

The primary intent of the inspector is examination, not alteration, of objects. The values of assignable variables may be changed from within the inspector, however. Assignable variables are generally limited to those for which assignments occur in the source program. It is also possible to invoke arbitrary procedures (including mutation procedures such as `set-car!`) on an object. No mechanism is provided for altering objects that are inherently immutable, e.g., nonassignable variables, procedures, and bignums, since doing so can violate assumptions made by the compiler and run-time system.

The user is presented with a prompt line that includes a printed representation of the current object, abbreviated if necessary to fit on the line. Various commands are provided for displaying objects and moving around inside of objects. On-line descriptions of the command options are provided. The command "?" displays commands that apply specifically to the current object. The command "??" displays commands that are always applicable. The command "h" provides a brief description of how to use the inspector. The end-of-file character or the command "q" exits the inspector.

`(inspect` *obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

Invokes the inspector on *obj*, as described above. The commands recognized by the inspector are listed below, categorized by the type of the current object.

Generally applicable commands

`help` or `h` displays a brief description of how to use the inspector.

`?` displays commands applicable to the current type of object.

`??` displays the generally applicable commands.

`print` or `p` prints the current object (using `pretty-print`).

`write` or `w` writes the current object (using `write`).

`size` writes the size in bytes occupied by the current object (determined via `compute-size`), including any objects accessible from the current object except those for which the size was previously requested during the same interactive inspector session.

`find` *expr* [*g*] evaluates *expr*, which should evaluate to a procedure of one argument, and searches (via `make-object-finder`) for the first occurrence of an object within the current object for which the predicate returns a true value, treating immediate values (e.g., fixnums), values in generations older than *g*, and values already visited during the search as leaves. If *g* is not unspecified, it defaults to the current maximum generation, i.e., the value of `collect-maximum-generation`. If specified, *g* must be an exact nonnegative integer less than or equal to the current maximum generation or the symbol `static` representing the static generation. If such an object is found, the inspector's focus moves to that object as if through a series of steps that lead from the current object to the located object, so that the `up` command can be used to determine where the object was found relative to the original object.

`find-next` repeats the last `find`, locating an occurrence not previously found, if any.

`up` or `u` *n* returns to the *nth* previous level. Used to move outwards in the structure of the inspected object. *n* defaults to 1.

`top` or `t` returns to the outermost level of the inspected object.

`forward` or `f` moves to the *nth* next expression. Used to move from one element to another of an object containing a sequence of elements, such as a list, vector, record, frame, or closure. *n* defaults to 1.

`back` or `b` moves to the *nth* previous expression. Used to move from one element to another of an object containing a sequence of elements, such as a list, vector, record, frame, or closure. *n* defaults to 1.

`=>` *expr* sends the current object to the procedure value of *expr*. *expr* may begin on the current or following line and may span multiple lines.

`file` *path* opens the source file at the specified path for listing. The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files.

`list` *line count* lists *count* lines of the current source file (see `file`) starting at *line*. *line* defaults to the end of the previous set of lines listed and *count* defaults to ten or the number of lines previously listed. If *line* is negative, listing begins *line* lines before the previous set of lines listed.

`files` shows the currently open source files.

`mark` or `m` m marks the current location with the symbolic mark m. If m is not specified, the current location is marked with a unique default mark.

`goto` or `g` m returns to the location marked m. If m is not specified, the inspector returns to the location marked with the default mark.

`new-cafe` or `n` enters a new read-eval-print loop (café), giving access to the normal top-level environment.

`quit` or `q` exits from the inspector.

`reset` or `r` resets to the current café.

`abort` or `a` x aborts from Scheme with exit status x, which defaults to -1.

Continuation commands

`show-frames` or `sf` shows the next n frames. If n is not specified, all frames are displayed.

`depth` displays the number of frames in the continuation.

`down` or `d` n move to the *nth* frame down in the continuation. n defaults to 1.

`show` or `s` shows the continuation (next frame) and, if available, the calling procedure source, the pending call source, the closure, and the frame and free-variable values. Source is available only if generation of inspector information was enabled during compilation of the corresponding lambda expression.

`show-local` or `sl` is like `show` or `s` except that free variable values are not shown. (If present, free variable values can be found by inspecting the closure.)

`length` or `l` displays the number of elements in the topmost frame of the continuation.

`ref` or `r` moves to the *nth* or named frame element. *n* defaults to 0. If multiple elements have the same name, only one is accessible by name, and the others must be accessed by number.

`code` or `c` moves to the source for the calling procedure.

`call` moves to the source for the pending call.

`file` opens the source file containing the pending call, if known. The parameter `source-directories` (Section 12.5) determines the list of source directories searched for source files identified by relative path names.

For absolute pathnames starting with a `/` (or `\` or a directory specifier under Windows), the inspector tries the absolute pathname first, then looks for the last (filename) component of the path in the list of source directories. For pathnames starting with `./` (or `.\` under Windows) or `../` (or `..\` under Windows), the inspector looks in `"."` or `".."` first, as appropriate, then for the entire `.-` or `..`-prefixed pathname in the source directories, then for the last (filename) component in the source directories. For other (relative) pathnames, the inspector looks for the entire relative pathname in the list of source directories, then the last (filename) component in the list of source directories.

If a file by the same name as but different contents from the original source file is found during this process, it will be skipped over. This typically happens because the file has been modified since it was compiled. Pass an explicit filename argument to force opening of a particular file (see the generally applicable commands above).

`eval` or `e` *expr* evaluates the expression *expr* in an environment containing bindings for the elements of the frame. Within the evaluated expression, the value of each frame element *n* is accessible via the variable `%`*n*. Named elements are accessible via their names as well. Names are available only if generation of inspector information was enabled during compilation of the corresponding lambda expression.

`set!` or `!` *n e* sets the value of the *nth* frame element to *e*, if the frame element corresponds to an assignable variable. *n* defaults to 0.

Procedure commands

`show` or `s` shows the source and free variables of the procedure. Source is available only if generation of inspector information was enabled during compilation of the corresponding lambda expression.

`code` or `c` moves to the source for the procedure.

`file` opens the file containing the procedure's source code, if known. See the description of the continuation `file` entry above for more information.

`length` or `l` displays the number of free variables whose values are recorded in the procedure object.

`ref` or `r` moves to the *nth* or named free variable. *n* defaults to 0. If multiple free variables have the same name, only one is accessible by name, and the others must be accessed by number.

`set!` or `!` *n e* sets the value of the *nth* free variable to *e*, if the variable is assignable. *n* defaults to 0.

`eval` or `e` *expr* evaluates the expression *expr* in an environment containing bindings for the free variables of the procedure. Within the evaluated expression, the value of each free variable *n* is accessible via the variable `%`*n*. Named free variables are accessible via their names as well. Names are available only if generation of inspector information was enabled during compilation of the corresponding lambda expression.

Pair (list) commands

`show` or `s` *n* shows the first *n* elements of the list. If *n* is not specified, all elements are displayed.

`length` or `l` displays the list length.

`car` moves to the object in the car of the current object.

`cdr` moves to the object in the cdr.

`ref` or `r` *n* moves to the *nth* element of the list. *n* defaults to 0.

`tail` *n* moves to the *nth* cdr of the list. *n* defaults to 1.

Vector, Bytevector, and Fxvector commands

`show` or `s` *n* shows the first *n* elements of the vector. If *n* is not specified, all elements are displayed.

`length` or `l` displays the vector length.

`ref` or `r` *n* moves to the *nth* element of the vector. *n* defaults to 0.

String commands

`show` or `s` *n* shows the first *n* elements of the string. If *n* is not specified, all elements are displayed.

`length` or `l` displays the string length.

`ref` or `r` *n* moves to the *nth* element of the string. *n* defaults to 0.

`unicode` *n* displays the first *n* elements of the string as hexadecimal Unicode scalar values.

`ascii` *n* displays the first *n* elements of the string as hexadecimal ASCII values, using `--` to denote characters whose Unicode scalar values are not in the ASCII range.

Symbol commands

`show` or `s` shows the fields of the symbol.

`value` or `v` moves to the top-level value of the symbol.

`name` or `n` moves to the name of the symbol.

`property-list` or `pl` moves to the property list of the symbol.

`ref` or `r` *n* moves to the *nth* field of the symbol. Field 0 is the top-level value of the symbol, field 1 is the symbol's name, and field 2 is its property list. *n* defaults to 0.

Character commands

`unicode` displays the hexadecimal Unicode scalar value for the character.

`ascii` displays the hexadecimal ASCII code for the character, using `--` to denote characters whose Unicode scalar values are not in the ASCII range.

Box commands

`show` or `s` shows the contents of the box.

`unbox` or `ref` or `r` moves to the boxed object.

Port commands

`show` or `s` shows the fields of the port, including the input and output size, index, and buffer fields.

`name` moves to the port's name.

`handler` moves to the port's handler.

`output-buffer` or `ob` moves to the port's output buffer.

`input-buffer` or `ib` moves to the port's input buffer.

Record commands

`show` or `s` shows the contents of the record.

`fields` moves to the list of field names of the record.

`name` moves to the name of the record.

`rtd` moves to the record-type descriptor of the record.

`ref` or `r` *name* moves to the named field of the record, if accessible.

`set!` or `!` *name value* sets the value of the named field of the record, if mutable.

Transport Link Cell (TLC) commands

`show` or `s` shows the fields of the TLC.

`keyval` moves to the keyval of the TLC.

`tconc` moves to the tconc of the TLC.

`next` moves to the next link of the TLC.

`ref` or `r` *n* moves to the *nth* field of the symbol. Field 0 is the keyval, field 1 the tconc, and field 2 the next link. *n* defaults to 0.

3.4. The Object Inspector

A facility for noninteractive inspection is also provided to allow construction of different inspection interfaces. Like the interactive facility, it allows objects to be examined in ways not ordinarily possible. The noninteractive system follows a simple, object-oriented protocol. Ordinary Scheme objects are encapsulated in procedures, or inspector objects, that take symbolic messages and return either information about the encapsulated object or new inspector objects that encapsulate pieces of the object.

`(inspect/object` *object*`)` **procedure**
returns: an inspector object procedure
libraries: `(chezscheme)`

`inspect/object` is used to turn an ordinary Scheme object into an inspector object. All inspector objects accept the messages `type`, `print`, `write`, and `size`. The `type` message returns a symbolic representation of the type of the object. The `print` and `write` messages must be accompanied by a port parameter. They cause a representation of the object to be written to the port, using the Scheme procedures `pretty-print` and `write`. The `size` message returns a fixnum representing the size in bytes occupied by the object, including any objects accessible from the current object except those for which the size was already requested via an inspector object derived from the argument of the same `inspect/object` call.

All inspector objects except for variable inspector objects accept the message `value`, which returns the actual object encapsulated in the inspector object.

```
(define x (inspect/object '(1 2 3)))
(x 'type)  ⇒  pair
(define p (open-output-string))
(x 'write p)
(get-output-string p)  ⇒  "(1 2 3)"
(x 'length)  ⇒  (proper 3)
(define y (x 'car))
(y 'type)  ⇒  simple
(y 'value)  ⇒  1
```

Pair inspector objects. Pair inspector objects contain Scheme pairs.

(*pair-object* `'type`) returns the symbol `pair`.

(*pair-object* `'car`) returns an inspector object containing the "car" field of the pair.

(*pair-object* `'cdr`) returns an inspector object containing the "cdr" field of the pair.

(*pair-object* `'length`) returns a list of the form (*type count*). The type field contains the symbol `proper`, the symbol `improper`, or the symbol `circular`, depending on the structure of the list. The count field contains the number of distinct pairs in the list.

Box inspector objects. Box inspector objects contain *Chez Scheme* boxes.

(*box-object* `'type`) returns the symbol `box`.

(*box-object* `'unbox`) returns an inspector object containing the contents of the box.

TLC inspector objects. Box inspector objects contain *Chez Scheme* boxes.

(*tlc-object* `'type`) returns the symbol `tlc`.

(*tlc-object* `'keyval`) returns an inspector object containing the TLC's keyval.

(*tlc-object* `'tconc`) returns an inspector object containing the TLC's tconc.

(*tlc-object* `'next`) returns an inspector object containing the TLC's next link.

Vector, String, Bytevector, and Fxvector inspector objects. Vector (bytevector, string, fxvector) inspector objects contain Scheme vectors (bytevectors, strings, fxvectors).

(*vector-object* `'type`) returns the symbol `vector` (`string`, `bytevector`, `fxvector`).

(*vector-object* `'length`) returns the number of elements in the vector or string.

(*vector-object* `'ref` *n*) returns an inspector object containing the *nth* element of the vector or string.

Simple inspector objects. Simple inspector objects contain unstructured, unmodifiable objects. These include numbers, booleans, the empty list, the end-of-file object, and the void object. They may be examined directly by asking for the `value` of the object.

(*simple-object* `'type`) returns the symbol `simple`.

Unbound inspector objects. Although unbound objects are not normally accessible to Scheme programs, they may be encountered when inspecting variables.

(*unbound-object* `'type`) returns the symbol `unbound`.

Procedure inspector objects. Procedure inspector objects contain Scheme procedures.

(*procedure-object* `'type`) returns the symbol `procedure`.

(*procedure-object* `'length`) returns the number of free variables.

(*procedure-object* `'ref` *n*) returns an inspector object containing the *nth* free variable of the procedure. See the description below of variable inspector objects. *n* must be nonnegative and less than the length of the procedure.

(*procedure-object* `'eval` *expr*) evaluates expr and returns its value. The values of the procedure's free variables are bound within the evaluated expression to identifiers of the form %*n*, where *n* is the location number displayed by the inspector. The values of named variables are also bound to their names.

(*procedure-object* `'code`) returns an inspector object containing the procedure's code object. See the description below of code inspector objects.

Continuation inspector objects. Continuations created by `call/cc` are actually procedures. However, when inspecting such a procedure the underlying data structure that embodies the continuation may be exposed. A continuation structure contains the location at which computation is to resume, the variable values necessary to perform the computation, and a link to the next continuation.

(*continuation-object* `'type`) returns the symbol `continuation`.

(*continuation-object* `'length`) returns the number of free variables.

(*continuation-object* `'ref` *n*) returns an inspector object containing the *nth* free variable of the continuation. See the description below of variable inspector objects. *n* must be nonnegative and less than the length of the continuation.

(*continuation-object* `'eval` *expr*) evaluates expr and returns its value. The values of frame locations are bound within the evaluated expression to identifiers of the form %*n*, where *n* is the location number displayed by the inspector. The values of named locations are also bound to their names.

(*continuation-object* `'code`) returns an inspector object containing the code object for the procedure that was active when the current continuation frame was created. See the description below of code inspector objects.

(*continuation-object* `'depth`) returns the number of frames in the continuation.

(*continuation-object* `'link`) returns an inspector object containing the next continuation frame. The depth must be greater than 1.

(*continuation-object* `'link*` *n*) returns an inspector object containing the *nth* continuation link. *n* must be less than the depth.

(*continuation-object* `'source`) returns an inspector object containing the source information attached to the continuation (representing the source for the application that resulted in the formation of the continuation) or `#f` if no source information is attached.

(*continuation-object* `'source-object`) returns an inspector object containing the source object for the procedure application that resulted in the formation of the continuation or `#f` if no source object is attached.

(*continuation-object* `'source-path`) attempts to find the pathname of the file containing the source for the procedure application that resulted in the formation of the continuation. If successful, three values are returned to identify the file and position of the application within the file: *path*, *line*, and *char*. Two values, a file name and an absolute character position, are returned if the file name is known but the named file cannot be found. The search may be unsuccessful even if a file by the expected name is found in the path if the file has been modified since the source code was compiled. If no file name is known, no values are returned. The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files identified by relative path names.

Code inspector objects. Code inspector objects contain *Chez Scheme* code objects.

(*code-object* `'type`) returns the symbol `code`.

(*code-object* `'name`) returns a string or `#f`. The name associated with a code inspector object is the name of the variable to which the procedure was originally bound or assigned. Since the binding of a variable can be changed, this name association may not always be accurate. `#f` is returned if the inspector cannot determine a name for the procedure.

(*code-object* `'source`) returns an inspector object containing the source information attached to the code object or `#f` if no source information is attached.

(*continuation-object* `'source-object`) returns an inspector object containing the source object for the code object or `#f` if no source object is attached.

(*code-object* `'source-path`) attempts to find the pathname of the file containing the source for the lambda expression that produced the code object. If successful, three values are returned to identify the file and position of the application within the file: *path*, *line*, and

char. Two values, a file name and an absolute character position, are returned if the file name is known but the named file cannot be found. The search may be unsuccessful even if a file by the expected name is found in the path if the file has been modified since the source code was compiled. If no file name is known, no values are returned. The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files identified by relative path names.

(*code-object* `'free-count`) returns the number of free variables in any procedure for which this is the corresponding code.

Variable inspector objects. Variable inspector objects encapsulate variable bindings. Although the actual underlying representation varies, the variable inspector object provides a uniform interface.

(*variable-object* `'type`) returns the symbol `variable`.

(*variable-object* `'name`) returns a symbol or `#f`. `#f` is returned if the name is not available or if the variable is a compiler-generated temporary variable. Variable names are not retained when the parameter `generate-inspector-information` (page 12.6) is false during compilation.

(*variable-object* `'ref`) returns an inspector object containing the current value of the variable.

(*variable-object* `'set!` *e*) returns unspecified, after setting the current value of the variable to *e*. An exception is raised with condition type `&assertion` if the variable is not assignable.

Port inspector objects. Port inspector objects contain ports.

(*port-object* `'type`) returns the symbol `port`.

(*port-object* `'input?`) returns `#t` if the port is an input port, `#f` otherwise.

(*port-object* `'output?`) returns `#t` if the port is an output port, `#f` otherwise.

(*port-object* `'binary?`) returns `#t` if the port is a binary port, `#f` otherwise.

(*port-object* `'closed?`) returns `#t` if the port is closed, `#f` if the port is open.

(*port-object* `'name`) returns an inspector object containing the port's name.

(*port-object* `'handler`) returns a procedure inspector object encapsulating the port handler, such as would be returned by `port-handler`.

(*port-object* `'output-size`) returns the output buffer size as a fixnum if the port is an output port (otherwise the value is unspecified).

(*port-object* `'output-index`) returns the output buffer index as a fixnum if the port is an output port (otherwise the value is unspecified).

(*port-object* `'output-buffer)` returns an inspector object containing the string used for buffered output.

(*port-object* `'input-size)` returns the input buffer size as a fixnum if the port is an input port (otherwise the value is unspecified).

(*port-object* `'input-index)` returns the input buffer index as a fixnum if the port is an input port (otherwise the value is unspecified).

(*port-object* `'input-buffer)` returns an inspector object containing the string used for buffered input.

Symbol inspector objects. Symbol inspector objects contain symbols. These include gensyms.

(*symbol-object* `'type)` returns the symbol `symbol`.

(*symbol-object* `'name)` returns a string inspector object. The string name associated with a symbol inspector object is the print representation of a symbol, such as would be returned by the procedure `symbol->string`.

(*symbol-object* `'gensym?)` returns `#t` if the symbol is a gensym, `#f` otherwise. Gensyms are created by `gensym`.

(*symbol-object* `'top-level-value)` returns an inspector object containing the global value of the symbol.

(*symbol-object* `'property-list)` returns an inspector object containing the property list for the symbol.

Record inspector objects. Record inspector objects contain records.

(*record-object* `'type)` returns the symbol `record`.

(*record-object* `'name)` returns a string inspector object corresponding to the name of the record type.

(*record-object* `'fields)` returns an inspector object containing a list of the field names of the record type.

(*record-object* `'length)` returns the number of fields.

(*record-object* `'rtd)` returns an inspector object containing the record-type descriptor of the record type.

(*record-object* `'accessible?` *name*`)` returns `#t` if the named field is accessible, `#f` otherwise. A field may be inaccessible if optimized away by the compiler.

(*record-object* `'ref` *name*) returns an inspector object containing the value of the named field. An exception is raised with condition type `&assertion` if the named field is not accessible.

(*record-object* `'mutable?` *name*) returns `#t` if the named field is mutable, `#f` otherwise. A field is immutable if it is not declared mutable or if the compiler optimizes away all assignments to the field.

(*record-object* `'set!` *name value*) sets the value of the named field to *value*. An exception is raised with condition type `&assertion` if the named field is not assignable.

3.5. Locating objects

(`make-object-finder` *pred*) **procedure**
(`make-object-finder` *pred g*) **procedure**
(`make-object-finder` *pred x g*) **procedure**
returns: see below
libraries: `(chezscheme)`

The procedure `make-object-finder` takes a predicate *pred* and two optional arguments: a starting point x and a maximum generation g. The starting point defaults to the value of the procedure `oblist`, and the maximum generation defaults to the value of the parameter `collect-maximum-generation`. `make-object-finder` returns an object finder p that can be used to search for objects satisfying *pred* within the starting-point object x. Immediate objects and objects in generations older than g are treated as leaves. p is a procedure accepting no arguments. If an object y satisfying *pred* can be found starting with x, p returns a list whose first element is y and whose remaining elements represent the path of objects from x to y, listed in reverse order. p can be invoked multiple times to find additional objects satisfying the predicate, if any. p returns `#f` if no more objects matching the predicate can be found.

p maintains internal state recording where it has been so it can restart at the point of the last found object and not return the same object twice. The state can be several times the size of the starting-point object x and all that is reachable from x.

The interactive inspector provides a convenient interface to the object finder in the form of `find` and `find-next` commands.

Relocation tables for static code objects are discarded by default, which prevents object finders from providing accurate results when static code objects are involved. That is, they will not find any objects pointed to directly from a code object that has been promoted to the static generation. If this is a problem, the command-line argument `--retain-static-relocation` can be used to prevent the relocation tables from being discarded.

3.6. Nested object size and composition

The procedures `compute-size` and `compute-composition` can be used to determine the size or composition of an object, including anything reachable via pointers from the object. Depending on the number of objects reachable from the object, the procedures potentially allocate a large amount of memory. In an application for which knowing the number, size, generation, and types of all objects in the heap is sufficient, `object-counts` is potentially much more efficient.

These procedures treat immediate objects such as fixnums, booleans, and characters as zero-count, zero-byte leaves.

By default, these procedures also treat static objects (those in the initial heap) as zero-count, zero-byte leaves. Both procedures accept an optional second argument that specifies the maximum generation of interest, with the symbol `static` being used to represent the static generation.

Objects sometimes point to a great deal more than one might expect. For example, if static data is included, the procedure value of `(lambda (x) x)` points indirectly to the exception handling subsystem (because of the argument-count check) and many other things as a result of that.

Relocation tables for static code objects are discarded by default, which prevents these procedures from providing accurate results when static code objects are involved. That is, they will not find any objects pointed to directly from a code object that has been promoted to the static generation. If accurate sizes and compositions for static code objects are required, the command-line argument `--retain-static-relocation` can be used to prevent the relocation tables from being discarded.

`(compute-size` *object*`)` **procedure**
`(compute-size` *object generation*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

object can be any object. *generation* must be a fixnum between 0 and the value of `collect-maximum-generation`, inclusive, or the symbol `static`. If *generation* is not supplied, it defaults to the value of `collect-maximum-generation`.

`compute-size` returns the amount of memory, in bytes, occupied by *object* and anything reachable from *object* in any generation less than or equal to *generation*. Immediate values such as fixnums, booleans, and characters have zero size.

The following examples are valid for machines with 32-bit pointers.

```
(compute-size 0)  ⇒  0
(compute-size (cons 0 0))  ⇒  8
(compute-size (cons (vector #t #f) 0))  ⇒  24
```

```
(compute-size
  (let ([x (cons 0 0)])
    (set-car! x x)
    (set-cdr! x x)
    x))                    ⇒  8

(define-record-type frob (fields x))
(collect 1 1) ; force rtd into generation 1
(compute-size
  (let ([x (make-frob 0)])
    (cons x x))
  0)                       ⇒  16
```

`(compute-composition` *object*`)` **procedure**
`(compute-composition` *object generation*`)` **procedure**

returns: see below
libraries: `(chezscheme)`

object can be any object. *generation* must be a fixnum between 0 and the value of `collect-maximum-generation`, inclusive, or the symbol `static`. If *generation* is not supplied, it defaults to the value of `collect-maximum-generation`.

`compute-composition` returns an association list representing the composition of *object*, including anything reachable from it in any generation less than or equal to *generation*. The association list has the following structure:

`((`*type count* `.` *bytes*`) ...)`

type is either the name of a primitive type, represented as a symbol, e.g., `pair`, or a record-type descriptor (rtd). *count* and *bytes* are nonnegative fixnums.

Immediate values such as fixnums, booleans, and characters are not included in the composition.

The following examples are valid for machines with 32-bit pointers.

```
(compute-composition 0)  ⇒  ()
(compute-composition (cons 0 0))  ⇒  ((pair 1 . 8))
(compute-composition
  (cons (vector #t #f) 0))  ⇒  ((pair 1 . 8) (vector 1 . 16))

(compute-composition
  (let ([x (cons 0 0)])
    (set-car! x x)
    (set-cdr! x x)
    x))                    ⇒  ((pair 1 . 8)

(define-record-type frob (fields x))
(collect 1 1) ; force rtd into generation 1
(compute-composition
  (let ([x (make-frob 0)])
    (cons x x))
```

```
0)                          ⇒  ((pair 1 . 8)
                               (#<record type frob> 1 . 8))
```

4. Foreign Interface

Chez Scheme provides two ways to interact with "foreign" code, i.e., code written in other languages. The first is via subprocess creation and communication, which is discussed in the Section 4.1. The second is via static or dynamic loading and invocation from Scheme of procedures written in C and invocation from C of procedures written in Scheme. These mechanisms are discussed in Sections 4.2 through 4.4.

The method for static loading of C object code is dependent upon which machine you are running; see the installation instructions distributed with *Chez Scheme*.

4.1. Subprocess Communication

Two procedures, `system` and `process`, are used to create subprocesses. Both procedures accept a single string argument and create a subprocess to execute the shell command contained in the string. The `system` procedure waits for the process to exit before returning, however, while the `process` procedure returns immediately without waiting for the process to exit. The standard input and output files of a subprocess created by `system` may be used to communicate with the user's console. The standard input and output files of a subprocess created by `process` may be used to communicate with the Scheme process.

`(system` *command*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

command must be a string.

The `system` procedure creates a subprocess to perform the operation specified by *command*. The subprocess may communicate with the user through the same console input and console output files used by the Scheme process. After creating the subprocess, `system` waits for the process to exit before returning.

When the subprocess exits, `system` returns the exit code for the subprocess, unless (on Unix-based systems) a signal caused the subprocess to terminate, in which case `system` returns the negation of the signal that caused the termination, e.g., -1 for `SIGHUP`.

`(open-process-ports` *command*`)` **procedure**
`(open-process-ports` *command b-mode*`)` **procedure**
`(open-process-ports` *command b-mode ?transcoder*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

command must be a string. If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure creates textual ports, each of whose transcoder is *?transcoder*. Otherwise, this procedure returns binary ports. *b-mode* specifies the buffer mode used by each of the ports returned by this procedure and defaults to `block`. Buffer modes are described in Section 7.2 of *The Scheme Programming Language, 4th Edition*.

`open-process-ports` creates a subprocess to perform the operation specified by *command*. Unlike `system`, `process` returns immediately after creating the subprocess, i.e., without waiting for the subprocess to terminate. It returns four values:

1. *to-stdin* is an output port to which Scheme can send output to the subprocess through the subprocess's standard input file.
2. *from-stdout* is an input port from which Scheme can read input from the subprocess through the subprocess's standard output file.
3. *from-stderr* is an input port from which Scheme can read input from the subprocess through the subprocess's standard error file.
4. *process-id* is an integer identifying the created subprocess provided by the host operating system.

If the process exits or closes its standard output file descriptor, any procedure that reads input from *from-stdout* will return an end-of-file object. Similarly, if the process exits or closes its standard error file descriptor, any procedure that reads input from *from-stderr* will return an end-of-file object.

The predicate `input-port-ready?` may be used to detect whether input has been sent by the subprocess to Scheme.

It is sometimes necessary to force output to be sent immediately to the subprocess by invoking `flush-output-port` on *to-stdin*, since *Chez Scheme* buffers the output for efficiency.

On UNIX systems, the *process-id* is the process identifier for the shell created to execute *command*. If *command* is used to invoke an executable file rather than a shell command, it may be useful to prepend *command* with the string `"exec "`, which causes the shell to load and execute the named executable directly, without forking a new process—the shell equivalent of a tail call. This will reduce by one the number of subprocesses created and cause *process-id* to reflect the process identifier for the executable once the shell has transferred control.

`(process` *command*`)` **procedure**
returns: see explanation
libraries: `(chezscheme)`

command must be a string.

`process` is similar to `open-process-ports`, but less general. It does not return a port from which the subproces's standard error output can be read, and it always creates textual ports. It returns a list of three values rather than the four separate values of `open-process-ports`. The returned list contains, in order: *from-stdout*, *to-stdin*, and *process-id*, which correspond to the second, first, and fourth return values of `open-process-ports`.

4.2. Calling out of Scheme

Chez Scheme's foreign-procedure interface allows a Scheme program to invoke procedures written in C or in languages that obey the same calling conventions as C. Two steps are necessary before foreign procedures can be invoked from Scheme. First, the foreign procedure must be compiled and loaded, either statically or dynamically, as described in Section 4.6. Then, access to the foreign procedure must be established in Scheme, as described in this section. Once access to a foreign procedure has been established it may be called as an ordinary Scheme procedure.

Since foreign procedures operate independently of the Scheme memory management and exception handling system, great care must be taken when using them. Although the foreign-procedure interface provides type checking (at optimize levels less than 3) and type conversion, the programmer must ensure that the sharing of data between Scheme and foreign procedures is done safely by specifying proper argument and result types.

Scheme-callable wrappers for foreign procedures can also be created via `ftype-ref` and function ftypes (Section 4.5).

`(foreign-procedure` *conv* `...` *entry-exp* `(`*param-type* `...)` *res-type*`)` **syntax**
returns: a procedure
libraries: `(chezscheme)`

entry-exp must evaluate to a string representing a valid foreign procedure entry point or an integer representing the address of the foreign procedure. The *param-types* and *res-type* must be symbols or structured forms as described below. When a `foreign-procedure` expression is evaluated, a Scheme procedure is created that will invoke the foreign procedure specified by *entry-exp*. When the procedure is called each argument is checked and converted according to the specified *param-type* before it is passed to the foreign procedure. The result of the foreign procedure call is converted as specified by the *res-type*. Multiple procedures may be created for the same foreign entry.

Each *conv* adjusts specifies the calling convention to be used. A `#f` is allowed as *conv* to indicate the default calling convention on the target machine (so the `#f` has no effect). Three other conventions are currently supported under Windows: `__stdcall`, `__cdecl`, and `__com` (32-bit only). Since `__cdecl` is the default, specifying `__cdecl` is equivalent to specifying `#f` or no convention. Finally, *conv* can be `__collect_safe` to indicate that garbage collection is allowed concurrent to a call of the foreign procedure.

Use `__stdcall` to access most Windows API procedures. Use `__cdecl` for Windows API varargs procedures, for C library procedures, and for most other procedures. Use `__com` to invoke COM interface methods; COM uses the `__stdcall` convention but additionally performs the indirections necessary to obtain the correct method from a COM instance. The address of the COM instance must be passed as the first argument, which should normally be declared as `iptr`. For the `__com` interface only, *entry-exp* must evaluate to the byte offset of the method in the COM vtable. For example,

```
(foreign-procedure __com 12 (iptr double-float) integer-32)
```

creates an interface to a COM method at offset 12 in the vtable encapsulated within the COM instance passed as the first argument, with the second argument being a double float and the return value being an integer.

Use `__collect_safe` to declare that garbage collection is allowed concurrent to the foreign procedure. The `__collect_safe` declaration allows concurrent collection by deactivating the current thread (see `fork-thread`) when the foreign procedure is called, and the thread is activated again when the foreign procedure returns. The `__collect_safe` declaration is useful, for example, when calling a blocking I/O call to allow other Scheme threads to run normally. Refrain from passing collectable memory to a `__collect_safe` foreign procedure, or use `lock-object` to lock the memory in place; see also `Sdeactivate_thread`. The `__collect_safe` declaration has no effect on a non-threaded version of the system.

For example, calling the C `sleep` function with the default convention will block other Scheme threads from performing a garbage collection, but adding the `__collect_safe` declaration avoids that problem:

```
(define c-sleep
  (foreign-procedure __collect_safe "sleep" (unsigned) unsigned))
(c-sleep 10) ; sleeps for 10 seconds without blocking other threads
```

If a foreign procedure that is called with `__collect_safe` can invoke callables, then each callable should also be declared with `__collect_safe` so that the callable reactivates the thread.

Complete type checking and conversion is performed on the parameters to a foreign procedure. The types `scheme-object`, `string`, `wstring`, `u8*`, `u16*`, `u32*`, `utf-8`, `utf-16le`, `utf-16be`, `utf-32le`, and `utf-32be`, must be used with caution, however, since they allow allocated Scheme objects to be used in places the Scheme memory management system cannot control. No problems will arise as long as such objects are not retained in foreign variables or data structures while Scheme code is running, and as long as they are not passed as arguments to a `__collect_safe` procedure, since garbage collection can occur only while Scheme code is running or when concurrent garbage collection is enabled. Other parameter types are converted to equivalent foreign representations and consequently they can be retained indefinitely in foreign variables and data structures.

For argument types `string`, `wstring`, `utf-8`, `utf-16le`, `utf-16be`, `utf-32le`, and `utf-32be`, an argument is converted to a fresh object that is passed to the foreign procedure. Since the fresh object is not accessible for locking before the call, it can never be treated correctly for a `__collect_safe` foreign procedure, so those types are disallowed as argument types

for a `__collect_safe` foreign procedure. For analogous reasons, those types are disallowed as the result of a `__collect_safe` foreign callable.

Following are the valid parameter types:

`integer-8`: Exact integers from -2^7 through $2^8 - 1$ are valid. Integers in the range 2^7 through $2^8 - 1$ are treated as two's complement representations of negative numbers, e.g., `#xff` is treated as -1. The argument is passed to C as an integer of the appropriate size (usually `signed char`).

`unsigned-8`: Exact integers from -2^7 to $2^8 - 1$ are valid. Integers in the range -2^7 through -1 are treated as the positive equivalents of their two's complement representation, e.g., -1 is treated as `#xff`. The argument is passed to C as an unsigned integer of the appropriate size (usually `unsigned char`).

`integer-16`: Exact integers from -2^{15} through $2^{16} - 1$ are valid. Integers in the range 2^{15} through $2^{16} - 1$ are treated as two's complement representations of negative numbers, e.g., `#xffff` is treated as -1. The argument is passed to C as an integer of the appropriate size (usually `short`).

`unsigned-16`: Exact integers from -2^{15} to $2^{16} - 1$ are valid. Integers in the range -2^{15} through -1 are treated as the positive equivalents of their two's complement representation, e.g., -1 is treated as `#xffff`. The argument is passed to C as an unsigned integer of the appropriate size (usually `unsigned short`).

`integer-32`: Exact integers from -2^{31} through $2^{32} - 1$ are valid. Integers in the range 2^{31} through $2^{32} - 1$ are treated as two's complement representations of negative numbers, e.g., `#xffffffff` is treated as -1. The argument is passed to C as an integer of the appropriate size (usually `int`).

`unsigned-32`: Exact integers from -2^{31} to $2^{32} - 1$ are valid. Integers in the range -2^{31} through -1 are treated as the positive equivalents of their two's complement representation, e.g., -1 is treated as `#xffffffff`. The argument is passed to C as an unsigned integer of the appropriate size (usually `unsigned int`).

`integer-64`: Exact integers from -2^{63} through $2^{64} - 1$ are valid. Integers in the range 2^{63} through $2^{64} - 1$ are treated as two's complement representations of negative numbers. The argument is passed to C as an integer of the appropriate size (usually `long long` or, on many 64-bit platforms, `long`).

`unsigned-64`: Exact integers from -2^{63} through $2^{64} - 1$ are valid. Integers in the range -2^{63} through -1 are treated as the positive equivalents of their two's complement representation, The argument is passed to C as an integer of the appropriate size (usually `unsigned long long` or, on many 64-bit platforms, `long`).

`double-float`: Only Scheme flonums are valid—other Scheme numeric types are not automatically converted. The argument is passed to C as a double float.

`single-float`: Only Scheme flonums are valid—other Scheme numeric types are not automatically converted. The argument is passed to C as a single float. Since *Chez Scheme* represents flonums in double-float format, the parameter is first converted into single-float format.

`short`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `short`.

`unsigned-short`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `unsigned short`.

`int`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `int`.

`unsigned`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `unsigned`.

`unsigned-int`: This type is an alias `unsigned`. fixed-size type above, depending on the size of a C `unsigned`.

`long`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `long`.

`unsigned-long`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `unsigned long`.

`long-long`: This type is an alias for the appropriate fixed-size type above, depending on the size of the nonstandard C type `long long`.

`unsigned-long-long`: This type is an alias for the appropriate fixed-size type above, depending on the size of the nonstandard C type `unsigned long long`.

`ptrdiff_t`: This type is an alias for the appropriate fixed-size type above, depending on its definition in the host machine's `stddef.h` include file.

`size_t`: This type is an alias for the appropriate unsigned fixed-size type above, depending on its definition in the host machine's `stddef.h` include file.

`ssize_t`: This type is an alias for the appropriate signed fixed-size type above, depending on its definition in the host machine's `stddef.h` include file.

`iptr`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C pointer.

`uptr`: This type is an alias for the appropriate (unsigned) fixed-size type above, depending on the size of a C pointer.

`void*`: This type is an alias for `uptr`.

`fixnum`: This type is equivalent to `iptr`, except only values in the fixnum range are valid. Transmission of fixnums is slightly faster than transmission of `iptr` values, but the fixnum range is smaller, so some `iptr` values do not have a fixnum representation.

`boolean`: Any Scheme object may be passed as a boolean. `#f` is converted to 0; all other objects are converted to 1. The argument is passed to C as an `int`.

`char`: Only Scheme characters with Unicode scalar values in the range 0 through 255 are valid `char` parameters. The character is converted to its Unicode scalar value, as with `char->integer`, and passed to C as an `unsigned char`.

`wchar_t`: Only Scheme characters are valid `wchar_t` parameters. Under Windows and any other system where `wchar_t` holds only 16-bit values rather than full Unicode scalar values, only characters with 16-bit Unicode scalar values are valid. On systems where `wchar_t` is a full 32-bit value, any Scheme character is valid. The character is converted to its Unicode scalar value, as with `char->integer`, and passed to C as a `wchar_t`.

`wchar`: This type is an alias for `wchar_t`.

`double`: This type is an alias for `double-float`.

`float`: This type is an alias for `single-float`.

`scheme-object`: The argument is passed directly to the foreign procedure; no conversion or type checking is performed. This form of parameter passing should be used with discretion. Scheme objects should not be preserved in foreign variables or data structures since the memory management system may relocate them between foreign procedure calls.

`ptr`: This type is an alias for `scheme-object`.

`u8*`: The argument must be a Scheme bytevector or `#f`. For `#f`, the null pointer (0) is passed to the foreign procedure. For a bytevector, a pointer to the first byte of the bytevector's data is passed. If the C routine to which the data is passed requires the input to be null-terminated, a null (0) byte must be included explicitly in the bytevector. The bytevector should not be retained in foreign variables or data structures, since the memory management system may relocate or discard them between foreign procedure calls, and use their storage for some other purpose.

`u16*`: Arguments of this type are treated just like arguments of type `u8*`. If the C routine to which the data is passed requires the input to be null-terminated, two null (0) bytes must be included explicitly in the bytevector, aligned on a 16-bit boundary.

`u32*`: Arguments of this type are treated just like arguments of type `u8*`. If the C routine to which the data is passed requires the input to be null-terminated, four null (0) bytes must be included explicitly in the bytevector, aligned on a 32-bit boundary.

`utf-8`: The argument must be a Scheme string or `#f`. For `#f`, the null pointer (0) is passed to the foreign procedure. A string is converted into a bytevector, as if via `string->utf8`,

with an added null byte, and the address of the first byte of the bytevector is passed to C. The bytevector should not be retained in foreign variables or data structures, since the memory management system may relocate or discard them between foreign procedure calls and use their storage for some other purpose. The `utf-8` argument type is not allowed for a `__collect_safe` foreign procedure.

`utf-16le`: Arguments of this type are treated like arguments of type `utf-8`, except they are converted as if via `string->utf16` with endianness `little`, and they are extended by two null bytes rather than one.

`utf-16be`: Arguments of this type are treated like arguments of type `utf-8`, except they are converted as if via `string->utf16` with endianness `big`, and they are extended by two null bytes rather than one.

`utf-32le`: Arguments of this type are treated like arguments of type `utf-8`, except they are converted as if via `string->utf32` with endianness `little`, and they are extended by four null bytes rather than one.

`utf-32be`: Arguments of this type are treated like arguments of type `utf-8`, except they are converted as if via `string->utf32` with endianness `big`, and they are extended by four null bytes rather than one.

`string`: This type is an alias for `utf-8`.

`wstring`: This type is an alias for `utf-16le`, `utf-16be`, `utf-32le`, or `utf-32be` as appropriate depending on the size of a C `wchar_t` and the endianness of the target machine. For example, `wstring` is equivalent to `utf-16le` under Windows running on Intel hardware.

`(*` *ftype*`)`: This type allows a pointer to a foreign type (ftype) to be passed. The argument must be an ftype pointer of type *ftype*, and the actual argument is the address encapsulated in the ftype pointer. See Section 4.5 for a description of foreign types.

`(&` *ftype*`)`: This type allows a foreign type (ftype) to be passed as a value, but represented on the Scheme side as a pointer to the foreign-type data. That is, a `(&` *ftype*`)` argument is represented on the Scheme side the same as a `(*` *ftype*`)` argument, but a `(&` *ftype*`)` argument is passed to the foreign procedure as the content at the foreign pointer's address instead of as the address. For example, if *ftype* is a `struct` type, then `(&` *ftype*`)` passes a struct argument instead of a struct-pointer argument. The *ftype* cannot refer to an array type.

The result types are similar to the parameter types with the addition of a `void` type. In general, the type conversions are the inverse of the parameter type conversions. No error checking is performed on return, since the system cannot determine whether a foreign result is actually of the indicated type. Particular caution should be exercised with the result types `scheme-object`, `double-float`, `double`, `single-float`, `float`, and the types that result in the construction of bytevectors or strings, since invalid return values may lead to invalid memory references as well as incorrect computations. Following are the valid result types:

`void`: The result of the foreign procedure call is ignored and an unspecified Scheme object is returned. `void` should be used when foreign procedures are called for effect only.

`integer-8`: The result is interpreted as a signed 8-bit integer and is converted to a Scheme exact integer.

`unsigned-8`: The result is interpreted as an unsigned 8-bit integer and is converted to a Scheme nonnegative exact integer.

`integer-16`: The result is interpreted as a signed 16-bit integer and is converted to a Scheme exact integer.

`unsigned-16`: The result is interpreted as an unsigned 16-bit integer and is converted to a Scheme nonnegative exact integer.

`integer-32`: The result is interpreted as a signed 32-bit integer and is converted to a Scheme exact integer.

`unsigned-32`: The result is interpreted as an unsigned 32-bit integer and is converted to a Scheme nonnegative exact integer.

`integer-64`: The result is interpreted as a signed 64-bit integer and is converted to a Scheme exact integer.

`unsigned-64`: The result is interpreted as an unsigned 64-bit integer and is converted to a Scheme nonnegative exact integer.

`double-float`: The result is interpreted as a double float and is translated into a *Chez Scheme* flonum.

`single-float`: The result is interpreted as a single float and is translated into a *Chez Scheme* flonum. Since *Chez Scheme* represents flonums in double-float format, the result is first converted into double-float format.

`short`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `short`.

`unsigned-short`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `unsigned short`.

`int`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `int`.

`unsigned`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `unsigned`.

`unsigned-int`: This type is an alias `unsigned`. fixed-size type above, depending on the size of a C `unsigned`.

`long`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `long`.

`unsigned-long`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C `unsigned long`.

`long-long`: This type is an alias for the appropriate fixed-size type above, depending on the size of the nonstandard C type `long long`.

`unsigned-long-long`: This type is an alias for the appropriate fixed-size type above, depending on the size of the nonstandard C type `unsigned long long`.

`ptrdiff_t`: This type is an alias for the appropriate fixed-size type above, depending on its definition in the host machine's `stddef.h` include file.

`size_t`: This type is an alias for the appropriate unsigned fixed-size type above, depending on its definition in the host machine's `stddef.h` include file.

`ssize_t`: This type is an alias for the appropriate signed fixed-size type above, depending on its definition in the host machine's `stddef.h` include file.

`iptr`: This type is an alias for the appropriate fixed-size type above, depending on the size of a C pointer.

`uptr`: This type is an alias for the appropriate (unsigned) fixed-size type above, depending on the size of a C pointer.

`void*`: This type is an alias for `uptr`.

`boolean`: This type converts a C `int` return value into a Scheme boolean. 0 is converted to `#f`; all other values are converted to `#t`.

`char`: This type converts a C `unsigned char` return value into a Scheme character, as if via `integer->char`.

`wchar_t`: This type converts a C `wchar_t` return value into a Scheme character, as if via `integer->char`. The `wchar_t` value must be a valid Unicode scalar value.

`wchar`: This type is an alias for `wchar_t`.

`double`: This type is an alias for `double-float`.

`float`: This type is an alias for `single-float`.

`scheme-object`: The result is assumed to be a valid Scheme object, and no conversion is performed. This type is inherently dangerous, since an invalid Scheme object can corrupt the memory management system with unpredictable (but always unpleasant) results. Since Scheme objects are actually typed pointers, even integers cannot safely be returned as type `scheme-object` unless they were created by the Scheme system.

`ptr`: This type is an alias for `scheme-object`.

`u8*`: The result is interpreted as a pointer to a null-terminated sequence of 8-bit unsigned integers (bytes). If the result is a null pointer, `#f` is returned. Otherwise, the sequence of bytes is stored in a freshly allocated bytevector of the appropriate length, and the bytevector is returned to Scheme.

`u16*`: The result is interpreted as a pointer to a null-terminated sequence of 16-bit unsigned integers. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of 16-bit integers is stored in a freshly allocated bytevector of the appropriate length, and the bytevector is returned to Scheme. The null terminator must be a properly aligned 16-bit word, i.e., two bytes of zero aligned on a 16-bit boundary.

`u32*`: The result is interpreted as a pointer to a null-terminated sequence of 32-bit unsigned integers. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of 16-bit integers is stored in a freshly allocated bytevector of the appropriate length, and the bytevector is returned to Scheme. The null terminator must be a properly aligned 32-bit word, i.e., four bytes of zero aligned on a 32-bit boundary.

`utf-8`: The result is interpreted as a pointer to a null-terminated sequence of 8-bit unsigned character values. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of bytes is converted into a Scheme string, as if via `utf8->string`, and the string is returned to Scheme.

`utf-16le`: The result is interpreted as a pointer to a null-terminated sequence of 16-bit unsigned integers. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of integers is converted into a Scheme string, as if via `utf16->string` with endianness `little`, and the string is returned to Scheme. A byte-order mark in the sequence of integers as treated as an ordinary character value and does not affect the byte ordering.

`utf-16be`: The result is interpreted as a pointer to a null-terminated sequence of 16-bit unsigned integers. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of integers is converted into a Scheme string, as if via `utf16->string` with endianness `big`, and the string is returned to Scheme. A byte-order mark in the sequence of integers as treated as an ordinary character value and does not affect the byte ordering.

`utf-32le`: The result is interpreted as a pointer to a null-terminated sequence of 32-bit unsigned integers. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of integers is converted into a Scheme string, as if via `utf32->string` with endianness `little`, and the string is returned to Scheme. A byte-order mark in the sequence of integers as treated as an ordinary character value and does not affect the byte ordering.

`utf-32be`: The result is interpreted as a pointer to a null-terminated sequence of 32-bit unsigned integers. If the result is a null pointer, `#f` is returned. Otherwise, the sequence of integers is converted into a Scheme string, as if via `utf32->string` with endianness `big`, and the string is returned to Scheme. A byte-order mark in the sequence of integers as treated as an ordinary character value and does not affect the byte ordering.

`string`: This type is an alias for `utf-8`.

`wstring`: This type is an alias for `utf-16le`, `utf-16be`, `utf-32le`, or `utf-32be` as appropriate depending on the size of a C `wchar_t` and the endianness of the target machine. For example, `wstring` is equivalent to `utf-16le` under Windows running on Intel hardware.

(`*` *ftype*): The result is interpreted as the address of a foreign object whose structure is described by *ftype*, and a freshly allocated ftype pointer encapsulating the address is returned. See Section 4.5 for a description of foreign types.

(`&` *ftype*): The result is interpreted as a foreign object whose structure is described by *ftype*, where the foreign procedure returns a *ftype* result, but the caller must provide an extra (`*` *ftype*) argument before all other arguments to receive the result. An unspecified Scheme object is returned when the foreign procedure is called, since the result is instead written into storage referenced by the extra argument. The *ftype* cannot refer to an array type.

Consider a C identity procedure:

```
int id(x) int x; { return x; }
```

After a file containing this procedure has been compiled and loaded (see Section 4.6) it can be accessed as follows:

```
(foreign-procedure "id"
  (int) int)  ⇒  #<procedure>
((foreign-procedure "id"
   (int) int)
 1)  ⇒  1
(define int-id
  (foreign-procedure "id"
    (int) int))
(int-id 1)  ⇒  1
```

The `"id"` entry can also be interpreted as accepting and returning a boolean:

```
(define bool-id
  (foreign-procedure "id"
    (boolean) boolean))
(bool-id #f)  ⇒  #f
(bool-id #t)  ⇒  #t
(bool-id 1)  ⇒  #t
```

As the last example reveals, `bool-id` is actually a conversion procedure. When a Scheme object is passed as type `boolean` it is converted to 0 or 1, and when it is returned it is converted to `#f` or `#t`. As a result objects are converted to normalized boolean values. The `"id"` entry can be used to create other conversion procedures by varying the type specifications:

```
(define int->bool
  (foreign-procedure "id"
    (int) boolean))
(int->bool 0)  ⇒  #f
(int->bool 5)  ⇒  #t
(map (foreign-procedure "id"
       (boolean) int)
     '(#t #f))  ⇒  (1 0)
(define void
  (foreign-procedure "id"
    (int) void))
(void 10)  ⇒  unspecified
```

There are, of course, simpler and more efficient ways of accomplishing these conversions directly in Scheme.

A foreign entry is resolved when a `foreign-procedure` expression is evaluated, rather than either when the code is loaded or each time the procedure is invoked. Thus, the following definition is always valid since the `foreign-procedure` expression is not immediately evaluated:

```
(define doit
  (lambda ()
    ((foreign-procedure "doit" () void))))
```

`doit` should not be invoked, however, before an entry for `"doit"` has been provided. Similarly, an entry for `"doit"` must exist before the following code is evaluated:

```
(define doit
  (foreign-procedure "doit" () void))
```

Although the second definition is more constraining on the load order of foreign files, it is more efficient since the entry resolution need be done only once.

It is often useful to define a template to be used in the creation of several foreign procedures with similar argument types and return values. For example, the following code creates two foreign procedures from a single foreign procedure expression, by abstracting out the foreign procedure name:

```
(define double->double
  (lambda (proc-name)
    (foreign-procedure proc-name
      (double)
      double)))

(define log10 (double->double "log10"))
(define gamma (double->double "gamma"))
```

Both `"log10"` and `"gamma"` must be available as foreign entries (see Section 4.6) before the corresponding definitions. The use of foreign procedure templates can simplify the coding process and reduce the amount of code generated when a large number of foreign

procedures are involved, e.g., when an entire library of foreign procedures is imported into Scheme.

4.3. Calling into Scheme

Section 4.2 describes the `foreign-procedure` form, which permits Scheme code to invoke C or C-compatible foreign procedures. This section describes the `foreign-callable` form, which permits C or C-compatible code to call Scheme procedures. A more primitive mechanism for calling Scheme procedures from C is described in Section 4.8.

As when calling foreign procedures from Scheme, great care must be taken when sharing data between Scheme and foreign code that calls Scheme to avoid corrupting Scheme's memory management system.

A foreign-callable wrapper for a Scheme procedure can also be created by passing the procedure to `make-ftype-pointer` with an appropriate function ftype (Section 4.5).

`(foreign-callable` *conv* `...` *proc-exp* `(`*param-type* `...)` *res-type*`)` **syntax**
returns: a code object
libraries: `(chezscheme)`

proc-exp must evaluate to a procedure, the Scheme procedure that is to be invoked by foreign code. The parameter and result types are as described for `foreign-procedure` in Section 4.2, except that the requirements and conversions are effectively reversed, e.g., the conversions described for `foreign-procedure` arguments are performed for `foreign-callable` return values. A `(&` *ftype*`)` argument to the callable refers to an address that is valid only during the dynamic extent of the callback invocation. A `(&` *ftype*`)` result type for a callable causes the Scheme procedure to receive an extra `(&` *ftype*`)` argument before all others; the Scheme procedure should write a result into the extra argument, and the direct result of the Scheme procedure is ignored. Type checking is performed for result values but not argument values, since the parameter values are provided by the foreign code and must be assumed to be correct.

Each *conv* adjusts the calling convention to be used. `foreign-callable` supports the same conventions as `foreign-procedure` with the exception of `__com`. The `__collect_safe` convention for a callable activates a calling thread if the thread is not already activated, and the thread's activation state is reverted when the callable returns. If a calling thread is not currently registered with the Scheme system, then reverting the thread's activation state implies destroying the thread's registration (see `Sdestroy_thread`).

The value produced by `foreign-callable` is a Scheme code object, which contains some header information as well as code that performs the call to the encapsulated Scheme procedure. The code object may be converted into a foreign-callable address via `foreign-callable-entry-point`, which returns an integer representing the address of the entry point within the code object. (The C-callable library function `Sforeign_callable_entry_point`, described in Section 4.8, may be used to obtain the entry point as well.) This is an implicit pointer into a Scheme object, and in many cases, it

is necessary to lock the code object (using `lock-object`) before converting it into an entry point to prevent Scheme's storage management system from relocating or destroying the code object, e.g., when the entry point is registered as a callback and retained in the "C" side indefinitely.

The following code creates a foreign-callable code object, locks the code object, and returns the entry point.

```
(let ([x (foreign-callable
           (lambda (x y) (pretty-print (cons x (* y 2))))
           (string integer-32)
           void)])
  (lock-object x)
  (foreign-callable-entry-point x))
```

Unless the entry point is intended to be permanent, a pointer to the code object returned by `foreign-callable` should be retained so that it can be unlocked when no longer needed.

Mixed use of `foreign-callable` and `foreign-procedure` may result in nesting of foreign and Scheme calls, and this results in some interesting considerations when continuations are involved, directly or indirectly (as via the default exception handler). See Section 4.4 for a discussion of the interaction between foreign calls and continuations.

The following example demonstrates how the "callback" functions required by many windowing systems might be defined in Scheme with the use of `foreign-callable`. Assume that the following C code has been compiled and loaded (see Section 4.6).

```
#include <stdio.h>

typedef void (*CB)(char);

CB callbacks[256];

void cb_init(void) {
   int i;

   for (i = 0; i < 256; i += 1)
       callbacks[i] = (CB)0;
}

void register_callback(char c, CB cb) {
    callbacks[c] = cb;
}

void event_loop(void) {
    CB f; char c;

    for (;;) {
        c = getchar();
        if (c == EOF) break;
        f = callbacks[c];
        if (f != (CB)0) f(c);
    }
```

```
}
```

Interfaces to these functions may be defined in Scheme as follows.

```
(define cb-init
  (foreign-procedure "cb_init" () void))
(define register-callback
  (foreign-procedure "register_callback" (char void*) void))
(define event-loop
  (foreign-procedure __collect_safe "event_loop" () void))
```

A callback for selected characters can then be defined.

```
(define callback
  (lambda (p)
    (let ([code (foreign-callable __collect_safe p (char) void)])
      (lock-object code)
      (foreign-callable-entry-point code))))
(define ouch
  (callback
    (lambda (c)
      (printf "Ouch! Hit by '~c'~%" c))))
(define rats
  (callback
    (lambda (c)
      (printf "Rats! Received '~c'~%" c))))

(cb-init)
(register-callback #\a ouch)
(register-callback #\c rats)
(register-callback #\e ouch)
```

This sets up the following interaction.

```
> (event-loop)
a
Ouch! Hit by 'a'
b
c
Rats! Received 'c'
d
e
Ouch! Hit by 'e'
```

The `__collect_safe` declarations in this example ensure that other threads can continue working while `event-loop` blocks waiting for input. A more well-behaved version of the example would save each code object returned by `foreign-callable` and unlock it when it is no longer registered as a callback.

`(foreign-callable-entry-point` *code*`)` **procedure**
returns: the address of the foreign-callable entry point in *code*
libraries: `(chezscheme)`

code should be a code object produced by `foreign-callable`.

`(foreign-callable-code-object` *address*`)` **procedure**
returns: the code object corresponding to the foreign-callable entry point *address*
libraries: `(chezscheme)`

address must be an exact integer and should be the address of the entry point of a code object produced by `foreign-callable`.

4.4. Continuations and Foreign Calls

`foreign-callable` and `foreign-procedure` allow arbitrary nesting of foreign and Scheme calls. Because other languages do not support the fully general first-class continuations of Scheme, the interaction between continuations and nested calls among Scheme and foreign procedures is problematic. *Chez Scheme* handles this interaction in a general manner by trapping attempts to return to *stale* foreign contexts rather than by restricting the use of continuations directly. A foreign context is a foreign frame and return point corresponding to a particular call from a foreign language, e.g., C, into Scheme. A foreign context becomes stale after a normal return to the context or after a return to some other foreign context beneath it on the control stack.

As a result of this treatment, Scheme continuations may be used to throw control either upwards or downwards logically through any mix of Scheme and foreign frames. Furthermore, until some return to a foreign context is actually performed, all return points remain valid. In particular, this means that programs that use continuations exclusively for nonlocal exits never attempt to return to a stale foreign context. (Nonlocal exits themselves are no problem and are implemented by the C library function `longjmp` or the equivalent.) Programs that use continuations more generally also function properly as long as they never actually return to a stale foreign context, even if control logically moves past stale foreign contexts via invocation of continuations.

One implication of this mechanism is that the C stack pointer is not automatically restored to its base value when a continuation is used on the Scheme side to perform a nonlocal exit. If the program continues to run after the nonlocal exit, any further build-up of the C stack will add to the existing build up, which might result in a C stack overflow. To avoid this situation, a program can arrange to set up a single C call frame before obtaining the continuation and return to the C frame after the nonlocal exit. The procedure `with-exit-proc` below arranges to do this without involving any C code.

```
(define with-exit-proc
  (lambda (p)
    (define th (lambda () (call/cc p)))
    (define-ftype ->ptr (function () ptr))
    (let ([fptr (make-ftype-pointer ->ptr th)])
      (let ([v ((ftype-ref ->ptr () fptr))])
        (unlock-object
          (foreign-callable-code-object
            (ftype-pointer-address fptr)))
        v))))
```

`with-exit-proc` behaves like `call/cc` except it resets the C stack when the continuation is invoked. To do this, it creates an ftype-pointer representing a foreign-callable entry point for `th` and creates a Scheme-callable procedure for that entry point. This creates a wrapper for `th` that involves a C call. When a call to the wrapper returns, either by explicit invocation of the continuation passed to `p` or by a normal return from `p`, the C stack is reset to its original value.

4.5. Foreign Data

The procedures described in this section directly create and manipulate foreign data, i.e., data that resides outside of the Scheme heap. With the exception of `foreign-alloc` and `foreign-sizeof`, these procedures are inherently unsafe in the sense that they do not (and cannot) check the validity of the addresses they are passed. Improper use of these procedures can result in invalid memory references, corrupted data, or system crashes.

This section also describes a higher-level syntactic mechanism for manipulating foreign data, including foreign structures, unions, arrays, and bit fields. The syntactic interface is safer than the procedural interface but must still assume that the addresses it's given are appropriate for the types of object being manipulated.

`(foreign-alloc` *n*`)` **procedure**
returns: the address of a freshly allocated block of foreign data *n* bytes long
libraries: `(chezscheme)`

n must be a positive fixnum. The returned value is an exact integer and is guaranteed to be properly aligned for any type of value according to the requirements of the underlying hardware. An exception is raised with condition type `&assertion` if the block of foreign data cannot be allocated.

`(foreign-free` *address*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedure frees the block of storage to which *address* points. *address* must be an exact integer in the range -2^{w-1} through 2^w-1, where w is the width in bits of a pointer, e.g., 64

for a 64-bit machine. It should be an address returned by an earlier call to `foreign-alloc` and not subsequently passed to `foreign-free`.

`(foreign-ref` *type address offset*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

`foreign-ref` extracts the value of type *type offset* bytes into the block of foreign data addressed by *address*.

type must be a symbol identifying the type of value to be extracted. The following types have machine-dependent sizes and correspond to the like-named C types:

- `short`,
- `unsigned-short`,
- `int`,
- `unsigned`,
- `unsigned-int`,
- `long`,
- `unsigned-long`,
- `long-long`,
- `unsigned-long-long`,
- `ptrdiff_t`,
- `size_t`,
- `ssize_t`,
- `char`,
- `wchar_t`,
- `float`,
- `double`, and
- `void*`.

The types `long-long` and `unsigned-long-long` correspond to the C types `long long` and `unsigned long long`. A value of type `char` is referenced as a single byte and converted (as if via `integer->char`) into a Scheme character. A value of type `wchar_t` is converted (as if via `integer->char`) into a Scheme character. The value must be a valid Unicode scalar value.

`wchar` is an alias for `wchar_t`.

Several additional machine-dependent types are recognized:

- `iptr`,
- `uptr`,
- `fixnum`, and

- `boolean`.

`uptr` is equivalent to `void*`; both are treated as unsigned integers the size of a pointer. `iptr` is treated as a signed integer the size of a pointer. `fixnum` is treated as an `iptr`, but with a range limited to the fixnum range. `boolean` is treated as an `int`, with zero converted to the Scheme value `#f` and all other values converted to `#t`.

Finally, several fixed-sized types are also supported:

- `integer-8`,
- `unsigned-8`,
- `integer-16`,
- `unsigned-16`,
- `integer-32`,
- `unsigned-32`,
- `integer-64`,
- `unsigned-64`,
- `single-float`, and
- `double-float`.

address must be an exact integer in the range -2^{w-1} through $2^w - 1$, where w is the width in bits of a pointer, e.g., 64 for a 64-bit machine. *offset* must be an exact fixnum. The sum of *address* and *offset* should address a readable block of memory large enough to hold a value of type *type*, within a block of storage previously returned by `foreign-alloc` and not subsequently freed by `foreign-free` or within a block of storage obtained via some other mechanism, e.g., a foreign call. For multiple-byte values, the native endianness of the machine is assumed.

`(foreign-set!` *type address offset value*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

`foreign-set!` stores a representation of *value* as type *type* *offset* bytes into the block of foreign data addressed by *address*.

type must be a symbol identifying the type of value to be stored, one of those listed in the description of `foreign-ref` above. Scheme characters are converted to type `char` or `wchar_t` as if via `char->integer`. For type `boolean`, Scheme `#f` is converted to the `int` 0, and any other Scheme object is converted to 1.

address must be an exact integer in the range -2^{w-1} through $2^w - 1$, where w is the width in bits of a pointer, e.g., 64 for a 64-bit machine. *offset* must be an exact fixnum. The sum of *address* and *offset* should address a writable block of memory large enough to hold a value of type *type*, within a block of storage previously returned by `foreign-alloc` and not subsequently freed by `foreign-free` or within a block of storage obtained via some other mechanism, e.g., a foreign call. *value* must be an appropriate value for *type*, e.g., a floating-point number for the float types or an exact integer within the appropriate range

for the integer types. For multiple-byte values, the native endianness of the machine is assumed.

`(foreign-sizeof` *type*`)` **procedure**
returns: the size in bytes of *type*
libraries: `(chezscheme)`

type must be one of the symbols listed in the description of `foreign-ref` above.

`(define-ftype` *ftype-name ftype*`)` **syntax**
`(define-ftype (`*ftype-name ftype*`) ...)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

A `define-ftype` form is a definition and can appear anywhere other definitions can appear. It establishes one or more foreign-type (ftype) bindings for the identifier *ftype-name* or identifiers *ftype-name* `...` to the foreign type represented *ftype* or the foreign types represented by *ftype* `....` Each *ftype-name* can be used to access foreign objects with the declared shape, and each can be used in the formation of other ftypes.

An *ftype* must take one of the following forms:

ftype-name
`(struct (`*field-name ftype*`) ...)`
`(union (`*field-name ftype*`) ...)`
`(array` *length ftype*`)`
`(*` *ftype*`)`
`(bits (`*field-name signedness bits*`) ...)`
`(function` *conv* `...` `(`*ftype* `...)` *ftype*`)`
`(packed` *ftype*`)`
`(unpacked` *ftype*`)`
`(endian` *endianness ftype*`)`

where *length* is an exact nonnegative integer, *bits* is an exact positive integer, *field-name* is an identifier, *conv* is `#f` or a string naming a valid convention as described on page 4.2, signedness is either `signed` or `unsigned`, and endianness is one of `native`, `big`, or `little`.

A restriction not reflected above is that `function` ftypes cannot be used as the types of field names or array elements. That is, function ftypes are valid only at the top level of an ftype, e.g,:

```
(define-ftype bvcopy_t (function (u8* u8* size_t) void))
```

or as the immediate sub-type of a pointer (`*`) ftype, as in the following definitions, which are equivalent assuming the definition of `bvcopy_t` above.

```
(define-ftype A
  (struct
    [x int]
    [f (* (function (u8* u8* size_t) void))]))
```

```
(define-ftype A
  (struct
    [x int]
    [f (* bvcopy_t)]))
```

That is, a function cannot be embedded within a struct, union, or array, but a pointer to a function can be so embedded.

The following definitions establish ftype bindings for `F`, `A`, and `E`.

```
(define-ftype F (function (wchar_t int) int))

(define-ftype A (array 10 wchar_t))

(define-ftype E
  (struct
    [a int]
    [b double]
    [c (array 25
         (struct
           [a short]
           [_ long]
           [b A]))]
    [d (endian big
         (union
           [v1 unsigned-32]
           [v2 (bits
                 [hi unsigned 12]
                 [lo unsigned 20])]))]
    [e (* A)]
    [f (* F)]))
```

The ftype `F` describes the type of a foreign function that takes two arguments, a wide character and an integer, and returns an integer. The ftype `A` is simply an array of 10 `wchar_t` values, and its size will be 10 times the size of a single `wchar_t`. The ftype `E` is a structure with five fields: an integer `a`, a double-float `b`, an array `c`, a union `d`, and a pointer `e`. The array `c` is an array of 25 structs, each of which contains a short integer, a long integer, and a `A` array. The size of the `c` array will be 25 times the size of a single `A` array, plus 25 times the space needed to store each of the short and long integers. The union `d` is either a 32-bit unsigned integer or a 32-bit unsigned integer split into high (12 bits) and low (20 bits) components. The fields of a union overlap so that writing to one effectively overlaps the other. Thus, one can use the `d` union type to split apart an unsigned integer by writing the integer into `v1` and reading the pieces from `hi` and `lo`. The pointer `e` points to an `A` array; it is not itself an array, and its size is just the size of a single pointer. Similarly, `f` points to a function, and its size is also that of a single pointer.

An underscore (`_`) can be used as the field name for one or more fields of a `struct`, `union`, or `bits` ftype. Such fields are included in the layout but are considered unnamed and cannot be accessed via the ftype operators described below. Thus, in the example above, the `long` field within the `c` array is inaccessible.

Non-underscore field names are handled symbolically, i.e., they are treated as symbols rather than identifiers. Each symbol must be unique (as a symbol) with respect to the other field names within a single `struct`, `union`, or `bits` ftype but need not be unique with respect to field names in other `struct`, `union`, or `bits` ftypes within the same ftype.

Each *ftype-name* in an *ftype* must either (a) have been defined previously by `define-ftype`, (b) be defined by the current `define-ftype`, or (c) be a base-type name, i.e., one of the type names supported by `foreign-ref` and `foreign-set!`. In case (b), any reference within one *ftype* to the *ftype-name* of one of the earlier bindings is permissible, but a reference to the *ftype-name* of the current or a subsequent binding can appear only within a pointer field.

For example, in:

```
(define-ftype
  [Qlist (struct
           [head int]
           [tail (* Qlist)])])
```

the reference to `Qlist` is permissible since it appears within a pointer field. Similarly, in:

```
(define-ftype
  [Qfrob (struct
           [head int]
           [tail (* Qsnark)])]
  [Qsnark (struct
            [head int]
            [xtra Qfrob]
            [tail (* Qfrob)])])
```

the mutually recursive references to `Qsnark` and `Qfrob` are permissible. In the following, however:

```
(define-ftype
  [Qfrob (struct
           [head int]
           [xtra Qfrob]
           [tail (* Qsnark)])]
  [Qsnark (struct
            [head int]
            [tail (* Qfrob)])])
```

the reference to `Qfrob` within the *ftype* for `Qfrob` is invalid, and in:

```
(define-ftype
  [Qfrob (struct
           [head int]
           [xtra Qsnark]
           [tail (* Qsnark)])]
  [Qsnark (struct
            [head int]
```

```
            [tail (* Qfrob)])])
```

the reference to `Qsnark` is similarly invalid.

By default, padding is inserted where appropriate to maintain proper alignment of multiple-byte scalar values in an attempt to mirror the target machine's C struct layout conventions, where such layouts are adequately documented. For packed ftypes (ftypes wrapped in a `packed` form with no closer enclosing `unpacked` form), this padding is not inserted.

Multiple-byte scalar values are stored in memory using the target machine's native "endianness," e.g., `little` on X86 and X86_64-based platforms and `big` on Sparc-based platforms. Big-endian or little-endian representation can be forced via the `endian` ftype with a `big` or `little` *endianness* specifier. The `native` specifier can be used to force a return back to `native` representation. Each `endian` form affects only ftypes nested syntactically within it and not nested within a closer `endian` form.

The total size n of the fields within an ftype bits form must be 8, 16, 24, 32, 40, 48, 56, or 64. padding must be added manually if needed. In little-endian representation, the first field occupies the low-order bits of the containing 8, 16, 24, 32, 40, 48, 56, or 64-bit word, with each subsequent field just above the preceding field. In big-endian representation, the first field occupies the high-order bits, with each subsequent field just below the preceding field.

Two ftypes are considered equivalent only if defined by the same `ftype` binding. If two ftype definitions look identical but appear in two parts of the same program, the ftypes are not identical, and attempts to access one using the name of the other via the operators described below will fail with a run-time exception.

Array bounds must always be constant. If an array's length cannot be known until run time, the array can be placed at the end of the ftype (and any containing ftype) and declared to have size zero, as illustrated by the example below.

```
(define-ftype Vec
  (struct
    [len int]
    [data (array 0 double)]))
(define make-Vec
  (lambda (n)
    (let ([fptr (make-ftype-pointer Vec
                  (foreign-alloc
                    (+ (ftype-sizeof Vec)
                       (* (ftype-sizeof double) n))))])
      (ftype-set! Vec (len) fptr n)
      fptr)))
(define x (make-Vec 100))
(/ (- (ftype-pointer-address (ftype-&ref Vec (data 10) x))
      (ftype-pointer-address x)                              ⇒  10
      (ftype-sizeof int))
   (ftype-sizeof double))
(foreign-free (ftype-pointer-address x))
```

No array bounds checks are performed for zero-length arrays. Only one variable-sized array

can appear in a single foreign object, but one can work around this by treating the object as multiple individual objects.

To avoid specifying the constant length of an array in more than one place, a macro that binds both a variable to the size as well as an ftype name to the ftype can be used. For example,

```
(define-syntax define-array
  (syntax-rules ()
    [(_ array-name type size-name size)
     (begin
       (define size-name size)
       (define-ftype array-name
         (array size type)))]))
(define-array A int A-size 100)
A-size  ⇒  100
(ftype-pointer-ftype
  (make-ftype-pointer A
    (foreign-alloc (ftype-sizeof A))))  ⇒  (array 100 int)
```

This technique can be used to define arbitrary ftypes with arbitrary numbers of array fields.

A struct ftype is an implicit subtype of the type of the first field of the struct. Similarly, an array ftype is an implicit subtype of the type of its elements. Thus, the struct or array extends the type of first field or element with additional fields or elements. This allows an instance of the struct or array to be treated as an instance of the type of its first field or element, without the need to use `ftype-&ref` to allocate a new pointer to the field or element.

`(ftype-sizeof` *ftype-name*`)` **syntax**
returns: the size in bytes of the ftype identified by *ftype-name*
libraries: `(chezscheme)`

The size includes the sizes of any ftypes directly embedded within the identified ftype but excludes those indirectly embedded via a pointer ftype. In the latter case, the size of the pointer is included.

ftype-name must not be defined as a function ftype, since the size of a function cannot generally be determined.

```
(define-ftype B
  (struct
    [b1 integer-32]
    [b2 (array 10 integer-32)]))
(ftype-sizeof B)  ⇒  44

(define-ftype C (* B))
(ftype-sizeof C)  ⇒  4  ; on 32-bit machines
(ftype-sizeof C)  ⇒  8  ; on 64-bit machines
```

```
(define-ftype BB
  (struct
    [bb1 B]
    [bb2 (* B)]))
(- (ftype-sizeof BB) (ftype-sizeof void*))  ⇒  44
```

`(make-ftype-pointer` *ftype-name expr*`)` **syntax**
returns: an ftype-pointer object
libraries: `(chezscheme)`

If *ftype-name* does not describe a function ftype, *expr* must evaluate to an *address* represented as an exact integer in the appropriate range for the target machine.

The ftype-pointer object returned by this procedure encapsulates the address and is tagged with a representation of the type identified by *ftype-name* to enable various forms of checking to be done by the access routines described below.

```
(make-ftype-pointer E #x80000000)  ⇒  #<ftype-pointer #x80000000>
```

The address will not typically be a constant, as shown. Instead, it might instead come from a call to `foreign-alloc`, e.g.:

```
(make-ftype-pointer E (foreign-alloc (ftype-sizeof E)))
```

It might also come from source outside of Scheme such as from a C routine called from Scheme via the foreign-procedure interface.

If *ftype-name* describes a function ftype, *expr* must evaluate to an address, procedure, or string. If it evaluates to address, the call behaves like any other call to `make-ftype-pointer` with an address argument.

If it evaluates to a procedure, a foreign-callable code object is created for the procedure, as if via `foreign-callable` (Section 4.3). The address encapsulated in the resulting ftype-pointer object is the address of the procedure's entry point.

```
(define fact
  (lambda (n)
    (if (= n 0) 1 (fact (- n 1)))))
(define-ftype fact_t (function (int) int))
(define fact-fptr (make-ftype-pointer fact_t fact))
```

The resulting ftype pointer can be passed to a C routine, if the argument is declared to be a pointer to the same ftype, and the C routine can invoke the function pointer it receives as it would any other function pointer. Thus, `make-ftype-pointer` with a function ftype is an alternative to `foreign-callable` for creating C-callable wrappers for Scheme procedures.

Since all Scheme objects, including code objects, can be relocated or even reclaimed by the garbage collector the foreign-callable code object is automatically locked, as if via `lock-object`, before it is embedded in the ftype pointer. The code object should be unlocked after its last use from C, since locked objects take up space, cause fragmentation, and increase the cost of collection. Since the system cannot determine automatically when

the last use from C occurs, the program must explicitly unlock the code object, which it can do by extracting the address from the ftype-pointer converting the address (back) into a code object, and passing it to `unlock-object`:

```
(unlock-object
  (foreign-callable-code-object
    (ftype-pointer-address fact-fptr)))
```

Once unlocked, the ftype pointer should not be used again, unless it is relocked, e.g., via:

```
(lock-object
  (foreign-callable-code-object
    (ftype-pointer-address fact-fptr)))
```

A program can determine whether an object is already locked via the `locked-object?` predicate.

A function ftype can be also used with `make-ftype-pointer` to create an ftype-pointer to a C function, either by providing the address of the C function or its name, represented as a string. For example, with the following definition of `bvcopy_t`,

```
(define-ftype bvcopy_t (function (u8* u8* size_t) void))
```

the two definitions of `bvcopy-ftpr` below are equivalent.

```
(define bvcopy-fptr (make-ftype-pointer bvcopy_t "memcpy"))
(define bvcopy-fptr (make-ftype-pointer bvcopy_t (foreign-entry "memcpy")))
```

A library that defines *memcpy* must be loaded first via `load-shared-object`, or `memcpy` must be registered via one of the methods described in Section 4.6.

`(ftype-pointer?` *obj*`)` **syntax**

returns: `#t` if *obj* is an ftype pointer, otherwise `#f`

`(ftype-pointer?` *ftype-name obj*`)` **syntax**

returns: `#t` if *obj* is an *ftype-name*, otherwise `#f`

libraries: `(chezscheme)`

```
(define-ftype Widget1 (struct [x int] [y int]))
(define-ftype Widget2 (struct [w Widget1] [b boolean]))

(define x1 (make-ftype-pointer Widget1 #x80000000))
(define x2 (make-ftype-pointer Widget2 #x80000000))

(ftype-pointer? x1)  ⇒  #t
(ftype-pointer? x2)  ⇒  #t

(ftype-pointer? Widget1 x1)  ⇒  #t
(ftype-pointer? Widget1 x2)  ⇒  #t

(ftype-pointer? Widget2 x1)  ⇒  #f
(ftype-pointer? Widget2 x2)  ⇒  #t
```

```
(ftype-pointer? #x80000000)  ⇒  #f
(ftype-pointer? Widget1 #x80000000)  ⇒  #f
```

(ftype-pointer-address *fptr*) **procedure**
returns: the address encapsulated within *fptr*
libraries: `(chezscheme)`

fptr must be an ftype-pointer object.

```
(define x (make-ftype-pointer E #x80000000))
(ftype-pointer-address x)  ⇒  #x80000000
```

(ftype-pointer=? $fptr_1$ $fptr_2$) **syntax**
returns: `#t` if $fptr_1$ and $fptr_2$ have the same address, otherwise `#f`
libraries: `(chezscheme)`

$fptr_1$ and $fptr_2$ must be ftype-pointer objects.

`ftype-pointer=?` might be defined as follows:

```
(define ftype-pointer=?
  (lambda (fptr1 fptr2)
    (= (ftype-pointer-address fptr1) (ftype-pointer-address fptr2))))
```

It is, however, guaranteed not to allocate bignums for the addresses even if the addresses do not fit in fixnum range.

(ftype-pointer-null? *fptr*) **syntax**
returns: `#t` if the address of *fptr* is 0, otherwise `#f`
libraries: `(chezscheme)`

fptr must be an ftype-pointer object.

`ftype-pointer-null?` might be defined as follows:

```
(define ftype-pointer-null?
  (lambda (fptr)
    (= (ftype-pointer-address fptr) 0)))
```

It is, however, guaranteed not to allocate a bignum for the address even if the address does not fit in fixnum range.

(ftype-&ref *ftype-name* (*a* ...) *fptr-expr*) **syntax**
(ftype-&ref *ftype-name* (*a* ...) *fptr-expr index*) **syntax**
returns: an ftype-pointer object
libraries: `(chezscheme)`

The ftype-pointer object returned by `ftype-&ref` encapsulates the address of some ob-

ject embedded directly or indirectly within the foreign object pointed to by the value of *fptr-expr*, offset by *index*, if present. The value of *fptr-expr* must be an ftype pointer (fptr) of the ftype identified by *ftype-name*, and *index* must either be the identifier `*` or evaluate to a fixnum, possibly negative. The index is automatically scaled by the size of the ftype identified by *ftype-name*, which allows the fptr to be treated as an array of *ftype-name* objects and *index* as an index into that array. An index of `*` or 0 is the same as no index.

The sequence of accessors *a* `...` must specify a valid path through the identified ftype. For `struct`, `union`, and `bits` ftypes, an accessor must be a valid field name for the ftype, while for pointer and array ftypes, an accessor must be the identifier `*` or evaluate to a fixnum index. For array ftypes, an index must be nonnegative, and for array ftypes with nonzero length, an index must also be less than the length.

The examples below assume the definitions of `B` and `BB` shown above in the description of `ftype-sizeof`. Fixed addresses are shown for illustrative purposes and are assumed to be valid, although addresses are generally determined at run time via `foreign-alloc` or some other mechanism.

```
(define x (make-ftype-pointer B #x80000000))
(ftype-&ref B () x)  ⇒  #<ftype-pointer #x80000000>
(let ([idx 1])              ⇒  #<ftype-pointer #x8000002C>
  (ftype-&ref B () x idx))
(let ([idx -1])             ⇒  #<ftype-pointer #x7FFFFFD4>
  (ftype-&ref B () x idx))
(ftype-&ref B (b1) x)  ⇒  #<ftype-pointer #x80000000>
(ftype-&ref B (b2) x)  ⇒  #<ftype-pointer #x80000004>
(ftype-&ref B (b2 5) x)  ⇒  #<ftype-pointer #x80000018>
(let ([n 5]) (ftype-&ref B (b2 n) x))  ⇒  #<ftype-pointer #x80000018>

(ftype-&ref B (b1 b2) x)  ⇒  syntax error
(ftype-&ref B (b2 15) x)  ⇒  run-time exception

(define y (make-ftype-pointer BB #x90000000))
(ftype-set! BB (bb2) y x)
(ftype-&ref BB (bb1 b2) y)  ⇒  #<ftype-pointer #x90000004>
(ftype-&ref BB (bb2 * b2) y)  ⇒  #<ftype-pointer #x80000004>
(let ([idx 1])                    ⇒  #<ftype-pointer #x80000030>
  (ftype-&ref BB (bb2 idx b2) y))
```

With no accessors and no index, as in the first use of `ftype-&ref` above, the returned `ftype-pointer` might be `eq?` to the input. Otherwise, the `ftype-pointer` is freshly allocated.

(**ftype-set!** *ftype-name* (*a* ...) *fptr-expr val-expr*) **syntax**
(**ftype-set!** *ftype-name* (*a* ...) *fptr-expr index val-expr*) **syntax**
returns: unspecified
(**ftype-ref** *ftype-name* (*a* ...) *fptr-expr*) **syntax**
(**ftype-ref** *ftype-name* (*a* ...) *fptr-expr index*) **syntax**
returns: an ftype-pointer object
libraries: (**chezscheme**)

These forms are used to store values into or retrieve values from the object pointed to by the value of *fptr-expr*, offset by *index*, if present. The value of *fptr-expr* must be an ftype pointer (fptr) of the ftype identified by *ftype-name*, and *index* must either be the identifier `*` or evaluate to a fixnum, possibly negative. The index is automatically scaled by the size of the ftype identified by *ftype-name*, which allows the fptr to be treated as an array of *ftype-name* objects and *index* as an index into that array. An index of `*` or 0 is the same as no index.

The sequence of accessors *a* ... must specify a valid path through the identified ftype. For `struct`, `union`, and `bits` ftypes, an accessor must be a valid field name for the ftype, while for pointer and array ftypes, an accessor must be the identifier `*` or evaluate to a fixnum index. For array ftypes, an index must be nonnegative, and for array ftypes with nonzero length, an index must also be less than the length. The field or element specified by the sequence of accessors must be a scalar field, e.g., a pointer field or a field containing a base type such as an `int`, `char`, or `double`.

For `ftype-set!`, *val-expr* must evaluate to a value of the appropriate type for the specified field, e.g., an ftype pointer of the appropriate type or an appropriate base-type value.

For both signed and unsigned integer fields, values in the range -2^{w-1} through $2^w - 1$ are accepted, where w is the width in bits of the integer field. For signed integer fields, values in the range 2^{w-1} through $2^w - 1$ are treated as two's complement representations of the corresponding negative numbers. For unsigned integer fields, values in the range -2^{w-1} through -1 are similarly treated as two's complement representations of the corresponding positive numbers.

`char` and `wchar_t` (`wchar`) field values are converted from (`ftype-set!`) or to (`ftype-ref`) Scheme characters, as if with `char->integer` and `integer->char`. Characters stored by `ftype-set!` into a `char` field must have Unicode scalar values in the range 0 through 255. Under Windows and any other system where `wchar_t` (`wchar`) is a 16-bit value, characters stored by `ftype-set!` into a `whar_t` (`wchar`) field must have Unicode scalar values in the range 0 through $2^{16} - 1$. On systems where `wchar_t` is a 32-bit value, any character can be stored in a `wchar_t` (`wchar`) field.

The examples below assume that `B` and `C` have been defined as shown in the description of `ftype-sizeof` above.

```
(define b
  (make-ftype-pointer B
    (foreign-alloc
      (* (ftype-sizeof B) 3))))
(define c
```

```
  (make-ftype-pointer C
    (foreign-alloc (ftype-sizeof C))))

(ftype-set! B (b1) b 5)
(ftype-set! B (b1) b 1 6)
(ftype-set! B (b1) c 5)  ⇒  exception: ftype mismatch
(ftype-set! B (b2) b 0)  ⇒  exception: not a scalar
(ftype-set! B (b2 -1) b 0)  ⇒  exception: invalid index
(ftype-set! B (b2 0) b 50)
(ftype-set! B (b2 4) b 55)
(ftype-set! B (b2 10) b 55)  ⇒  exception: invalid index

(ftype-set! C () c (ftype-&ref B () b 1))

(= (ftype-pointer-address (ftype-ref C () c))       ⇒  #t
   (+ (ftype-pointer-address b) (ftype-sizeof B)))
(= (ftype-pointer-address (ftype-&ref C (*) c))  ⇒  #t
   (+ (ftype-pointer-address b) (ftype-sizeof B)))
(= (ftype-pointer-address (ftype-&ref C (-1) c))  ⇒  #t
   (ftype-pointer-address b))

(ftype-ref C (-1 b1) c)  ⇒  5
(ftype-ref C (* b1) c)  ⇒  6
(ftype-ref C (-1 b2 0) c)  ⇒  50
(let ([i 4]) (ftype-ref C (-1 b2 i) c))  ⇒  55

(ftype-set! C (-1 b2 0) c 75)
(ftype-ref B (b2 0) b)  ⇒  75
(foreign-free (ftype-pointer-address c))
(foreign-free (ftype-pointer-address b))
```

A function ftype pointer can be converted into a Scheme-callable procedure via `ftype-ref`. Assuming that a library defining *memcpy* has been loaded via `load-shared-object` or `memcpy` has been registered via one of the methods described in Section 4.6, A Scheme-callable `memcpy` can be defined as follows.

```
(define-ftype bvcopy_t (function (u8* u8* size_t) void))
(define bvcopy-fptr (make-ftype-pointer bvcopy_t "memcpy"))
(define bvcopy (ftype-ref bvcopy_t () bvcopy-fptr))

(define bv1 (make-bytevector 8 0))
(define bv2 (make-bytevector 8 57))
bv1  ⇒  #vu8(0 0 0 0 0 0 0 0)
bv2  ⇒  #vu8(57 57 57 57 57 57 57 57)
(bvcopy bv1 bv2 5)
bv1  ⇒  #vu8(57 57 57 57 57 0 0 0)
```

An ftype pointer can also be obtained as a return value from a C function declared to return a pointer to a function ftype.

Thus, `ftype-ref` with a function ftype is an alternative to `foreign-procedure` (Section 4.2) for creating Scheme-callable wrappers for C functions.

`(ftype-pointer-ftype` *fptr*`)` **procedure**
returns: *fptr*'s ftype, represented as an s-expression
libraries: `(chezscheme)`

fptr must be an ftype-pointer object.

```
(define-ftype Q0
  (struct
    [x int]
    [y int]))
(define-ftype Q1
  (struct
    [x double]
    [y char]
    [z (endian big
         (bits
           [_ unsigned 3]
           [a unsigned 9]
           [b unsigned 4]))]
    [w (* Q0)]))
(define q1 (make-ftype-pointer Q1 0))
(ftype-pointer-ftype q1)  ⇒  (struct
                               [x double]
                               [y char]
                               [z (endian big
                                    (bits
                                      [_ unsigned 3]
                                      [a unsigned 9]
                                      [b unsigned 4]))]
                               [w (* Q0)])
```

`(ftype-pointer->sexpr` *fptr*`)` **procedure**
returns: an s-expression representation of the object to which *fptr* points
libraries: `(chezscheme)`

fptr must be an ftype-pointer object.

For each unnamed field, i.e., each whose field name is an underscore, the corresponding field value in the resulting s-expression is also an underscore. Similarly, if a field is inaccessible, i.e., if its address is invalid, the value is the symbol `invalid`.

```
(define-ftype Frob
  (struct
    [p boolean]
    [q char]))
(define-ftype Snurk
  (struct
    [a Frob]
    [b (* Frob)]
    [c (* Frob)]
```

```
    [d (bits
         [_ unsigned 15]
         [dx signed 17])]
    [e (array 5 double)]))
(define x
  (make-ftype-pointer Snurk
    (foreign-alloc (ftype-sizeof Snurk))))
(ftype-set! Snurk (b) x
  (make-ftype-pointer Frob
    (foreign-alloc (ftype-sizeof Frob))))
(ftype-set! Snurk (c) x
  (make-ftype-pointer Frob 0))
(ftype-set! Snurk (a p) x #t)
(ftype-set! Snurk (a q) x #\A)
(ftype-set! Snurk (b * p) x #f)
(ftype-set! Snurk (b * q) x #\B)
(ftype-set! Snurk (d dx) x -2500)
(do ([i 0 (fx+ i 1)])
    ((fx= i 5))
  (ftype-set! Snurk (e i) x (+ (* i 5.0) 3.0)))
(ftype-pointer->sexpr x)  ⇒  (struct
                               [a (struct [p #t] [q #\A])]
                               [b (* (struct [p #f] [q #\B]))]
                               [c (* (struct [p invalid] [q invalid]))]
                               [d (bits [_ _] [dx -2500])]
                               [e (array 5 3.0 8.0 13.0 18.0 23.0)])
```

4.6. Providing Access to Foreign Procedures

Access to foreign procedures can be provided in several ways:

- Foreign procedures may be loaded from "shared objects" using `load-shared-object`.
- A new *Chez Scheme* image can be built with additional foreign code linked in. (Consult with the person who installed *Chez Scheme* at your site for details.) These entries are typically registered via `Sforeign_symbol` or `Sregister_symbol`, documented in Section 4.8.
- Additional entries may be dynamically loaded or otherwise obtained by foreign code. These are also typically registered using `Sforeign_symbol` or `Sregister_symbol`.
- The address of an entry, i.e., a function pointer, may be passed into Scheme and used as the value of the entry expression in a foreign-procedure expression. This allows foreign entry points to be used even when they are not registered by name.

`(foreign-entry? entry-name)` **procedure**
returns: `#t` if *entry-name* is an existing foreign procedure entry point, `#f` otherwise
libraries: `(chezscheme)`

entry-name must be a string. `foreign-entry?` may be used to determine if an entry exists for a foreign procedure.

The following examples assume that a library that defines *strlen* has been loaded via `load-shared-object` or that `strlen` has been registered via one of the other methods described in this section.

```
(foreign-entry? "strlen")  ⇒  #t
((foreign-procedure "strlen"
    (string) size_t)
 "hey!")  ⇒  4
```

`(foreign-entry entry-name)` **procedure**
returns: the address of *entry-name* as an exact integer
libraries: `(chezscheme)`

entry-name must be a string naming an existing foreign entry point.

The following examples assume that a library that defines *strlen* has been loaded via `load-shared-object` or that `strlen` has been registered via one of the other methods described in this section.

```
(let ([addr (foreign-entry "strlen")])
  (and (integer? addr) (exact? addr)))  ⇒  #t
```

```
(define-ftype strlen-type (function (string) size_t))
(define strlen
  (ftype-ref strlen-type ()
    (make-ftype-pointer strlen-type "strlen")))
(strlen "hey!")  ⇒  4
```

`(foreign-address-name address)` **procedure**
returns: the entry name corresponding to *address*, if known, otherwise `#f`
libraries: `(chezscheme)`

The following examples assume that a library that defines *strlen* has been loaded via `load-shared-object` or that `strlen` has been registered via one of the other methods described in this section.

```
(foreign-address-name (foreign-entry "strlen"))  ⇒  "strlen"
```

`(load-shared-object` *path*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. `load-shared-object` loads the shared object named by *path*. Shared objects may be system libraries or files created from ordinary C programs. All external symbols in the shared object, along with external symbols available in other shared objects linked with the shared object, are made available as foreign entries.

This procedure is supported for most platforms upon which *Chez Scheme* runs.

If *path* does not begin with a "." or "/", the shared object is searched for in a default set of directories determined by the system.

On most Unix systems, `load-shared-object` is based on the system routine `dlopen`. Under Windows, `load-shared-object` is based on `LoadLibrary`. Refer to the documentation for these routines and for the C compiler and loader for precise rules for locating and building shared objects.

`load-shared-object` can be used to access built-in C library functions, such as `getenv`. The name of the shared object varies from one system to another. On Linux systems:

```
(load-shared-object "libc.so.6")
```

On Solaris, OpenSolaris, FreeBSD, NetBSD, and OpenBSD systems:

```
(load-shared-object "libc.so")
```

On MacOS X systems:

```
(load-shared-object "libc.dylib")
```

On Windows:

```
(load-shared-object "crtdll.dll")
```

Once the C library has been loaded, `getenv` should be available as a foreign entry.

```
(foreign-entry? "getenv")  ⇒  #t
```

An equivalent Scheme procedure may be defined and invoked as follows.

```
(define getenv
  (foreign-procedure "getenv"
    (string)
    string))
(getenv "HOME")  ⇒  "/home/elmer/fudd"
(getenv "home")  ⇒  #f
```

`load-shared-object` can be used to access user-created libraries as well. Suppose the C file `"even.c"` contains

```
int even(n) int n; { return n == 0 || odd(n - 1); }
```

and the C file `"odd.c"` contains

```
int odd(n) int n; { return n != 0 && even(n - 1); }
```

The files must be compiled and linked into a shared object before they can be loaded. How this is done depends upon the host system. On Linux, FreeBSD, OpenBSD, and OpenSolaris systems:

```
(system "cc -fPIC -shared -o evenodd.so even.c odd.c")
```

Depending on the host configuration, the `-m32` or `-m64` option might be needed to specify 32-bit or 64-bit compilation as appropriate.

On MacOS X (Intel or PowerPC) systems:

```
(system "cc -dynamiclib -o evenodd.so even.c odd.c")
```

Depending on the host configuration, the `-m32` or `-m64` option might be needed to specify 32-bit or 64-bit compilation as appropriate.

On 32-bit Sparc Solaris:

```
(system "cc -KPIC -G -o evenodd.so even.c odd.c")
```

On 64-bit Sparc Solaris:

```
(system "cc -xarch=v9 -KPIC -G -o evenodd.so even.c odd.c")
```

On Windows, we build a DLL (dynamic link library) file. In order to make the compiler generate the appropriate entry points, we alter `even.c` to read

```
#ifdef WIN32
#define EXPORT extern __declspec (dllexport)
#else
#define EXPORT extern
#endif

EXPORT int even(n) int n; { return n == 0 || odd(n - 1); }
```

and `odd.c` to read

```
#ifdef WIN32
#define EXPORT extern __declspec (dllexport)
#else
#define EXPORT extern
#endif

EXPORT int odd(n) int n; { return n != 0 && even(n - 1); }
```

We can then build the DLL as follows, giving it the extension ".so" rather than ".dll" for consistency with the other systems.

```
(system "cl -c -DWIN32 even.c")
```

```
(system "cl -c -DWIN32 odd.c")
(system "link -dll -out:evenodd.so even.obj odd.obj")
```

The resulting ".so" file can be loaded into Scheme and `even` and `odd` made available as foreign procedures:

```
(load-shared-object "./evenodd.so")
(let ([odd (foreign-procedure "odd"
             (integer-32) boolean)]
      [even (foreign-procedure "even"
              (integer-32) boolean)])
  (list (even 100) (odd 100)))  ⇒  (#t #f)
```

The filename is given as `"./evenodd.so"` rather than simply `"evenodd.so"`, because some systems look for shared libraries in a standard set of system directories that does not include the current directory.

`(remove-foreign-entry` *entry-name*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`remove-foreign-entry` blocks further access to the entry specified by the string *entry-name*. An exception is raised with condition type `&assertion` if the entry does not exist. Since access previously established by `foreign-procedure` is not affected, `remove-foreign-entry` may be used to clean up after the desired interface to a group of foreign procedures has been established.

`remove-foreign-entry` can be used to remove entries registered using `Sforeign_symbol` and `Sregister_symbol` but not entries created as a result of a call to `load-shared-object`.

4.7. Using Other Foreign Languages

Although the *Chez Scheme* foreign procedure interface is oriented primarily toward procedures defined in C or available in C libraries, it is possible to invoke procedures defined in other languages that follow C calling conventions. One source of difficulty may be the interpretation of names. Since Unix-based C compilers often prepend an underscore to external names, the foreign interface attempts to interpret entry names in a manner consistent with the host C compiler. Occasionally, such as for assembly coded files, this entry name interpretation may not be desired. It can be prevented by prefixing the entry name with an "=" character. For example, after loading an assembly file containing a procedure `"foo"` one might have:

```
(foreign-entry? "foo")  ⇒  #f
(foreign-entry? "=foo")  ⇒  #t
```

4.8. C Library Routines

Additional foreign interface support is provided via a set of C preprocessor macros and C-callable library functions. Some of these routines allow C programs to examine, allocate, and alter Scheme objects. Others permit C functions to call Scheme procedures via a more primitive interface than that defined in Section 4.3. Still others permit the development of custom executable images and use of the Scheme system as a subordinate program within another program, e.g., for use as an extension language.

C code that uses these routines must include the `"scheme.h"` header file distributed with *Chez Scheme* and must be linked (statically or dynamically) with the *Chez Scheme* kernel. The header file contains definitions for the preprocessor macros and `extern` declarations for the library functions. The file is customized to the release of *Chez Scheme* and machine type with which it is distributed; it should be left unmodified to facilitate switching among *Chez Scheme* releases, and the proper version of the header file should always be used with C code compiled for use with a particular version of *Chez Scheme*. The version and machine type are defined in `"scheme.h"` under the names `VERSION` and `MACHINE_TYPE`.

The name of each routine begins with a capital `S`, e.g., `Sfixnump`. Many of the names are simple translations of the names of closely related Scheme procedures, e.g., `Sstring_to_symbol` is the C interface equivalent of `string->symbol`. Most externally visible entries in the *Chez Scheme* executable that are not documented here begin with capital `S` followed by an underscore (`S_`); their use should be avoided.

In addition to the various macros and external declarations given in `scheme.h`, the header file also defines (`typedefs`) several types used in the header file:

- `ptr`: type of a Scheme value,
- `iptr`: a signed integer the same size as a Scheme value, and
- `uptr`: an unsigned integer the same size as a Scheme value.
- `string_char`: type of a single Scheme string element.
- `octet`: type of a single Scheme bytevector element (unsigned char).

These types may vary depending upon the platform, although `ptr` is typically `void *`, `iptr` is typically `long int`, and `uptr` is typically `unsigned long int`.

Under Windows, defining `SCHEME_IMPORT` before including scheme.h causes scheme.h to declare its entry points using `extern declspec (dllimport)` rather than `extern declspec (dllexport)` (the default). Not defining `SCHEME_IMPORT` and instead defining `SCHEME_STATIC` causes scheme.h to declare exports using just `extern`. The static libraries distributed with Chez Scheme are built using `SCHEME_STATIC`.

The remainder of this section describes each of the C interface routines in turn. A declaration for each routine is given in ANSI C function prototype notation to precisely specify the argument and result types. Scheme objects have the C type `ptr`, which is defined in `"scheme.h"`. Where appropriate, C values are accepted as arguments or returned as values in place of Scheme objects.

The preprocessor macros may evaluate their arguments more than once (or not at all), so care should be taken to ensure that this does not cause problems.

Customization. The functions described here are used to initialize the Scheme system, build the Scheme heap, and run the Scheme system from a separate program.

[func] `char * Skernel_version(void)`
[func] `void Sscheme_init(void (*`*abnormal_exit*`)(void))`
[func] `void Sset_verbose(int` *v*`)`
[func] `void Sregister_boot_file(const char *`*name*`)`
[func] `void Sbuild_heap(const char *`*exec*`, void (*`*custom_init*`)(void))`
[func] `void Senable_expeditor(const char *`*history_file*`)`
[func] `void Sretain_static_relocation(void)`
[func] `int Sscheme_start(int` *argc*`, char *`*argv*`[])`
[func] `int Sscheme_script(char *`*scriptfile*`, int` *argc*`, char *`*argv*`[])`
[func] `int Sscheme_program(char *`*programfile*`, int` *argc*`, char *`*argv*`[])`
[func] `void Scompact_heap(void)`
[func] `void Sscheme_deinit(void)`

`Skernel_version` returns a string representing the Scheme version. It should be compared against the value of the VERSION preprocessor macro before any of the initialization functions listed above are used to verify that the correct `"scheme.h"` header file has been used.

`Sscheme_init` causes the Scheme system to initialize its static memory in preparation for boot file registration. The *abnormal_exit* parameter should be a (possibly null) pointer to a C function of no arguments that takes appropriate action if the initialization or subsequent heap-building process fails. If null, the default action is to call `exit(1)`.

`Sset_verbose` sets verbose mode on for nonzero values of *v* and off when *v* is zero. In verbose mode, the system displays a trace of the search process for subsequently registered boot files.

`Sregister_boot_file` searches for the named boot file and register it for loading. The file is opened but not loaded until the heap is built via `Sbuild_heap`. For the first boot file registered only, the system also searches for the boot files upon which the named file depends, either directly or indirectly.

`Sbuild_heap` creates the Scheme heap from the registered boot files. *exec* is assumed to be the name of or path to the executable image and is used when no boot files have been registered as the base name for the boot-file search process. *exec* may be null only if one or more boot files have been registered. *custom_init* must be a (possibly null) pointer to a C function of no arguments; if non-null, it is called before any boot files are loaded.

`Sscheme_start` invokes the interactive startup procedure, i.e., the value of the parameter `scheme-start`, with one Scheme string argument for the first *argc* elements of *argv*, not including `argv[0]`. `Sscheme_script` similarly invokes the script startup procedure, i.e., the value of the parameter `scheme-script`, with one Scheme string argument for *scriptfile* and the first *argc* elements of *argv*, not including `argv[0]`. `Sscheme_program` similarly invokes the program startup procedure, i.e., the value of the parameter `scheme-program`, with one

Scheme string argument for *programfile* and the first *argc* elements of *argv*, not including `argv[0]`.

`Senable_expeditor` enables the expression editor (Section 2.2, Chapter 14), which is disabled by default, and determines the history file from which it restores and to which it saves the history. This procedure must be called after the heap is built, or an error will result. It must also be called before `Sscheme_start` in order to be effective. If the *history_file* argument is the null pointer, the history is not restored or saved. The preprocessor variable `FEATURE_EXPEDITOR` is defined in `scheme.h` if support for the expression editor has been compiled into the system.

`Sretain_static_relocation` causes relocation information to be retained for static generation code objects created by heap compaction for the benefit of `compute-size` and related procedures.

`Scompact_heap` compacts the Scheme heap and places all objects currently in the heap into a *static* generation. Objects in the static generation are never collected. That is, they are never moved during collection and the storage used for them is never reclaimed even if they become inaccessible. `Scompact_heap` is called implicitly after any boot files have been loaded.

`Sscheme_deinit` closes any open files, tears down the Scheme heap, and puts the Scheme system in an uninitialized state.

Predicates. The predicates described here correspond to the similarly named Scheme predicates. A trailing letter `p`, for "predicate," is used in place of the question mark that customarily appears at the end of a Scheme predicate name. Each predicate accepts a single Scheme object and returns a boolean (C integer) value.

[macro] `int Sfixnump(ptr` *obj*`)`
[macro] `int Scharp(ptr` *obj*`)`
[macro] `int Snullp(ptr` *obj*`)`
[macro] `int Seof_objectp(ptr` *obj*`)`
[macro] `int Sbwp_objectp(ptr` *obj*`)`
[macro] `int Sbooleanp(ptr` *obj*`)`
[macro] `int Spairp(ptr` *obj*`)`
[macro] `int Ssymbolp(ptr` *obj*`)`
[macro] `int Sprocedurep(ptr` *obj*`)`
[macro] `int Sflonump(ptr` *obj*`)`
[macro] `int Svectorp(ptr` *obj*`)`
[macro] `int Sbytevectorp(ptr` *obj*`)`
[macro] `int Sfxvectorp(ptr` *obj*`)`
[macro] `int Sstringp(ptr` *obj*`)`
[macro] `int Sbignump(ptr` *obj*`)`
[macro] `int Sboxp(ptr` *obj*`)`
[macro] `int Sinexactnump(ptr` *obj*`)`
[macro] `int Sexactnump(ptr` *obj*`)`
[macro] `int Sratnump(ptr` *obj*`)`
[macro] `int Sinputportp(ptr` *obj*`)`

[macro] `int Soutputportp(ptr` *obj*`)`
[macro] `int Srecordp(ptr` *obj*`)`

Accessors. Some of the accessors described here correspond to similarly named Scheme procedures, while others are unique to this interface. `Sfixnum_value`, `Schar_value`, `Sboolean_value`, and `Sflonum_value` return the C equivalents of the given Scheme value.

[macro] `iptr Sfixnum_value(ptr` *fixnum*`)`
[macro] `uptr Schar_value(ptr` *character*`)`
[macro] `int Sboolean_value(ptr` *obj*`)`
[macro] `double Sflonum_value(ptr` *flonum*`)`

`Sinteger_value` and `Sunsigned_value` are similar to `Sfixnum_value`, except they accept not only fixnum arguments but bignum arguments in the range of C integer or unsigned values. `Sinteger_value` and `Sunsigned_value` accept the same range of Scheme integer values. They differ only in the result type, and so allow differing interpretations of negative and large unsigned values.

[func] `iptr Sinteger_value(ptr` *integer*`)`
[macro] `uptr Sunsigned_value(ptr` *integer*`)`

`Sinteger32_value`, `Sunsigned32_value`, `Sinteger64_value`, and `Sunsigned64_value` accept signed or unsigned Scheme integers in the 32- or 64-bit range and return integers of the appropriate type for the machine type.

[func] `<32-bit int type> Sinteger32_value(ptr` *integer*`)`
[macro] `<32-bit unsigned type> Sunsigned32_value(ptr` *integer*`)`
[func] `<64-bit int type> Sinteger64_value(ptr` *integer*`)`
[macro] `<64-bit unsigned type> Sunsigned64_value(ptr` *integer*`)`

`Scar`, `Scdr`, `Ssymbol_to_string` (corresponding to `symbol->string`), and `Sunbox` are identical to their Scheme counterparts.

[macro] `ptr Scar(ptr` *pair*`)`
[macro] `ptr Scdr(ptr` *pair*`)`
[macro] `ptr Ssymbol_to_string(ptr` *sym*`)`
[macro] `ptr Sunbox(ptr` *box*`)`

`Sstring_length`, `Svector_length`, `Sbytevector_length`, and `Sfxvector_length` each return a C integer representing the length (in elements) of the object.

[macro] `iptr Sstring_length(ptr` *str*`)`
[macro] `iptr Svector_length(ptr` *vec*`)`
[macro] `iptr Sbytevector_length(ptr` *bytevec*`)`
[macro] `iptr Sfxvector_length(ptr` *fxvec*`)`

`Sstring_ref`, `Svector_ref`, `Sbytevector_u8_ref`, and `Sfxvector_ref` correspond to their Scheme counterparts, except that the index arguments are C integers, the return value for

`Sstring_ref` is a C character, and the return value for `Sbytevector_u8_ref` is an octet (unsigned char).

[macro] `char Sstring_ref(ptr` *str*`, iptr` *i*`)`
[macro] `ptr Svector_ref(ptr` *vec*`, iptr` *i*`)`
[macro] `octet Sbytevector_u8_ref(ptr` *fxvec*`, iptr` *i*`)`
[macro] `ptr Sfxvector_ref(ptr` *fxvec*`, iptr` *i*`)`

A Scheme bytevector is represented as a length field followed by a sequence of octets (unsignec chars). `Sbytevector_data` returns a pointer to the start of the sequence of octets. Extreme care should be taken to stop dereferencing the pointer returned by `Sbytevector_data` or to lock the bytevector into memory (see `Slock_object` below) before any Scheme code is executed, whether by calling into Scheme or returning to a Scheme caller. The storage manager may otherwise relocate or discard the object into which the pointer points and may copy other data over the object.

[macro] `octet * Sbytevector_data(ptr` *bytevec*`)`

Mutators. Changes to mutable objects that contain pointers, such as pairs and vectors, must be tracked on behalf of the storage manager, as described in one of the references [13]. The operations described here perform this tracking automatically where necessary.

[func] `void Sset_box(ptr` *box*`, ptr` *obj*`)`
[func] `void Sset_car(ptr` *pair*`, ptr` *obj*`)`
[func] `void Sset_cdr(ptr` *pair*`, ptr` *obj*`)`
[macro] `void Sstring_set(ptr` *str*`, iptr` *i*`, char` *c*`)`
[func] `void Svector_set(ptr` *vec*`, iptr` *i*`, ptr` *obj*`)`
[macro] `void Sbytevector_u8_set(ptr` *bytevec*`, iptr` *i*`, octet` *n*`)`
[macro] `void Sfxvector_set(ptr` *fxvec*`, iptr` *i*`, ptr` *fixnum*`)`

Some Scheme objects, such as procedures and numbers, are not mutable, so no operators are provided for altering the contents of those objects.

Constructors. The constructors described here create Scheme objects. Some objects, such as fixnums and the empty list, are represented as immediate values that do not require any heap allocation; others, such as pairs and vectors, are represented as pointers to heap allocated objects.

`Snil`, `Strue`, `Sfalse`, `Sbwp_object`, `Seof_object`, and `Svoid` construct constant immediate values representing the empty list (`()`), the boolean values (`#t` and `#f`), the broken-weak-pointer object (`#!bwp`), the eof object (`#!eof`), and the void object.

[macro] `ptr Snil`
[macro] `ptr Strue`
[macro] `ptr Sfalse`
[macro] `ptr Sbwp_object`
[macro] `ptr Seof_object`
[macro] `ptr Svoid`

Fixnums, characters, booleans, flonums, and strings may be created from their C equivalents.

[macro] `ptr Sfixnum(iptr` *n*`)`
[macro] `ptr Schar(char` *c*`)`
[macro] `ptr Sboolean(int` *b*`)`
[func] `ptr Sflonum(double x)`
[func] `ptr Sstring(const char *`*s*`)`
[func] `ptr Sstring_of_length(const char *`*s*`, iptr` *n*`)`
[func] `ptr Sstring_utf8(const char *`*s*`, iptr` *n*`)`
;

`Sstring` creates a Scheme copy of the C string *s*, while `Sstring_of_length` creates a Scheme string of length *n* and copies the first *n* bytes from *s* into the new Scheme string.

If the C string is encoded in UTF-8, use `Sstring_utf8` instead. Specify the number of bytes to convert as *n* or use −1 to convert until the null terminator.

It is possible to determine whether a C integer is within fixnum range by comparing the fixnum value of a fixnum created from a C integer with the C integer:

```
#define fixnum_rangep(x) (Sfixnum_value(Sfixnum(x)) == x)
```

`Sinteger` and `Sunsigned` may be used to create Scheme integers whether they are in fixnum range or not.

[func] `ptr Sinteger(iptr` *n*`)`
[func] `ptr Sunsigned(uptr` *n*`)`

`Sinteger` and `Sunsigned` differ in their treatment of negative C integer values as well as C unsigned integer values that would appear negative if cast to integers. `Sinteger` converts such values into negative Scheme values, whereas `Sunsigned` converts such values into the appropriate positive Scheme values. For example, assuming a 32-bit, two's complement representation for `iptrs`, `Sinteger(-1)` and `Sunsigned((iptr)0xffffffff)` both evaluate to the Scheme integer `-1`, whereas `Sunsigned(0xffffffff)` and `Sunsigned((uptr)-1)` both evaluate to the Scheme integer `#xffffffff` (`4294967295`).

Whichever routine is used, `Sinteger_value` and `Sunsigned_value` always reproduce the corresponding C input value, thus the following are all equivalent to *x* if *x* is an iptr.

```
Sinteger_value(Sinteger(x))
(iptr)Sunsigned_value(Sinteger(x))
Sinteger_value(Sunsigned((uptr)x))
(iptr)Sunsigned_value(Sunsigned((uptr)x))
```

Similarly, the following are all equivalent to *x* if *x* is a uptr.

```
(uptr)Sinteger_value(Sinteger((iptr)x))
Sunsigned_value(Sinteger((iptr)x))
(uptr)Sinteger_value(Sunsigned(x))
Sunsigned_value(Sunsigned(x))
```

`Sinteger32`, `Sunsigned32`, `Sinteger64`, and `Sunsigned64` are like the generic equivalents but restrict their arguments to the 32- or 64-bit range.

[func] `ptr Sinteger32(<32-bit int type>` *n*`)`
[func] `ptr Sunsigned32(<32-bit unsigned type>` *n*`)`
[func] `ptr Sinteger64(<64-bit int type>` *n*`)`
[func] `ptr Sunsigned64(<64-bit unsigned type>` *n*`)`

`Scons` and `Sbox` are identical to their Scheme counterparts.

[func] `ptr Scons(ptr` obj_1`, ptr` obj_2`)`
[func] `ptr Sbox(ptr` *obj*`)`

`Sstring_to_symbol` is similar to its Scheme counterpart, `string->symbol`, except that it takes a C string (character pointer) as input.

[func] `ptr Sstring_to_symbol(const char *`*s*`)`

`Smake_string`, `Smake_vector`, `Smake_bytevector`, and `Smake_fxvector` are similar to their Scheme counterparts.

[func] `ptr Smake_string(iptr` *n*`, int` *c*`)`
[func] `ptr Smake_vector(iptr` *n*`, ptr` *obj*`)`
[func] `ptr Smake_bytevector(iptr` *n*`, int` *fill*`)`
[func] `ptr Smake_fxvector(iptr` *n*`, ptr` *fixnum*`)`

`Smake_uninitialized_string` is similar to the one-argument `make-string`.

[func] `ptr Smake_uninitialized_string(iptr` *n*`)`

Windows-specific helper functions. The following helper functions are provided on Windows only.

[func] `char * Sgetenv(const char *`*name*`)`

`Sgetenv` returns the UTF-8-encoded value of UTF-8-encoded environment variable *name* if found and NULL otherwise. Call `free` on the returned value when it is no longer needed.

[func] `wchar_t * Sutf8_to_wide(const char *\s)`
[func] `char * Swide_to_utf8(const wchar_t *\s)`

`Sutf8_to_wide` and `Swide_to_utf8` convert between UTF-8-encoded and UTF-16LE-encoded null-terminated strings. Call `free` on the returned value when it is no longer needed.

Accessing top-level values. Top-level variable bindings may be accessed or assigned via `Stop_level_value` and `Sset_top_level_value`.

[func] `ptr Stop_level_value(ptr` *sym*`)`
[func] `void Sset_top_level_value(ptr` *sym*`, ptr` *obj*`)`

These procedures give fast access to the bindings in the original interaction environment and do not reflect changes to the `interaction-environment` parameter or top-level module imports. To access the current interaction-environment binding for a symbol, it is necessary to call the Scheme `top-level-value` and `set-top-level-value!` procedures instead.

Locking Scheme objects. The storage manager periodically relocates objects in order to reclaim storage and compact the heap. This relocation is completely transparent to Scheme programs, since all pointers to a relocated object are updated to refer to the new location of the object. The storage manager cannot, however, update Scheme pointers that reside outside of the Scheme heap.

As a general rule, all pointers from C variables or data structures to Scheme objects should be discarded before entry (or reentry) into Scheme. That is, if a C procedure receives an object from Scheme or obtains it via the mechanisms described in this section, all pointers to the object should be considered invalid once the C procedure calls into Scheme or returns back to Scheme. Dereferencing an invalid pointer or passing it back to Scheme can have disastrous effects, including unrecoverable memory faults. The foregoing does not apply to immediate objects, e.g., fixnums, characters, booleans, or the empty list. It does apply to all heap-allocated objects, including pairs, vectors, strings, all numbers other than fixnums, ports, procedures, and records.

In practice, the best way to ensure that C code does not retain pointers to Scheme objects is to immediately convert the Scheme objects into C equivalents, if possible. In certain cases, it is not possible to do so, yet retention of the Scheme object is essential to the design of the C portions of the program. In these cases, the object may be *locked* via the library routine `Slock_object` (or from Scheme, the equivalent procedure `lock-object`).

[func] `void Slock_object(ptr` *obj*`)`

Locking an object prevents the storage manager from reclaiming or relocating the object. Locking should be used sparingly, as it introduces memory fragmentation and increases storage management overhead. Locking can also lead to accidental retention of storage if objects are not unlocked. Locking objects that have been made static via heap compaction (see `Scompact_heap` above) is unnecessary but harmless.

Objects may be unlocked via `Sunlock_object` (`unlock-object`).

[func] `void Sunlock_object(ptr` *obj*`)`

An object may be locked more than once by successive calls to `Slock_object` or `lock-object`, in which case it must be unlocked by an equal number of calls to `Sunlock_object` or `unlock-object` before it is truly unlocked.

The function `Sunlocked_objectp` can be used to determine if an object is locked.

[func] `int Sunlocked_objectp(ptr` *obj*`)`

When a foreign procedure call is made into Scheme, a return address pointing into the Scheme code object associated with the foreign procedure is passed implicitly to the C routine. The system therefore locks the code object before calls are made from C back into

Scheme and unlocks it upon return from Scheme. This locking is performed automatically; user code should never need to lock such code objects.

An object contained within a locked object, such as an object in the car of a locked pair, need not also be locked unless a separate C pointer to the object exists.

Registering foreign entry points. Foreign entry points may be made visible to Scheme via `Sforeign_symbol` or `Sregister_symbol`.

[func] `void Sforeign_symbol(const char *`*name*`, void *`*addr*`)`
[func] `void Sregister_symbol(const char *`*name*`, void *`*addr*`)`

External entry points in object files or shared objects loaded as a result of a call to `load-shared-object` are automatically made visible by the system. Once a foreign entry point is made visible, it may be named in a `foreign-procedure` expression to create a Scheme-callable version of the entry point. `Sforeign_symbol` and `Sregister_symbol` allow programs to register nonexternal entry points, entry points in code linked statically with *Chez Scheme*, and entry points into code loaded directly from C, i.e., without `load-shared-object`. `Sforeign_symbol` and `Sregister_symbol` differ only in that `Sforeign_symbol` raises an exception when an attempt is made to register an existing name, whereas `Sregister_symbol` permits existing names to be redefined.

Obtaining Scheme entry points. `Sforeign_callable_entry_point` extracts the entry point from a code object produced by `foreign-callable`, performing the same operation as its Scheme counterpart, i.e., the Scheme procedure `foreign-callable-entry-point`.

[func] `(void (*) (void)) Sforeign_callable_entry_point(ptr` *code*`)`

This can be used to avoid converting the code object into an address until just when it is needed, which may eliminate the need to lock the code object in some circumstances, assuming that the code object is not saved across any calls back into Scheme.

The inverse translation can be made via `Sforeign_callable_code_object`.

[func] `ptr Sforeign_callable_code_object((void (*addr)(void)))`

Low-level support for calls into Scheme. Support for calling Scheme procedures from C is provided by the set of routines documented below. Calling a Scheme procedure that expects a small number of arguments (0–3) involves the use of one of the following routines.

[func] `ptr Scall0(ptr` *procedure*`)`
[func] `ptr Scall1(ptr` *procedure*`, ptr` obj_1`)`
[func] `ptr Scall2(ptr` *procedure*`, ptr` obj_1`, ptr` obj_2`)`
[func] `ptr Scall3(ptr` *procedure*`, ptr` obj_1`, ptr` obj_2`, ptr` obj_3`)`

In each case, the first argument, *procedure*, should be a Scheme procedure. The remaining arguments, which should be Scheme objects, are passed to the procedure. The tools described earlier in this section may be used to convert C datatypes into their Scheme equivalents. A program that automatically generates conversion code from declarations

that are similar to `foreign-procedure` expressions is distributed with *Chez Scheme*. It can be found in the Scheme library directory on most systems in the file `"foreign.ss"`.

A Scheme procedure may be obtained in a number of ways. For example, it may be received as an argument in a call from Scheme into C, obtained via another call to Scheme, extracted from a Scheme data structure, or obtained from the top-level environment via `Stop_level_value`.

A more general interface involving the following routines is available for longer argument lists.

[func] `void Sinitframe(iptr` *n*`)`
[func] `void Sput_arg(iptr` *i*`, ptr` *obj*`)`
[func] `ptr Scall(ptr` *procedure*`, iptr` *n*`)`

A C procedure first calls `Sinitframe` with one argument, the number of arguments to be passed to Scheme. It then calls `Sput_arg` once for each argument (in any order), passing `Sput_arg` the argument number (starting with `1`) and the argument. Finally, it calls `Scall` to perform the call, passing it the Scheme procedure and the number of arguments (the same number as in the call to `Sinitframe`). Programmers should ensure a Scheme call initiated via `Sinitframe` is completed via `Scall` before any other calls to Scheme are made and before a return to Scheme is attempted. If for any reason the call is not completed after `Sinitframe` has been called, it may not be possible to return to Scheme.

The following examples serve to illustrate both the simpler and more general interfaces.

```
/* a particularly silly way to multiply two floating-point numbers */
double mul(double x, double y) {
    ptr times = Stop_level_value(Sstring_to_symbol("*"));

    return Sflonum_value(Scall2(times, Sflonum(x), Sflonum(y)));
}

/* an equally silly way to call printf with five arguments */

/* it is best to define interfaces such as the one below to handle
 * calls into Scheme to prevent accidental attempts to nest frame
 * creation and to help ensure that initiated calls are completed
 * as discussed above.  Specialized versions tailored to particular
 * C argument types may be defined as well, with embedded conversions
 * to Scheme objects. */
ptr Scall5(ptr p, ptr x1, ptr x2, ptr x3, ptr x4, ptr x5) {
    Sinitframe(5);
    Sput_arg(1, x1);
    Sput_arg(2, x2);
    Sput_arg(3, x3);
    Sput_arg(4, x4);
    Sput_arg(5, x5);
    Scall(p, 5);
}
```

```
static void dumpem(char *s, int a, double b, ptr c, char *d) {
    printf(s, a, b, c, d);
}

static void foo(int x, double y, ptr z, char *s) {
    ptr ois, sip, read, expr, eval, c_dumpem;
    char *sexpr = "(foreign-procedure \"dumpem\" (string integer-32\
 double-float scheme-object string) void)";

  /* this series of statements is carefully crafted to avoid referencing
     variables holding Scheme objects after calls into Scheme */
    ois = Stop_level_value(Sstring_to_symbol("open-input-string"));
    sip = Scall1(ois, Sstring(sexpr));
    read = Stop_level_value(Sstring_to_symbol("read"));
    expr = Scall1(read, sip);
    eval = Stop_level_value(Sstring_to_symbol("eval"));
    Sforeign_symbol("dumpem", (void *)dumpem);
    c_dumpem = Scall1(eval, expr);
    Scall5(c_dumpem,
           Sstring("x = %d, y = %g, z = %x, s = %s\n"),
           Sinteger(x),
           Sflonum(y),
           z,
           Sstring(s));
}
```

Calls from C to Scheme should not be made from C interrupt handlers. When Scheme calls into C, the system saves the contents of certain dedicated machine registers in a register save area. When C then calls into Scheme, the registers are restored from the register save area. Because an interrupt can occur at any point in a computation, the contents of the register save locations would typically contain invalid information that would cause the Scheme system to fail to operate properly.

Activating, deactivating, and destroying threads. Three functions are provided by the threaded versions of Scheme to allow C code to notify Scheme when a thread should be activated, deactivated, or destroyed.

[func] `int Sactivate_thread(void)`
[func] `void Sdeactivate_thread(void)`
[func] `int Sdestroy_thread(void)`

A thread created via the Scheme procedure `fork-thread` starts in the active state and need not be activated. Any thread that has been deactivated, and any thread created by some mechanism other than `fork-thread` must, however, be activated before it can access Scheme data or execute Scheme code. A foreign callable that is declared with `__collect_safe` can activate a calling thread. Otherwise, `Sactivate_thread` must be used to activate a thread. It returns 1 the first time the thread is activated and 0 on each subsequent call until the activation is destroyed with `Sdestroy_thread`.

Since active threads operating in C code prevent the storage management system from

garbage collecting, a thread should be deactivated via `Sdeactivate_thread` or through a `foreign-procedure __collect_safe` declaration whenever the thread may spend a significant amount of time in C code. This is especially important whenever the thread calls a C library function, like `read`, that may block indefinitely. Once deactivated, the thread must not touch any Scheme data or execute any Scheme code until it is reactivated, with one exception. The exception is that the thread may access or even modify a locked Scheme object, such as a locked string, that contains no pointers to other, unlocked Scheme objects. (Objects that are not locked may be relocated by the garbage collector while the thread is inactive.)

`Sdestroy_thread` is used to notify the Scheme system that the thread is shut down and any thread-specific data can be released.

Low-level synchronization primitives. The header file defines several preprocessor macros that can be used to lock memory locations in a manner identical to the corresponding ftype lock operations (sections 15.4 and 15.5).

[macro] `void INITLOCK(void *`*addr*`)`
[macro] `void SPINLOCK(void *`*addr*`)`
[macro] `void UNLOCK(void *`*addr*`)`
[macro] `void LOCKED_INCR(void *`*addr*`, int *`*ret*`)`
[macro] `void LOCKED_DECR(void *`*addr*`, int *`*ret*`)`

`LOCKED_INCR` and `LOCKED_DECR` set *ret* to a nonzero (true) value if the incremented or decremented value is 0. Otherwise they set *ret* to 0.

4.9. Example: Socket Operations

This section presents a simple socket interface that employs a combination of Scheme and C code. The C code defines a set of convenient low-level operating-system interfaces that can be used in the higher-level Scheme code to open, close, read from, and write to sockets.

The C code (csocket.c) is given below, followed by the Scheme code (socket.ss). The code should require little or no modification to run on most Unix systems and can be modified to work under Windows (using the Windows *WinSock* interface).

A sample session demonstrating the socket interface follows the code. See Section 9.17 for an example that demonstrates how to use the same socket interface to build a process port that allows transparent input from and output to a subprocess via a Scheme port.

C code.

```
/* csocket.c */
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <unistd.h>

/* c_write attempts to write the entire buffer, pushing through
   interrupts, socket delays, and partial-buffer writes */
int c_write(int fd, char *buf, ssize_t start, ssize_t n) {
    ssize_t i, m;

    buf += start;
    m = n;
    while (m > 0) {
        if ((i = write(fd, buf, m)) < 0) {
            if (errno != EAGAIN && errno != EINTR)
                return i;
        } else {
            m -= i;
            buf += i;
        }
    }
    return n;
}

/* c_read pushes through interrupts and socket delays */
int c_read(int fd, char *buf, size_t start, size_t n) {
    int i;

    buf += start;
    for (;;) {
        i = read(fd, buf, n);
        if (i >= 0) return i;
        if (errno != EAGAIN && errno != EINTR) return -1;
    }
}

/* bytes_ready(fd) returns true if there are bytes available
   to be read from the socket identified by fd */
int bytes_ready(int fd) {
    int n;

    (void) ioctl(fd, FIONREAD, &n);
    return n;
}

/* socket support */
```

```
/* do_socket() creates a new AF_UNIX socket */
int do_socket(void) {

    return socket(AF_UNIX, SOCK_STREAM, 0);
}

/* do_bind(s, name) binds name to the socket s */
int do_bind(int s, char *name) {
    struct sockaddr_un sun;
    int length;

    sun.sun_family = AF_UNIX;
    (void) strcpy(sun.sun_path, name);
    length = sizeof(sun.sun_family) + sizeof(sun.sun_path);

    return bind(s, (struct sockaddr*)(&sun), length);
}

/* do_accept accepts a connection on socket s */
int do_accept(int s) {
    struct sockaddr_un sun;
    socklen_t length;

    length = sizeof(sun.sun_family) + sizeof(sun.sun_path);

    return accept(s, (struct sockaddr*)(&sun), &length);
}

/* do_connect initiates a socket connection */
int do_connect(int s, char *name) {
    struct sockaddr_un sun;
    int length;

    sun.sun_family = AF_UNIX;
    (void) strcpy(sun.sun_path, name);
    length = sizeof(sun.sun_family) + sizeof(sun.sun_path);

    return connect(s, (struct sockaddr*)(&sun), length);
}

/* get_error returns the operating system's error status */
char* get_error(void) {
    extern int errno;
    return strerror(errno);
}
```

Scheme code.

```
;;; socket.ss

;;; Requires csocket.so, built from csocket.c.
(load-shared-object "./csocket.so")

;;; Requires from C library:
```

```
;;;   close, dup, execl, fork, kill, listen, tmpnam, unlink
(case (machine-type)
  [(i3le ti3le a6le ta6le) (load-shared-object "libc.so.6")]
  [(i3osx ti3osx a6osx ta6osx) (load-shared-object "libc.dylib")]
  [else (load-shared-object "libc.so")])

;;; basic C-library stuff

(define close
  (foreign-procedure "close" (int)
    int))

(define dup
  (foreign-procedure "dup" (int)
    int))

(define execl4
  (let ((execl-help
         (foreign-procedure "execl"
           (string string string string void*)
           int)))
    (lambda (s1 s2 s3 s4)
      (execl-help s1 s2 s3 s4 0))))

(define fork
  (foreign-procedure "fork" ()
    int))

(define kill
  (foreign-procedure "kill" (int int)
    int))

(define listen
  (foreign-procedure "listen" (int int)
    int))

(define tmpnam
  (foreign-procedure "tmpnam" (void*)
    string))

(define unlink
  (foreign-procedure "unlink" (string)
    int))

;;; routines defined in csocket.c

(define accept
  (foreign-procedure "do_accept" (int)
    int))

(define bytes-ready?
  (foreign-procedure "bytes_ready" (int)
    boolean))
```

```
(define bind
  (foreign-procedure "do_bind" (int string)
    int))

(define c-error
  (foreign-procedure "get_error" ()
    string))

(define c-read
  (foreign-procedure "c_read" (int u8* size_t size_t)
    ssize_t))

(define c-write
  (foreign-procedure "c_write" (int u8* size_t ssize_t)
    ssize_t))

(define connect
  (foreign-procedure "do_connect" (int string)
    int))

(define socket
  (foreign-procedure "do_socket" ()
    int))

;;; higher-level routines

(define dodup
 ; (dodup old new) closes old and dups new, then checks to
 ; make sure that resulting fd is the same as old
  (lambda (old new)
    (check 'close (close old))
    (unless (= (dup new) old)
      (error 'dodup
        "couldn't set up child process io for fd ~s" old))))

(define dofork
 ; (dofork child parent) forks a child process and invokes child
 ; without arguments and parent with the child's pid
  (lambda (child parent)
    (let ([pid (fork)])
      (cond
        [(= pid 0) (child)]
        [(> pid 0) (parent pid)]
        [else (error 'fork (c-error))]))))

(define setup-server-socket
 ; create a socket, bind it to name, and listen for connections
  (lambda (name)
    (let ([sock (check 'socket (socket))])
      (unlink name)
      (check 'bind (bind sock name))
      (check 'listen (listen sock 1))
      sock)))
```

```
(define setup-client-socket
 ; create a socket and attempt to connect to server
  (lambda (name)
    (let ([sock (check 'socket (socket))])
      (check 'connect (connect sock name))
      sock)))

(define accept-socket
 ; accept a connection
  (lambda (sock)
    (check 'accept (accept sock))))

(define check
 ; signal an error if status x is negative, using c-error to
 ; obtain the operating-system's error message
  (lambda (who x)
    (if (< x 0)
        (error who (c-error))
        x)))

(define terminate-process
 ; kill the process identified by pid
  (lambda (pid)
    (define sigterm 15)
    (kill pid sigterm)
    (void)))
```

Sample session.

```
> (define client-pid)
> (define client-socket)
> (let* ([server-socket-name (tmpnam 0)]
         [server-socket (setup-server-socket server-socket-name)])
   ; fork a child, use it to exec a client Scheme process, and set
   ; up server-side client-pid and client-socket variables.
    (dofork   ; child
      (lambda ()
       ; the child establishes the socket input/output fds as
       ; stdin and stdout, then starts a new Scheme session
        (check 'close (close server-socket))
        (let ([sock (setup-client-socket server-socket-name)])
          (dodup 0 sock)
          (dodup 1 sock))
        (check 'execl (execl4 "/bin/sh" "/bin/sh" "-c" "exec scheme -q"))
        (errorf 'client "returned!"))
      (lambda (pid) ; parent
       ; the parent waits for a connection from the client
        (set! client-pid pid)
        (set! client-socket (accept-socket server-socket))
        (check 'close (close server-socket)))))
> (define put ; procedure to send data to client
```

```
    (lambda (x)
      (let ([s (format "~s~%" x)])
        (c-write client-socket s (string-length s)))
      (void)))
> (define get ; procedure to read data from client
    (let ([buff (make-string 1024)])
      (lambda ()
        (let ([n (c-read client-socket buff (string-length buff))])
          (printf "client:~%~a~%server:~%" (substring buff 0 n))))))
> (get)
server:
> (put '(let ([x 3]) x))
> (get)
client:
3
server:
> (terminate-process client-pid)
> (exit)
```

5. *Binding Forms*

This chapter describes *Chez Scheme* extensions to the set of Revised^6 Report binding forms. See Chapter 4 of *The Scheme Programming Language, 4th Edition* or the Revised^6 Report for a description of standard binding forms.

5.1. Definitions

A definition in Revised^6 Report Scheme is a variable definition, keyword definition, or derived definition, i.e., a syntactic extension that expands into a definition. In addition, the forms within a `begin` expression appearing after a sequence of definitions is spliced onto the end of the sequence of definitions so that definitions at the front of the `begin` expression are treated as if they were part of the outer sequence of definitions. A `let-syntax` or `letrec-syntax` form is treated similarly, so that definitions at the front of the body are treated as if they were part of the outer sequence of definitions, albeit scoped where the bindings of the `let-syntax` or `letrec-syntax` form are visible.

Chez Scheme extends the set of definitions to include `module` forms, `import` forms, `import-only` forms, `meta` definitions, and `alias` forms, although the `module`, `import`, `import-only`, `meta`, and `alias` keywords are not available in a library or RNRS top-level program unless the `scheme` library is included in the library or top-level programs imports. These forms are described in Chatper 11.

In Revised^6 Report Scheme, definitions can appear at the front of a `lambda` or similar body (e.g., a `let` or `letrec` body), at the front of a library body, or intermixed with expressions within an RNRS top-level program body. In *Chez Scheme*, definitions may also be used in the interactive top-level, i.e., they can be intermixed with expressions in the REPL or in program text to be loaded from a file via `load` (Section 12.4). The Revised^6 Report does not mandate the existence nor specify the semantics of an interactive top-level, nor of a `load` procedure.

The macro expander uses the same two-pass algorithm for expanding top-level `begin` expressions as it uses for a `lambda`, `library`, or top-level program body. (This algorithm is described in Section 11.1 of *The Scheme Programming Language, 4th Edition*.) As a result,

```
(begin
  (define-syntax a (identifier-syntax 3))
  (define x a))
```

and

```
(begin
  (define x a)
  (define-syntax a (identifier-syntax 3)))
```

both result in the giving `x` the value 3, even though an unbound variable reference to `a` would result if the two forms within the latter `begin` expression where run independently at top level.

Similarly, the `begin` form produced by a use of

```
(define-syntax define-constant
  (syntax-rules ()
    [(_ x e)
     (begin
       (define t e)
       (define-syntax x (identifier-syntax t)))]))
```

and the `begin` form produced by a use of

```
(define-syntax define-constant
  (syntax-rules ()
    [(_ x e)
     (begin
       (define-syntax x (identifier-syntax t))
       (define t e))]))
```

are equivalent.

The Revised[6] Report specifies that internal variable definitions be treated like `letrec*`, while earlier reports required internal variable definitions to be treated like `letrec`. By default, *Chez Scheme* implements the Revised[6] Report semantics for internal variable definitions, as for all other things, but this behavior may be overridden via the `internal-defines-as-letrec*` parameter.

`internal-defines-as-letrec*` **thread parameter**
libraries: `(chezscheme)`

When this parameter is set to `#t` (the default), internal variable definitions are evaluated using `letrec*` semantics. It may be set to `#f` to revert to the `letrec` semantics for internal variable definitions, for backward compatibility.

5.2. Multiple-value Definitions

`(define-values formals` *expr*`)` **syntax**
libraries: `(chezscheme)`

A `define-values` form is a definition and can appear anywhere other definitions can appear.

It is like a `define` form but permits an arbitrary formals list (like `lambda`) on the left-hand side. It evaluates *expr* and binds the variables appearing in *formals* to the resulting values, in the same manner as the formal parameters of a procedure are bound to its arguments.

```
(let ()
  (define-values (x y) (values 1 2))
  (list x y))  ⇒  (1 2)
(let ()
  (define-values (x y . z) (values 1 2 3 4))
  (list x y z))  ⇒  (1 2 (3 4))
```

A `define-values` form expands into a sequence of definitions, the first for a hidden temporary bound to a data structure holding the values returned by *expr* and the remainder binding each of the formals to the corresponding value or list of values, extracted from the data structure via a reference to the temporary. Because the temporary must be defined before the other variables are defined, this works for internal `define-values` forms only if `internal-defines-as-letrec*` is set to the default value `#t`.

5.3. Recursive Bindings

`(rec` *var expr*`)` **syntax**
returns: value of *expr*
libraries: `(chezscheme)`

The syntactic form `rec` creates a recursive object from *expr* by establishing a binding of *var* within *expr* to the value of *expr*. In essence, it is a special case of `letrec` for self-recursive objects.

This form is useful for creating recursive objects (especially procedures) that do not depend on external variables for the recursion, which are sometimes undesirable because the external bindings can change. For example, a recursive procedure defined at top level depends on the value of the top-level variable given as its name. If the value of this variable should change, the meaning of the procedure itself would change. If the procedure is defined instead with `rec`, its meaning is independent of the variable to which it is bound.

```
(map (rec sum
       (lambda (x)
         (if (= x 0)
             0
             (+ x (sum (- x 1))))))
     '(0 1 2 3 4 5))  ⇒  (0 1 3 6 10 15)

(define cycle
  (rec self
    (list (lambda () self))))

(eq? ((car cycle)) cycle)  ⇒  #t
```

The definition below expands `rec` in terms of `letrec`.

```
(define-syntax rec
  (syntax-rules ()
    [(_ x e) (letrec ((x e)) x)]))
```

5.4. Fluid Bindings

`(fluid-let ((`*var expr*`) ...)` $body_1$ $body_2$ `...)` **syntax**
returns: the values of the body $body_1$ $body_2$ `...`
libraries: `(chezscheme)`

The syntactic form `fluid-let` provides a way to temporarily assign values to a set of variables. The new values are in effect only during the evaluation of the body of the `fluid-let` expression. The scopes of the variables are not determined by `fluid-let`; as with `set!`, the variables must be bound at top level or by an enclosing `lambda` or other binding form. It is possible, therefore, to control the scope of a variable with `lambda` or `let` while establishing a temporary value with `fluid-let`.

Although it is similar in appearance to `let`, its operation is more like that of `set!`. Each *var* is assigned, as with `set!`, to the value of the corresponding *expr* within the body $body_1$ $body_2$ `....` Should the body exit normally or by invoking a continuation made outside of the body (see `call/cc`), the values in effect before the bindings were changed are restored. Should control return back to the body by the invocation of a continuation created within the body, the bindings are changed once again to the values in effect when the body last exited.

Fluid bindings are most useful for maintaining variables that must be shared by a group of procedures. Upon entry to the group of procedures, the shared variables are fluidly bound to a new set of initial values so that on exit the original values are restored automatically. In this way, the group of procedures itself can be reentrant; it may call itself directly or indirectly without affecting the values of its shared variables.

Fluid bindings are similar to *special* bindings in Common Lisp [30], except that (1) there is a single namespace for both lexical and fluid bindings, and (2) the scope of a fluidly bound variable is not necessarily global.

```
(let ([x 3])
  (+ (fluid-let ([x 5])
       x)
     x))  ⇒  8

(let ([x 'a])
  (letrec ([f (lambda (y) (cons x y))])
    (fluid-let ([x 'b])
      (f 'c))))  ⇒  (b . c)

(let ([x 'a])
  (call/cc
    (lambda (k)
       (fluid-let ([x 'b])
```

```
        (letrec ([f (lambda (y) (k '*))])
          (f '*)))))
  x)  ⇒  a
```

`fluid-let` may be defined in terms of `dynamic-wind` as follows.

```
(define-syntax fluid-let
  (lambda (x)
    (syntax-case x ()
      [(_ () b1 b2 ...) #'(let () b1 b2 ...)]
      [(_ ((x e) ...) b1 b2 ...)
       (andmap identifier? #'(x ...))
       (with-syntax ([(y ...) (generate-temporaries #'(x ...))])
         #'(let ([y e] ...)
             (let ([swap (lambda ()
                           (let ([t x]) (set! x y) (set! y t))
                           ...)])
               (dynamic-wind swap (lambda () b1 b2 ...) swap))))])))
```

5.5. Top-Level Bindings

The procedures described in this section allow the direct manipulation of top-level bindings for variables and keywords. They are intended primarily to support the definition of interpreters or compilers for Scheme in Scheme but may be used to access or alter top-level bindings anywhere within a program whether at top level or not.

`(define-top-level-value` *symbol obj*`)` **procedure**
`(define-top-level-value` *symbol obj env*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`define-top-level-value` is used to establish a binding for the variable named by *symbol* to the value *obj* in the environment *env*. If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3).

An exception is raised with condition type `&assertion` if *env* is not mutable.

A call to `define-top-level-value` is similar to a top-level `define` form, except that a call to `define-top-level-value` need not occur at top-level and the variable for which the binding is to be established can be determined at run time, as can the environment.

```
(begin
  (define-top-level-value 'xyz "hi")
  xyz)  ⇒  "hi"

(let ([var 'xyz])
  (define-top-level-value var "mom")
  (list var xyz))  ⇒  (xyz "mom")
```

`(set-top-level-value!` *symbol obj*`)` **procedure**
`(set-top-level-value!` *symbol obj env*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`set-top-level-value!` assigns the variable named by *symbol* to the value *obj* in the environment *env*. If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3).

An exception is raised with condition type `&assertion` if the identifier named by *symbol* is not defined as a variable in *env* or if the variable or environment is not mutable.

`set-top-level-value!` is similar to `set!` when `set!` is used on top-level variables except that the variable to be assigned can be determined at run time, as can the environment.

```
(let ([v (let ([cons list])
           (set-top-level-value! 'cons +)
           (cons 3 4))])
  (list v (cons 3 4)))  ⇒  ((3 4) 7)
```

`(top-level-value` *symbol*`)` **procedure**
`(top-level-value` *symbol env*`)` **procedure**
returns: the top-level value of the variable named by *symbol* in *env*
libraries: `(chezscheme)`

If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3).

An exception is raised with condition type `&assertion` if the identifier named by *symbol* is not defined as a variable in *env*.

`top-level-value` is similar to a top-level variable reference except that the variable to be referenced can be determined at run time, as can the environment.

```
(let ([cons +])
  (list (cons 3 4)
        ((top-level-value 'cons) 3 4)))  ⇒  (7 (3 . 4))
```

```
(define e (copy-environment (scheme-environment)))
(define-top-level-value 'pi 3.14 e)
(top-level-value 'pi e)  ⇒  3.14
(set-top-level-value! 'pi 3.1416 e)
(top-level-value 'pi e)  ⇒  3.1416
```

`(top-level-bound?` *symbol*`)` **procedure**
`(top-level-bound?` *symbol env*`)` **procedure**
returns: `#t` if *symbol* is defined as a variable in *env*, `#f` otherwise
libraries: `(chezscheme)`

If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-

level evaluation environment (Section 12.3).

This predicate is useful in an interpreter to check for the existence of a top-level binding before requesting the value with `top-level-value`.

```
(top-level-bound? 'xyz)  ⇒  #f

(begin
  (define-top-level-value 'xyz 3)
  (top-level-bound? 'xyz))  ⇒  #t

(define e (copy-environment (interaction-environment)))
(define-top-level-value 'pi 3.14 e)
(top-level-bound? 'pi)  ⇒  #f
(top-level-bound? 'pi e)  ⇒  #t
```

`(top-level-mutable?` *symbol*`)` **procedure**
`(top-level-mutable?` *symbol env*`)` **procedure**
returns: `#t` if *symbol* is mutable in *env*, `#f` otherwise
libraries: `(chezscheme)`

If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3).

This predicate is useful in an interpreter to check whether a variable can be assigned before assigning it with `set-top-level-value!`.

```
(define xyz 3)
(top-level-mutable? 'xyz)  ⇒  #t
(set-top-level-value! 'xyz 4)
(top-level-value 'xyz)  ⇒  4

(define e (copy-environment (interaction-environment) #f))
(top-level-mutable? 'xyz e)  ⇒  #f
(set-top-level-value! 'xyz e)  ⇒  exception: xyz is immutable
```

`(define-top-level-syntax` *symbol obj*`)` **procedure**
`(define-top-level-syntax` *symbol obj env*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`define-top-level-syntax` is used to establish a top-level binding for the identifier named by *symbol* to the value of *obj* in the environment *env*. The value must be a procedure, the result of a call to `make-variable-transformer`, or the result of a call to `top-level-syntax`. If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3).

An exception is raised with condition type `&assertion` if *env* is not mutable.

A call to `define-top-level-syntax` is similar to a top-level `define-syntax` form, except that a call to `define-top-level-syntax` need not occur at top-level and the identifier

for which the binding is to be established can be determined at run time, as can the environment.

```
(define-top-level-syntax 'let1
  (syntax-rules ()
    [(_ x e b1 b2 ...) (let ([x e]) b1 b2 ...)]))
(let1 a 3 (+ a 1))  ⇒  4
```

`define-top-level-syntax` can also be used to attach to an identifier arbitrary compile-time bindings obtained via `top-level-syntax`.

`(top-level-syntax` *symbol*`)` **procedure**
`(top-level-syntax` *symbol env*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`top-level-syntax` is used to retrieve the transformer, compile-time value, or other compile-time binding to which the identifier named by *symbol* is bound in the environment *env*. If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3). All identifiers bound in an environment have compile-time bindings, including variables.

An exception is raised with condition type `&assertion` if the identifier named by *symbol* is not defined as a keyword in *env*.

```
(define-top-level-syntax 'also-let (top-level-syntax 'let))
(also-let ([x 3] [y 4]) (+ x y))  ⇒  7
```

```
(define foo 17)
(define-top-level-syntax 'also-foo (top-level-syntax 'foo))
also-foo  ⇒  17
(set! also-foo 23)
also-foo  ⇒  23
foo  ⇒  23
```

The effect of the last example can be had more clearly with `alias`:

```
(define foo 17)
(alias also-foo foo)
also-foo  ⇒  17
(set! also-foo 23)
also-foo  ⇒  23
foo  ⇒  23
```

`(top-level-syntax?` *symbol*`)` **procedure**
`(top-level-syntax?` *symbol env*`)` **procedure**
returns: `#t` if *symbol* is bound as a keyword in *env*, `#f` otherwise
libraries: `(chezscheme)`

If *env* is not provided, it defaults to the value of `interaction-environment`, i.e., the top-level evaluation environment (Section 12.3).

All identifiers bound in an environment have compile-time bindings, including variables, so this predicate amounts to a bound check, but is more general than `top-level-bound?`, which returns true only for bound variables.

```
(define xyz 'hello)
(top-level-syntax? 'cons)  ⇒  #t
(top-level-syntax? 'lambda)  ⇒  #t
(top-level-syntax? 'hello)  ⇒  #t

(top-level-syntax? 'cons (scheme-environment))  ⇒  #t
(top-level-syntax? 'lambda (scheme-environment))  ⇒  #t
(top-level-syntax? 'hello (scheme-environment))  ⇒  #f
```

6. Control Structures

This chapter describes *Chez Scheme* extensions to the set of standard control structures. See Chapter 5 of *The Scheme Programming Language, 4th Edition* or the Revised^6 Report on Scheme for a description of standard control structures.

6.1. Conditionals

`(exclusive-cond` *clause*$_1$ *clause*$_2$ `...)` **syntax**
returns: see below
libraries: `(chezscheme)`

`exclusive-cond` is a version of `cond` (Section 5.3 of TSPLFOUR) that differs from `cond` in that the tests embedded within the clauses are assumed to be exclusive in the sense that if one of the tests is true, the others are not. This allows the implementation to reorder clauses when profiling information is available at expansion time (Section 12.7).

The `(`*test*`)` form of clause is not supported. The order chosen when profiling information is available is based on the relative numbers of times the RHS of each clause is executed, and `(`*test*`)` has no RHS. `(`*test* `=> values)` is equivalent, abeit less concise.

`(case` *expr*$_0$ *clause*$_1$ *clause*$_2$ `...)` **syntax**
returns: see below
libraries: `(chezscheme)`

Each clause but the last must take one of the forms:

`((`*key* `...)` *expr*$_1$ *expr*$_2$ `...)`
`(`*key* *expr*$_1$ *expr*$_2$ `...)`

where each *key* is a datum distinct from the other keys. The last clause may be in the above form or it may be an `else` clause of the form

`(else` *expr*$_1$ *expr*$_2$ `...)`

expr$_0$ is evaluated and the result is compared (using `equal?`) against the keys of each clause in order. If a clause containing a matching key is found, the expressions *expr*$_1$ *expr*$_2$ `...` are evaluated in sequence and the values of the last expression are returned.

If none of the clauses contains a matching key and an `else` clause is present, the expressions *expr*$_1$ *expr*$_2$ ... of the `else` clause are evaluated in sequence and the values of the last expression are returned.

If none of the clauses contains a matching key and no `else` clause is present, the value or values are unspecified.

The Revised[6] Report version of `case` does not support singleton keys (the second of the first two clause forms above) and uses `eqv?` rather than `equal?` as the comparison procedure. Both versions are defined in terms of `exclusive-cond` so that if profiling information is available at expansion time, the clauses will be reordered to put those that are most frequently executed first.

```
(let ([ls '(ii iv)])
  (case (car ls)
    [i 1]
    [ii 2]
    [iii 3]
    [(iiii iv) 3]
    [else 'out-of-range]))  ⇒  2

(define p
  (lambda (x)
    (case x
      [("abc" "def") 'one]
      [((a b c)) 'two]
      [else #f])))

(p (string #\d #\e #\f))  ⇒  one
(p '(a b c))  ⇒  two
```

(record-case *expr clause*$_1$ *clause*$_2$ ...**)** **syntax**
returns: see explanation
libraries: `(chezscheme)`

`record-case` is a restricted form of `case` that supports the destructuring of *records*, or *tagged lists*. A record has as its first element a tag that determines what "type" of record it is; the remaining elements are the fields of the record.

Each clause but the last must take the form

((*key* ...) *formals body*$_1$ *body*$_2$ ...)

where each *key* is a datum distinct from the other keys. The last clause may be in the above form or it may be an `else` clause of the form

(`else` *body*$_1$ *body*$_2$ ...)

expr must evaluate to a pair. *expr* is evaluated and the car of its value is compared (using `eqv?`) against the keys of each clause in order. If a clause containing a matching key is found, the variables in *formals* are bound to the remaining elements of the list and the expressions *body*$_1$ *body*$_2$ ... are evaluated in sequence. The value of the last expression is

returned. The effect is identical to the application of

`(lambda` *formals* *body*$_1$ *body*$_2$ `...)`

to the cdr of the list.

If none of the clauses contains a matching key and an `else` clause is present, the expressions *body*$_1$ *body*$_2$ `...` of the `else` clause are evaluated in sequence and the value of the last expression is returned.

If none of the clauses contains a matching key and no `else` clause is present, the value is unspecified.

```
(define calc
  (lambda (x)
    (record-case x
      [(add) (x y) (+ x y)]
      [(sub) (x y) (- x y)]
      [(mul) (x y) (* x y)]
      [(div) (x y) (/ x y)]
      [else (assertion-violationf 'calc "invalid expression ~s" x)])))

(calc '(add 3 4))  ⇒  7
(calc '(div 3 4))  ⇒  3/4
```

6.2. Mapping and Folding

`(ormap` *procedure* *list*$_1$ *list*$_2$ `...)` **procedure**
returns: see explanation
libraries: `(chezscheme)`

`ormap` is identical to the Revised[6] Report `exists`.

`(andmap` *procedure* *list*$_1$ *list*$_2$ `...)` **procedure**
returns: see explanation
libraries: `(chezscheme)`

`andmap` is identical to the Revised[6] Report `for-all`.

6.3. Continuations

Chez Scheme supports one-shot continuations as well as the standard multi-shot continuations obtainable via `call/cc`. One-shot continuations are continuations that may be invoked at most once, whether explicitly or implicitly. They are obtained with `call/1cc`.

(call/1cc *procedure*) **procedure**
returns: see below
libraries: (chezscheme)

call/1cc obtains its continuation and passes it to *procedure*, which should accept one argument. The continuation itself is represented by a procedure. This procedure normally takes one argument but may take an arbitrary number of arguments depending upon whether the context of the call to call/1cc expects multiple return values or not. When this procedure is applied to a value or values, it returns the values to the continuation of the call/1cc application.

The continuation obtained by call/1cc is a "one-shot continuation." A one-shot continuation should not be returned to multiple times, either by invoking the continuation or returning normally from *procedure* more than once. A one-shot continuation is "promoted" into a normal (multishot) continuation, however, if it is still active when a normal continuation is obtained by call/cc. After a one-shot continuation is promoted into a multishot continuation, it behaves exactly as if it had been obtained via call/cc. This allows call/cc and call/1cc to be used together transparently in many applications.

One-shot continuations may be more efficient for some applications than multishot continuations. See the paper "Representing control in the presence of one-shot continuations" [3] for more information about one-shot continuations, including how they are implemented in *Chez Scheme*.

The following examples highlight the similarities and differences between one-shot and normal continuations.

```
(define prod
 ; compute the product of the elements of ls, bugging out
 ; with no multiplications if a zero element is found
  (lambda (ls)
    (lambda (k)
      (if (null? ls)
          1
          (if (= (car ls) 0)
              (k 0)
              (* (car ls) ((prod (cdr ls)) k)))))))

(call/cc (prod '(1 2 3 4)))  ⇒  24
(call/1cc (prod '(1 2 3 4)))  ⇒  24

(call/cc (prod '(1 2 3 4 0)))  ⇒  0
(call/1cc (prod '(1 2 3 4 0)))  ⇒  0

(let ([k (call/cc (lambda (x) x))])
  (k (lambda (x) 0)))  ⇒  0

(let ([k (call/1cc (lambda (x) x))])
  (k (lambda (x) 0)))  ⇒  exception
```

`(dynamic-wind` *in body out*`)` **procedure**
`(dynamic-wind` *critical? in body out*`)` **procedure**
returns: values resulting from the application of *body*
libraries: `(chezscheme)`

The first form is identical to the Revised[6] Report `dynamic-wind`. When the optional *critical?* argument is present and non-false, the *in* thunk is invoked in a critical section along with the code that records that the body has been entered, and the *out* thunk is invoked in a critical section along with the code that records that the body has been exited. Extreme caution must be taken with this form of `dynamic-wind`, since an error or long-running computation can leave interrupts and automatic garbage collection disabled.

6.4. Engines

Engines are a high-level process abstraction supporting *timed preemption* [15, 24]. Engines may be used to simulate multiprocessing, implement operating system kernels, and perform nondeterministic computations.

`(make-engine` *thunk*`)` **procedure**
returns: an engine
libraries: `(chezscheme)`

An engine is created by passing a thunk (no argument procedure) to `make-engine`. The body of the thunk is the computation to be performed by the engine. An engine itself is a procedure of three arguments:

ticks: a positive integer that specifies the amount of *fuel* to be given to the engine. An engine executes until this fuel runs out or until its computation finishes.

complete: a procedure of one or more arguments that specifies what to do if the computation finishes. Its arguments are the amount of fuel left over and the values produced by the computation.

expire: a procedure of one argument that specifies what to do if the fuel runs out before the computation finishes. Its argument is a new engine capable of continuing the computation from the point of interruption.

When an engine is applied to its arguments, it sets up a timer to fire in *ticks* time units. (See `set-timer` on page 322.) If the engine computation completes before the timer expires, the system invokes *complete*, passing it the number of *ticks* left over and the values produced by the computation. If, on the other hand, the timer goes off before the engine computation completes, the system creates a new engine from the continuation of the interrupted computation and passes this engine to *expire*. *complete* and *expire* are invoked in the continuation of the engine invocation.

An implementation of engines is given in Section 12.11. of *The Scheme Programming Language, 4th Edition*.

Do not use the timer interrupt (see `set-timer`) and engines at the same time, since engines are implemented in terms of the timer.

The following example creates an engine from a trivial computation, 3, and gives the engine 10 ticks.

```
(define eng
  (make-engine
    (lambda () 3)))

(eng 10
     (lambda (ticks value) value)
     (lambda (x) x))  ⇒  3
```

It is often useful to pass `list` as the *complete* procedure to an engine, causing an engine that completes to return a list whose first element is the ticks remaining and whose remaining elements are the values returned by the computation.

```
(define eng
  (make-engine
    (lambda () 3)))

(eng 10
     list
     (lambda (x) x))  ⇒  (9 3)
```

In the example above, the value is 3 and there are 9 ticks left over, i.e., it takes one unit of fuel to evaluate 3. (The fuel amounts given here are for illustration only. Your mileage may vary.)

Typically, the engine computation does not finish in one try. The following example displays the use of an engine to compute the 10th Fibonacci number in steps.

```
(define fibonacci
  (lambda (n)
    (let fib ([i n])
      (cond
        [(= i 0) 0]
        [(= i 1) 1]
        [else (+ (fib (- i 1))
                 (fib (- i 2)))]))))

(define eng
  (make-engine
    (lambda ()
      (fibonacci 10))))

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired"))  ⇒  "expired"
```

```
(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired"))  ⇒  "expired"

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired"))  ⇒  "expired"

(eng 50
     list
     (lambda (new-eng)
       (set! eng new-eng)
       "expired"))  ⇒  (21 55)
```

Each time the engine's fuel runs out, the *expire* procedure assigns `eng` to the new engine. The entire computation requires four blocks of 50 ticks to complete; of the last 50 it uses all but 21. Thus, the total amount of fuel used is 179 ticks. This leads to the following procedure, `mileage`, which "times" a computation using engines:

```
(define mileage
  (lambda (thunk)
    (let loop ([eng (make-engine thunk)] [total-ticks 0])
      (eng 50
           (lambda (ticks . values)
             (+ total-ticks (- 50 ticks)))
           (lambda (new-eng)
             (loop new-eng
                   (+ total-ticks 50)))))))

(mileage (lambda () (fibonacci 10)))  ⇒  179
```

The choice of 50 for the number of ticks to use each time is arbitrary, of course. It might make more sense to pass a much larger number, say 10000, in order to reduce the number of times the computation is interrupted.

The next procedure is similar to `mileage`, but it returns a list of engines, one for each tick it takes to complete the computation. Each of the engines in the list represents a "snapshot" of the computation, analogous to a single frame of a moving picture. `snapshot` might be useful for "single stepping" a computation.

```
(define snapshot
  (lambda (thunk)
    (let again ([eng (make-engine thunk)])
      (cons eng
            (eng 1 (lambda (t . v) '()) again)))))
```

The recursion embedded in this procedure is rather strange. The complete procedure

performs the base case, returning the empty list, and the expire procedure performs the recursion.

The next procedure, `round-robin`, could be the basis for a simple time-sharing operating system. `round-robin` maintains a queue of processes (a list of engines), cycling through the queue in a *round-robin* fashion, allowing each process to run for a set amount of time. `round-robin` returns a list of the values returned by the engine computations in the order that the computations complete. Each computation is assumed to produce exactly one value.

```
(define round-robin
  (lambda (engs)
    (if (null? engs)
        '()
        ((car engs)
         1
         (lambda (ticks value)
           (cons value (round-robin (cdr engs))))
         (lambda (eng)
           (round-robin
             (append (cdr engs) (list eng))))))))
```

Since the amount of fuel supplied each time, one tick, is constant, the effect of `round-robin` is to return a list of the values sorted from the quickest to complete to the slowest to complete. Thus, when we call `round-robin` on a list of engines, each computing one of the Fibonacci numbers, the output list is sorted with the earlier Fibonacci numbers first, regardless of the order of the input list.

```
(round-robin
  (map (lambda (x)
         (make-engine
           (lambda ()
             (fibonacci x))))
       '(4 5 2 8 3 7 6 2)))  ⇒  (1 1 2 3 5 8 13 21)
```

More interesting things can happen if the amount of fuel varies each time through the loop. In this case, the computation would be nondeterministic, i.e., the results would vary from call to call.

The following syntactic form, `por` (parallel-or), returns the first of its expressions to complete with a true value. `por` is implemented with the procedure `first-true`, which is similar to `round-robin` but quits when any of the engines completes with a true value. If all of the engines complete, but none with a true value, `first-true` (and hence `por`) returns `#f`. Also, although `first-true` passes a fixed amount of fuel to each engine, it chooses the next engine to run at random, and is thus nondeterministic.

```
(define-syntax por
  (syntax-rules ()
    [(_ x ...)
     (first-true
```

```
      (list (make-engine (lambda () x)) ...))]))

(define first-true
  (let ([pick
         (lambda (ls)
           (list-ref ls (random (length ls))))])
    (lambda (engs)
      (if (null? engs)
          #f
          (let ([eng (pick engs)])
            (eng 1
                 (lambda (ticks value)
                   (or value
                       (first-true
                         (remq eng engs))))
                 (lambda (new-eng)
                   (first-true
                     (cons new-eng
                           (remq eng engs))))))))))
```

The list of engines is maintained with `pick`, which randomly chooses an element of the list, and `remq`, which removes the chosen engine from the list. Since `por` is nondeterministic, subsequent uses with the same expressions may not return the same values.

```
(por 1 2 3)  ⇒  2
(por 1 2 3)  ⇒  3
(por 1 2 3)  ⇒  2
(por 1 2 3)  ⇒  1
```

Furthermore, even if one of the expressions is an infinite loop, `por` still finishes as long as one of the other expressions completes and returns a true value.

```
(por (let loop () (loop)) 2)  ⇒  2
```

With `engine-return` and `engine-block`, it is possible to terminate an engine explicitly. `engine-return` causes the engine to complete, as if the computation had finished. Its arguments are passed to the *complete* procedure along with the number of ticks remaining. It is essentially a nonlocal exit from the engine. Similarly, `engine-block` causes the engine to expire, as if the timer had run out. A new engine is made from the continuation of the call to `engine-block` and passed to the *expire* procedure.

`(engine-block)` **procedure**
returns: does not return
libraries: `(chezscheme)`

This causes a running engine to stop, create a new engine capable of continuing the computation, and pass the new engine to the original engine's third argument (the expire procedure). Any remaining fuel is forfeited.

```
(define eng
  (make-engine
    (lambda ()
      (engine-block)
      "completed")))

(eng 100
     (lambda (ticks value) value)
     (lambda (x)
        (set! eng x)
        "expired"))  ⇒  "expired"

(eng 100
     (lambda (ticks value) value)
     (lambda (x)
        (set! eng x)
        "expired"))  ⇒  "completed"
```

(engine-return *obj* **...)** **procedure**

returns: does not return
libraries: (chezscheme)

This causes a running engine to stop and pass control to the engine's *complete* argument. The first argument passed to the complete procedure is the amount of fuel remaining, as usual, and the remaining arguments are the objects *obj* ... passed to `engine-return`.

```
(define eng
  (make-engine
    (lambda ()
      (reverse (engine-return 'a 'b 'c)))))

(eng 100
     (lambda (ticks . values) values)
     (lambda (new-eng) "expired"))  ⇒  (a b c)
```

7. Operations on Objects

This chapter describes operations specific to *Chez Scheme* on nonnumeric objects, including standard objects such as pairs and numbers and *Chez Scheme* extensions such as boxes and records. Chapter 8 describes operations on numbers. See Chapter 6 of *The Scheme Programming Language, 4th Edition* or the Revised[6] Report on Scheme for a description of standard operations on objects.

7.1. Missing R6RS Type Predicates

`(enum-set?` *obj*`)` **procedure**
returns: `#t` if *obj* is an enum set, `#f` otherwise
libraries: `(chezscheme)`

This predicate is not defined by the Revised[6] Report, but should be.

`(record-constructor-descriptor?` *obj*`)` **procedure**
returns: `#t` if *obj* is a record constructor descriptor, `#f` otherwise
libraries: `(chezscheme)`

This predicate is not defined by the Revised[6] Report, but should be.

7.2. Pairs and Lists

`(atom?` *obj*`)` **procedure**
returns: `#t` if *obj* is not a pair, `#f` otherwise
libraries: `(chezscheme)`

`atom?` is equivalent to `(lambda (x) (not (pair? x)))`.

```
(atom? '(a b c))  ⇒  #f
(atom? '(3 . 4))  ⇒  #f
(atom? '())  ⇒  #t
(atom? 3)  ⇒  #t
```

`(list-head` *list n*`)` **procedure**
returns: a list of the first *n* elements of *list*
libraries: `(chezscheme)`

n must be an exact nonnegative integer less than or equal to the length of *list*.

`list-head` and the standard Scheme procedure `list-tail` may be used together to split a list into two separate lists. While `list-tail` performs no allocation but instead returns a sublist of the original list, `list-head` always returns a copy of the first portion of the list.

`list-head` may be defined as follows.

```
(define list-head
  (lambda (ls n)
    (if (= n 0)
        '()
        (cons (car ls) (list-head (cdr ls) (- n 1))))))
```

```
(list-head '(a b c) 0)  ⇒  ()
(list-head '(a b c) 2)  ⇒  (a b)
(list-head '(a b c) 3)  ⇒  (a b c)
(list-head '(a b c . d) 2)  ⇒  (a b)
(list-head '(a b c . d) 3)  ⇒  (a b c)
(list-head '#1=(a . #1#) 5)  ⇒  (a a a a a)
```

`(last-pair` *list*`)` **procedure**
returns: the last pair of a *list*
libraries: `(chezscheme)`

list must not be empty. `last-pair` returns the last pair (not the last element) of *list*. *list* may be an improper list, in which case the last pair is the pair containing the last element and the terminating object.

```
(last-pair '(a b c d))  ⇒  (d)
(last-pair '(a b c . d))  ⇒  (c . d)
```

`(list-copy` *list*`)` **procedure**
returns: a copy of *list*
libraries: `(chezscheme)`

`list-copy` returns a list `equal?` to *list*, using new pairs to reform the top-level list structure.

```
(list-copy '(a b c))  ⇒  (a b c)
```

```
(let ([ls '(a b c)])
  (equal? ls (list-copy ls)))  ⇒  #t
```

```
(let ([ls '(a b c)])
  (let ([ls-copy (list-copy ls)])
    (or (eq? ls-copy ls)
        (eq? (cdr ls-copy) (cdr ls))
        (eq? (cddr ls-copy) (cddr ls)))))  ⇒  #f
```

`(list*` *obj ... final-obj*`)` **procedure**
returns: a list of *obj ...* terminated by *final-obj*
libraries: `(chezscheme)`

`list*` is identical to the Revised[6] Report `cons*`.

`(make-list` *n*`)` **procedure**
`(make-list` *n obj*`)` **procedure**
returns: a list of *n objs*
libraries: `(chezscheme)`

n must be a nonnegative integer. If *obj* is omitted, the elements of the list are unspecified.

```
(make-list 0 '())  ⇒  ()
(make-list 3 0)  ⇒  (0 0 0)
(make-list 2 "hi")  ⇒  ("hi" "hi")
```

`(iota` *n*`)` **procedure**
returns: a list of integers from 0 (inclusive) to *n* (exclusive)
libraries: `(chezscheme)`

n must be an exact nonnegative integer.

```
(iota 0)  ⇒  ()
(iota 5)  ⇒  (0 1 2 3 4)
```

`(enumerate` *ls*`)` **procedure**
returns: a list of integers from 0 (inclusive) to the length of *ls* (exclusive)
libraries: `(chezscheme)`

```
(enumerate '())  ⇒  ()
(enumerate '(a b c))  ⇒  (0 1 2)
(let ([ls '(a b c)])
  (map cons ls (enumerate ls)))  ⇒  ((a . 0) (b . 1) (c . 2))
```

(remq! *obj list*) **procedure**
(remv! *obj list*) **procedure**
(remove! *obj list*) **procedure**
returns: a list containing the elements of *list* with all occurrences of *obj* removed
libraries: (chezscheme)

These procedures are similar to the Revised[6] Report remq, remv, and remove procedures, except remq!, remv! and remove! use pairs from the input list to build the output list. They perform less allocation but are not necessarily faster than their nondestructive counterparts. Their use can easily lead to confusing or incorrect results if used indiscriminately.

```
(remq! 'a '(a b a c a d))  ⇒  (b c d)

(remv! #\a '(#\a #\b #\c))  ⇒  (#\b #\c)

(remove! '(c) '((a) (b) (c)))  ⇒  ((a) (b))
```

(substq *new old tree*) **procedure**
(substv *new old tree*) **procedure**
(subst *new old tree*) **procedure**
(substq! *new old tree*) **procedure**
(substv! *new old tree*) **procedure**
(subst! *new old tree*) **procedure**
returns: a tree with *new* substituted for occurrences of *old* in *tree*
libraries: (chezscheme)

These procedures traverse *tree*, replacing all objects equivalent to the object *old* with the object *new*.

The equivalence test for substq and substq! is eq?, for substv and substv! is eqv?, and for subst and subst! is equal?.

substq!, substv!, and subst! perform the substitutions destructively. They perform less allocation but are not necessarily faster than their nondestructive counterparts. Their use can easily lead to confusing or incorrect results if used indiscriminately.

```
(substq 'a 'b '((b c) b a))  ⇒  ((a c) a a)

(substv 2 1 '((1 . 2) (1 . 4) . 1))  ⇒  ((2 . 2) (2 . 4) . 2)

(subst 'a
       '(a . b)
       '((a . b) (c a . b) . c))  ⇒  (a (c . a) . c)

(let ([tr '((b c) b a)])
  (substq! 'a 'b tr)
  tr)  ⇒  ((a c) a a)
```

`(reverse!` *list*`)` **procedure**
returns: a list containing the elements of *list* in reverse order
libraries: `(chezscheme)`

`reverse!` destructively reverses *list* by reversing its links. Using `reverse!` in place of `reverse` reduces allocation but is not necessarily faster than `reverse`. Its use can easily lead to confusing or incorrect results if used indiscriminately.

```
(reverse! '()) ⇒ ()
(reverse! '(a b c)) ⇒ (c b a)

(let ([x '(a b c)])
  (reverse! x)
  x) ⇒ (a)

(let ([x '(a b c)])
  (set! x (reverse! x))
  x) ⇒ (c b a)
```

`(append!` *list* `...)` **procedure**
returns: the concatenation of the input lists
libraries: `(chezscheme)`

Like `append`, `append!` returns a new list consisting of the elements of the first list followed by the elements of the second list, the elements of the third list, and so on. Unlike `append`, `append!` reuses the pairs in all of the arguments in forming the new list. That is, the last cdr of each list argument but the last is changed to point to the next list argument. If any argument but the last is the empty list, it is essentially ignored. The final argument (which need not be a list) is not altered.

`append!` performs less allocation than `append` but is not necessarily faster. Its use can easily lead to confusing or incorrect results if used indiscriminately.

```
(append! '(a b) '(c d)) ⇒ (a b c d)

(let ([x '(a b)])
  (append! x '(c d))
  x) ⇒ (a b c d)
```

7.3. Characters

Chez Scheme extends the syntax of characters in two ways. First, a `#\` prefix followed by exactly three octal digits is read as a character whose numeric code is the octal value of the three digits, e.g., `#\044` is read as `#\$`. Second, it recognizes several nonstandard named characters: `#\rubout` (which is the same as `#\delete`), `#\bel` (which is the same as `#\alarm`), `#\vt` (which is the same as `#\vtab`), `#\nel` (the Unicode NEL character), and `#\ls` (the Unicode LS character). The set of nonstandard character names may be changed via the procedure `char-name` (page 9.14).

These extensions are disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

`(char=?` $char_1$ $char_2$ `...)` **procedure**
`(char<?` $char_1$ $char_2$ `...)` **procedure**
`(char>?` $char_1$ $char_2$ `...)` **procedure**
`(char<=?` $char_1$ $char_2$ `...)` **procedure**
`(char>=?` $char_1$ $char_2$ `...)` **procedure**
`(char-ci=?` $char_1$ $char_2$ `...)` **procedure**
`(char-ci<?` $char_1$ $char_2$ `...)` **procedure**
`(char-ci>?` $char_1$ $char_2$ `...)` **procedure**
`(char-ci<=?` $char_1$ $char_2$ `...)` **procedure**
`(char-ci>=?` $char_1$ $char_2$ `...)` **procedure**
returns: `#t` if the relation holds, `#f` otherwise
libraries: `(chezscheme)`

These predicates are identical to the Revised[6] Report counterparts, except they are extended to accept one or more rather than two or more arguments. When passed one argument, each of these predicates returns `#t`.

```
(char>? #\a)  ⇒  #t
(char<? #\a)  ⇒  #t
(char-ci=? #\a)  ⇒  #t
```

`(char-` $char_1$ $char_2$`)` **procedure**
returns: the integer difference between $char_1$ and $char_2$
libraries: `(chezscheme)`

`char-` subtracts the integer value of $char_2$ from the integer value of $char_1$ and returns the difference. The following examples assume that the integer representation is the ASCII code for the character.

```
(char- #\f #\e)  ⇒  1
```

```
(define digit-value
 ; returns the digit value of the base-r digit c, or #f if c
 ; is not a valid digit
  (lambda (c r)
    (let ([v (cond
               [(char<=? #\0 c #\9) (char- c #\0)]
               [(char<=? #\A c #\Z) (char- c #\7)]
               [(char<=? #\a c #\z) (char- c #\W)]
               [else 36])])
      (and (fx< v r) v))))
(digit-value #\8 10)  ⇒  8
(digit-value #\z 10)  ⇒  #f
(digit-value #\z 36)  ⇒  35
```

`char-` might be defined as follows.

```
(define char-
  (lambda (c1 c2)
    (- (char->integer c1) (char->integer c2))))
```

7.4. Strings

Chez Scheme extends the standard string syntax with two character escapes: `\'`, which produces the single quote character, and `\`*nnn*, i.e., backslash followed by 3 octal digits, which produces the character equivalent of the octal value of the 3 digits. These extensions are disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

All strings are mutable by default, including constants. A program can create immutable strings via `string->immutable-string`. Any attempt to modify an immutable string causes an exception to be raised.

The length and indices of a string in *Chez Scheme* are always fixnums.

`(string=?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string<?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string>?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string<=?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string>=?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string-ci=?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string-ci<?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string-ci>?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string-ci<=?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
`(string-ci>=?` $string_1$ $string_2$ $string_3$ ...`)` **procedure**
returns: `#t` if the relation holds, `#f` otherwise
libraries: `(chezscheme)`

These predicates are identical to the Revised[6] Report counterparts, except they are extended to accept one or more rather than two or more arguments. When passed one argument, each of these predicates returns `#t`.

```
(string>? "a")  ⇒  #t
(string<? "a")  ⇒  #t
(string-ci=? "a")  ⇒  #t
```

`(string-copy!` *src src-start dst dst-start n*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

src and *dst* must be strings, and *dst* must be mutable. *src-start*, *dst-start*, and *n* must be exact nonnegative integers. The sum of *src-start* and *n* must not exceed the length of *src*,

and the sum of *dst-start* and *n* must not exceed the length of *dst*.

`string-copy!` overwrites the *n* bytes of *dst* starting at *dst-start* with the *n* bytes of *dst* starting at *src-start*. This works even if *dst* is the same string as *src* and the source and destination locations overlap. That is, the destination is filled with the characters that appeared at the source before the operation began.

```
(define s1 "to boldly go")
(define s2 (make-string 10 #\-))

(string-copy! s1 3 s2 1 3)
s2  ⇒  "-bol------"

(string-copy! s1 7 s2 4 2)
s2  ⇒  "-bolly----"

(string-copy! s2 2 s2 5 4)
s2  ⇒  "-bollolly-"
```

(substring-fill! *string start end char***)** **procedure**
returns: unspecified
libraries: `(chezscheme)`

string must be mutable. The characters of *string* from *start* (inclusive) to *end* (exclusive) are set to *char*. *start* and *end* must be nonnegative integers; *start* must be strictly less than the length of *string*, while *end* may be less than or equal to the length of *string*. If $end \leq start$, the string is left unchanged.

```
(let ([str (string-copy "a tpyo typo")])
  (substring-fill! str 2 6 #\X)
  str)  ⇒  "a XXXX typo"
```

(string-truncate! *string n***)** **procedure**
returns: *string* or the empty string
libraries: `(chezscheme)`

string must be mutable. *n* must be an exact nonnegative fixnum not greater than the length of *string*. If *n* is zero, `string-truncate!` returns the empty string. Otherwise, *string-truncate!* destructively truncates *string* to its first *n* characters and returns *string*.

```
(define s (make-string 7 #\$))
(string-truncate! s 0)  ⇒  ""
s  ⇒  "$$$$$$$"
(string-truncate! s 3)  ⇒  "$$$"
s  ⇒  "$$$"
```

`(mutable-string?` *obj*`)` **procedure**
returns: `#t` if *obj* is a mutable string, `#f` otherwise
`(immutable-string?` *obj*`)` **procedure**
returns: `#t` if *obj* is an immutable string, `#f` otherwise
libraries: `(chezscheme)`

```
(mutable-string? (string #\a #\b #\c))  ⇒  #t
(mutable-string? (string->immutable-string "abc"))  ⇒  #f
(immutable-string? (string #\a #\b #\c))  ⇒  #f
(immutable-string? (string->immutable-string "abc"))  ⇒  #t
(immutable-string? (cons 3 4))  ⇒  #f
```

`(string->immutable-string` *string*`)` **procedure**
returns: an immutable string equal to *string*
libraries: `(chezscheme)`

The result is *string* itself if *string* is immutable; otherwise, the result is an immutable string with the same content as *string*.

```
(define s (string->immutable-string (string #\x #\y #\z)))
(string-set! s 0 #\a)  ⇒  exception: not mutable
```

7.5. Vectors

Chez Scheme extends the syntax of vectors to allow the length of the vector to be specified between the `#` and open parenthesis, e.g., `#3(a b c)`. If fewer elements are supplied in the syntax than the specified length, each element after the last printed element is the same as the last printed element. This extension is disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

The length and indices of a vector in *Chez Scheme* are always fixnums.

All vectors are mutable by default, including constants. A program can create immutable vectors via `vector->immutable-vector`. Any attempt to modify an immutable vector causes an exception to be raised.

`(vector-copy` *vector*`)` **procedure**
returns: a copy of *vector*
libraries: `(chezscheme)`

`vector-copy` creates a new vector of the same length and contents as *vector*. The elements themselves are not copied.

```
(vector-copy '#(a b c))  ⇒  #(a b c)
```

```
(let ([v '#(a b c)])
  (eq? v (vector-copy v)))  ⇒  #f
```

`(vector-set-fixnum!` *vector n fixnum*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

vector must be mutable. `vector-set-fixnum!` changes the *n*th element of *vector* to *fixnum*. *n* must be an exact nonnegative integer strictly less than the length of *vector*.

It is faster to store a fixnum than an arbitrary value, since for arbitrary values, the system has to record potential assignments from older to younger objects to support generational garbage collection. Care must be taken to ensure that the argument is indeed a fixnum, however; otherwise, the collector may not properly track the assignment. The primitive performs a fixnum check on the argument except at optimization level 3.

See also the description of fixnum-only vectors (fxvectors) below.

```
(let ([v (vector 1 2 3 4 5)])
  (vector-set-fixnum! v 2 73)
  v)  ⇒  #(1 2 73 4 5)
```

`(mutable-vector?` *obj*`)` **procedure**
returns: `#t` if *obj* is a mutable vector, `#f` otherwise
`(immutable-vector?` *obj*`)` **procedure**
returns: `#t` if *obj* is an immutable vector, `#f` otherwise
libraries: `(chezscheme)`

```
(mutable-vector? (vector 1 2 3))  ⇒  #t
(mutable-vector? (vector->immutable-vector (vector 1 2 3)))  ⇒  #f
(immutable-vector? (vector 1 2 3))  ⇒  #f
(immutable-vector? (vector->immutable-vector (vector 1 2 3)))  ⇒  #t
(immutable-vector? (cons 3 4))  ⇒  #f
```

`(vector->immutable-vector` *vector*`)` **procedure**
returns: an immutable vector equal to *vector*
libraries: `(chezscheme)`

The result is *vector* itself if *vector* is immutable; otherwise, the result is an immutable vector with the same content as *vector*.

```
(define v (vector->immutable-vector (vector 1 2 3)))
(vector-set! v 0 0)  ⇒  exception: not mutable
```

7.6. Fixnum-Only Vectors

Fixnum-only vectors, or "fxvectors," are like vectors but contain only fixnums. Fxvectors are written with the `#vfx` prefix in place of the `#` prefix for vectors, e.g., `#vfx(1 2 3)` or `#10vfx(2)`. The fxvector syntax is disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

The length and indices of an fxvector are always fixnums.

Updating an fxvector is generally less expensive than updating a vector, since for vectors, the system records potential assignments from older to younger objects to support generational garbage collection. The storage management system also takes advantage of the fact that fxvectors contain no pointers to place them in an area of memory that does not have to be traced during collection.

All fxvectors are mutable by default, including constants. A program can create immutable fxvectors via `fxvector->immutable-fxvector`. Any attempt to modify an immutable fxvector causes an exception to be raised.

See also `vector-set-fixnum!` above.

`(fxvector? `*obj*`)` **procedure**
returns: `#t` if *obj* is an fxvector, `#f` otherwise
libraries: `(chezscheme)`

```
(fxvector? #vfx())  ⇒  #t
(fxvector? #vfx(1 2 3))  ⇒  #t
(fxvector? (fxvector 1 2 3))  ⇒  #t
(fxvector? '#(a b c))  ⇒  #f
(fxvector? '(a b c))  ⇒  #f
(fxvector? "abc")  ⇒  #f
```

`(fxvector `*fixnum* `...)` **procedure**
returns: an fxvector of the fixnums *fixnum* `...`
libraries: `(chezscheme)`

```
(fxvector)  ⇒  #vfx()
(fxvector 1 3 5)  ⇒  #vfx(1 3 5)
```

`(make-fxvector `*n*`)` **procedure**
`(make-fxvector `*n* *fixnum*`)` **procedure**
returns: an fxvector of length *n*
libraries: `(chezscheme)`

n must be a fixnum. If *fixnum* is supplied, each element of the fxvector is initialized to *fixnum*; otherwise, the elements are unspecified.

```
(make-fxvector 0)  ⇒  #vfx()
(make-fxvector 0 7)  ⇒  #vfx()
(make-fxvector 5 7)  ⇒  #vfx(7 7 7 7 7)
```

`(fxvector-length `*fxvector*`)` **procedure**
returns: the number of elements in *fxvector*
libraries: `(chezscheme)`

```
(fxvector-length #vfx())  ⇒  0
(fxvector-length #vfx(1 2 3))  ⇒  3
(fxvector-length #10vfx(1 2 3))  ⇒  10
(fxvector-length (fxvector 1 2 3 4))  ⇒  4
(fxvector-length (make-fxvector 300))  ⇒  300
```

(`fxvector-ref` *fxvector n*) **procedure**
returns: the *n*th element (zero-based) of *fxvector*
libraries: `(chezscheme)`

n must be a nonnegative fixnum strictly less than the length of *fxvector*.

```
(fxvector-ref #vfx(-1 2 4 7) 0)  ⇒  -1
(fxvector-ref #vfx(-1 2 4 7) 1)  ⇒  2
(fxvector-ref #vfx(-1 2 4 7) 3)  ⇒  7
```

(`fxvector-set!` *fxvector n fixnum*) **procedure**
returns: unspecified
libraries: `(chezscheme)`

fxvector must be mutable. *n* must be a nonnegative fixnum strictly less than the length of *fxvector*. `fxvector-set!` changes the *n*th element of *fxvector* to *fixnum*.

```
(let ([v (fxvector 1 2 3 4 5)])
  (fxvector-set! v 2 (fx- (fxvector-ref v 2)))
  v)  ⇒  #vfx(1 2 -3 4 5)
```

(`fxvector-fill!` *fxvector fixnum*) **procedure**
returns: unspecified
libraries: `(chezscheme)`

fxvector must be mutable. `fxvector-fill!` replaces each element of *fxvector* with *fixnum*.

```
(let ([v (fxvector 1 2 3)])
  (fxvector-fill! v 0)
  v)  ⇒  #vfx(0 0 0)
```

(`fxvector->list` *fxvector*) **procedure**
returns: a list of the elements of *fxvector*
libraries: `(chezscheme)`

```
(fxvector->list (fxvector))  ⇒  ()
(fxvector->list #vfx(7 5 2))  ⇒  (7 5 2)

(let ([v #vfx(1 2 3 4 5)])
  (apply fx* (fxvector->list v)))  ⇒  120
```

`(list->fxvector` *list*`)` **procedure**
returns: an fxvector of the elements of *list*
libraries: `(chezscheme)`

list must consist entirely of fixnums.

```
(list->fxvector '())  ⇒  #vfx()
(list->fxvector '(3 5 7))  ⇒  #vfx(3 5 7)
```

```
(let ([v #vfx(1 2 3 4 5)])
  (let ([ls (fxvector->list v)])
    (list->fxvector (map fx* ls ls))))  ⇒  #vfx(1 4 9 16 25)
```

`(fxvector-copy` *fxvector*`)` **procedure**
returns: a copy of *fxvector*
libraries: `(chezscheme)`

`fxvector-copy` creates a new fxvector with the same length and contents as *fxvector*.

```
(fxvector-copy #vfx(3 4 5))  ⇒  #vfx(3 4 5)
```

```
(let ([v #vfx(3 4 5)])
  (eq? v (fxvector-copy v)))  ⇒  #f
```

`(mutable-fxvector?` *obj*`)` **procedure**
returns: `#t` if *obj* is a mutable fxvector, `#f` otherwise
`(immutable-fxvector?` *obj*`)` **procedure**
returns: `#t` if *obj* is an immutable fxvector, `#f` otherwise
libraries: `(chezscheme)`

```
(mutable-fxvector? (fxvector 1 2 3))  ⇒  #t
(mutable-fxvector? (fxvector->immutable-fxvector (fxvector 1 2 3)))  ⇒  #f
(immutable-fxvector? (fxvector 1 2 3))  ⇒  #f
(immutable-fxvector? (fxvector->immutable-fxvector (fxvector 1 2 3)))  ⇒  #t
(immutable-fxvector? (cons 3 4))  ⇒  #f
```

`(fxvector->immutable-fxvector` *fxvector*`)` **procedure**
returns: either an immutable copy of *fxvector* or *fxvector* itself
libraries: `(chezscheme)`

The result is *fxvector* itself if *fxvector* is immutable; otherwise, the result is an immutable fxvector with the same content as *fxvector*.

```
(define v (fxvector->immutable-fxvector (fxvector 1 2 3)))
(fxvector-set! v 0 0)  ⇒  exception: not mutable
```

7.7. Bytevectors

As with vectors, *Chez Scheme* extends the syntax of bytevectors to allow the length of the vector to be specified between the `#` and open parenthesis, e.g., `#3vu8(1 105 73)`. If fewer elements are supplied in the syntax than the specified length, each element after the last printed element is the same as the last printed element. This extension is disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

Chez Scheme also extends the set of bytevector primitives, including primitives for loading and storing 3, 5, 6, and 7-byte quantities.

The length and indices of a bytevector in *Chez Scheme* are always fixnums.

All bytevectors are mutable by default, including constants. A program can create immutable bytevectors via `bytevector->immutable-bytevector`. Any attempt to modify an immutable bytevector causes an exception to be raised.

`(bytevector` *fill* `...)` **procedure**
returns: a new bytevector containing *fill* `...`
libraries: `(chezscheme)`

Each *fill* value must be an exact integer representing a signed or unsigned 8-bit value, i.e., a value in the range -128 to 255 inclusive. A negative fill value is treated as its two's complement equivalent.

```
(bytevector)  ⇒  #vu8()
(bytevector 1 3 5)  ⇒  #vu8(1 3 5)
(bytevector -1 -3 -5)  ⇒  #vu8(255 253 251)
```

`(bytevector->s8-list` *bytevector*`)` **procedure**
returns: a new list of the 8-bit signed elements of *bytevector*
libraries: `(chezscheme)`

The values in the returned list are exact eight-bit signed integers, i.e., values in the range -128 to 127 inclusive. `bytevector->s8-list` is similar to the Revised[6] Report `bytevector->u8-list` except the values in the returned list are signed rather than unsigned.

```
(bytevector->s8-list (make-bytevector 0))  ⇒  ()
(bytevector->s8-list #vu8(1 127 128 255))  ⇒  (1 127 -128 -1)

(let ([v #vu8(1 2 3 255)])
  (apply * (bytevector->s8-list v)))  ⇒  -6
```

`(s8-list->bytevector` *list*`)` **procedure**
returns: a new bytevector of the elements of *list*
libraries: `(chezscheme)`

list must consist entirely of exact eight-bit signed integers, i.e., values in the range -128 to 127 inclusive. `s8-list->bytevector` is similar to the Revised[6] Report procedure `u8-list->bytevector`, except the elements of the input list are signed rather than unsigned.

```
(s8-list->bytevector '())  ⇒  #vu8()
(s8-list->bytevector '(1 127 -128 -1))  ⇒  #vu8(1 127 128 255)
```

```
(let ([v #vu8(1 2 3 4 5)])
  (let ([ls (bytevector->s8-list v)])
    (s8-list->bytevector (map - ls))))  ⇒  #vu8(255 254 253 252 251)
```

`(bytevector-truncate!` *bytevector n*`)` **procedure**
returns: *bytevector* or the empty bytevector
libraries: `(chezscheme)`

bytevector must be mutable. *n* must be an exact nonnegative fixnum not greater than the length of *bytevector*. If *n* is zero, `bytevector-truncate!` returns the empty bytevector. Otherwise, *bytevector-truncate!* destructively truncates *bytevector* to its first *n* bytes and returns *bytevector*.

```
(define bv (make-bytevector 7 19))
(bytevector-truncate! bv 0)  ⇒  #vu8()
bv  ⇒  #vu8(19 19 19 19 19 19 19)
(bytevector-truncate! bv 3)  ⇒  #vu8(19 19 19)
bv  ⇒  #vu8(19 19 19)
```

(bytevector-u24-ref *bytevector n eness*) **procedure**
returns: the 24-bit unsigned integer at index *n* (zero-based) of *bytevector*
(bytevector-s24-ref *bytevector n eness*) **procedure**
returns: the 24-bit signed integer at index *n* (zero-based) of *bytevector*
(bytevector-u40-ref *bytevector n eness*) **procedure**
returns: the 40-bit unsigned integer at index *n* (zero-based) of *bytevector*
(bytevector-s40-ref *bytevector n eness*) **procedure**
returns: the 40-bit signed integer at index *n* (zero-based) of *bytevector*
(bytevector-u48-ref *bytevector n eness*) **procedure**
returns: the 48-bit unsigned integer at index *n* (zero-based) of *bytevector*
(bytevector-s48-ref *bytevector n eness*) **procedure**
returns: the 48-bit signed integer at index *n* (zero-based) of *bytevector*
(bytevector-u56-ref *bytevector n eness*) **procedure**
returns: the 56-bit unsigned integer at index *n* (zero-based) of *bytevector*
(bytevector-s56-ref *bytevector n eness*) **procedure**
returns: the 56-bit signed integer at index *n* (zero-based) of *bytevector*
libraries: (chezscheme)

n must be an exact nonnegative integer and indexes the starting byte of the value. The sum of *n* and the number of bytes occupied by the value (3 for 24-bit values, 5 for 40-bit values, 6 for 48-bit values, and 7 for 56-bit values) must not exceed the length of *bytevector*. *eness* must be a valid endianness symbol naming the endianness.

The return value is an exact integer in the appropriate range for the number of bytes occupied by the value. Signed values are the equivalent of the stored value treated as a two's complement value.

(bytevector-u24-set! *bytevector n u24 eness*) **procedure**
(bytevector-s24-set! *bytevector n s24 eness*) **procedure**
(bytevector-u40-set! *bytevector n u40 eness*) **procedure**
(bytevector-s40-set! *bytevector n s40 eness*) **procedure**
(bytevector-u48-set! *bytevector n u48 eness*) **procedure**
(bytevector-s48-set! *bytevector n s48 eness*) **procedure**
(bytevector-u56-set! *bytevector n u56 eness*) **procedure**
(bytevector-s56-set! *bytevector n s56 eness*) **procedure**
returns: unspecified
libraries: (chezscheme)

bytevector must be mutable. *n* must be an exact nonnegative integer and indexes the starting byte of the value. The sum of *n* and the number of bytes occupied by the value must not exceed the length of *bytevector*. *u24* must be a 24-bit unsigned value, i.e., a value in the range 0 to $2^{24} - 1$ inclusive; *s24* must be a 24-bit signed value, i.e., a value in the range -2^{23} to $2^{23} - 1$ inclusive; *u40* must be a 40-bit unsigned value, i.e., a value in the range 0 to $2^{40} - 1$ inclusive; *s40* must be a 40-bit signed value, i.e., a value in the range -2^{39} to $2^{39} - 1$ inclusive; *u48* must be a 48-bit unsigned value, i.e., a value in the range 0 to $2^{48} - 1$ inclusive; *s48* must be a 48-bit signed value, i.e., a value in the range -2^{47} to $2^{47} - 1$ inclusive; *u56* must be a 56-bit unsigned value, i.e., a value in the range 0 to $2^{56} - 1$

inclusive; and *s56* must be a 56-bit signed value, i.e., a value in the range -2^{55} to $2^{55}-1$ inclusive. *eness* must be a valid endianness symbol naming the endianness.

These procedures store the given value in the 3, 5, 6, or 7 bytes starting at index n (zero-based) of *bytevector*. Negative values are stored as their two's complement equivalent.

`(mutable-bytevector?` *obj*`)` **procedure**
returns: `#t` if *obj* is a mutable bytevector, `#f` otherwise

`(immutable-bytevector?` *obj*`)` **procedure**
returns: `#t` if *obj* is an immutable bytevector, `#f` otherwise
libraries: `(chezscheme)`

```
(mutable-bytevector? (bytevector 1 2 3))  ⇒  #t
(mutable-bytevector?
  (bytevector->immutable-bytevector (bytevector 1 2 3)))  ⇒  #f
(immutable-bytevector? (bytevector 1 2 3))  ⇒  #f
(immutable-bytevector?
  (bytevector->immutable-bytevector (bytevector 1 2 3)))  ⇒  #t
(immutable-bytevector? (cons 3 4))  ⇒  #f
```

`(bytevector->immutable-bytevector` *bytevector*`)` **procedure**
returns: an immutable bytevector equal to *bytevector*
libraries: `(chezscheme)`

The result is *bytevector* itself if *bytevector* is immutable; otherwise, the result is an immutable bytevector with the same content as *bytevector*.

```
(define bv (bytevector->immutable-bytevector (bytevector 1 2 3)))
(bytevector-u8-set! bv 0 0)  ⇒  exception: not mutable
```

`(bytevector-compress` *bytevector*`)` **procedure**
returns: a new bytevector containing compressed content of *bytevector*
libraries: `(chezscheme)`

The result is the raw compressed data with a minimal header to record the uncompressed size and the compression mode. The result does not include the header that is written by port-based compression using the `compressed` option.

`(bytevector-uncompress` *bytevector*`)` **procedure**
returns: a bytevector containing uncompressed content of *bytevector*
libraries: `(chezscheme)`

Uncompresses a *bytevector* produced by `bytevector-compress` to a new bytevector with the same content as the original given to `bytevector-compress`.

7.8. Boxes

Boxes are single-cell objects that are primarily useful for providing an "extra level of indirection." This extra level of indirection is typically used to allow more than one body of code or data structure to share a reference, or pointer, to an object. For example, boxes may be used to implement *call-by-reference* semantics in an interpreter for a language employing this parameter passing discipline.

Boxes are written with the prefix `#&` (pronounced "hash-ampersand"). For example, `#&(a b c)` is a box holding the list `(a b c)`. The box syntax is disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

All boxes are mutable by default, including constants. A program can create immutable boxes via `box-immutable`. Any attempt to modify an immutable box causes an exception to be raised.

`(box? `*obj*`)` **procedure**
returns: `#t` if *obj* is a box, `#f` otherwise
libraries: `(chezscheme)`

```
(box? '#&a)  ⇒  #t
(box? 'a)  ⇒  #f
(box? (box 3))  ⇒  #t
```

`(box `*obj*`)` **procedure**
returns: a new box containing *obj*
libraries: `(chezscheme)`

```
(box 'a)  ⇒  #&a
(box (box '(a b c)))  ⇒  #&#&(a b c)
```

`(unbox `*box*`)` **procedure**
returns: contents of *box*
libraries: `(chezscheme)`

```
(unbox #&a)  ⇒  a
(unbox #&#&(a b c))  ⇒  #&(a b c)

(let ([b (box "hi")])
  (unbox b))  ⇒  "hi"
```

`(set-box! `*box obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

box must be mutable. `set-box!` sets the contents of *box* to *obj*.

```
(let ([b (box 'x)])
  (set-box! b 'y)
  b)  ⇒  #&y

(let ([incr!
       (lambda (x)
         (set-box! x (+ (unbox x) 1)))])
  (let ([b (box 3)])
    (incr! b)
    (unbox b)))  ⇒  4
```

`(mutable-box?` *obj*`)` **procedure**
returns: `#t` if *obj* is a mutable box, `#f` otherwise
`(immutable-box?` *obj*`)` **procedure**
returns: `#t` if *obj* is an immutable box, `#f` otherwise
libraries: `(chezscheme)`

```
(mutable-box? (box 1))  ⇒  #t
(mutable-box? (box-immutable 1))  ⇒  #f
(immutable-box? (box 1))  ⇒  #f
(immutable-box? (box-immutable 1))  ⇒  #t
(mutable-box? (cons 3 4))  ⇒  #f
```

`(box-immutable` *obj*`)` **procedure**
returns: a new immutable box containing *obj*
libraries: `(chezscheme)`

Boxes are typically intended to support shared, mutable structure, so immutable boxes are not often useful.

```
(define b (box-immutable 1))
(set-box! b 0)  ⇒  exception: not mutable
```

7.9. Symbols

Chez Scheme extends the standard symbol syntax in several ways:

- Symbol names may begin with `@`, but `,@abc` is parsed as `(unquote-splicing abc)`; to produce `(unquote @abc)` one can type `, @abc`, `\x40;abc`, or `,|@abc|`.
- The single-character sequences `{` and `}` are read as symbols.
- A symbol's name may begin with any character that might normally start a number, including a digit, `.`, `+`, `-`, as long as the delimited sequence of characters starting with that character cannot be parsed as a number.
- A symbol whose name contains arbitrary characters may be written by escaping them with `\` or with `|`. `\` is used to escape a single character (except 'x', since `\x` marks

the start of a hex scalar value), whereas `|` is used to escape the group of characters that follow it up through the matching `|`.

The printer always prints symbols using the standard R6RS syntax, so that, e.g., `@abc` prints as `\x40;abc` and `1-` prints as `\x31;-`. ’

Gensyms are printed `#{` and `}` brackets that enclose both the “pretty” and “unique” names, e.g., `#{g1426 e5g1c94g642dssw-a}`. They may also be printed using the pretty name only with the prefix `#:`, e.g., `#:g1426`.

These extensions are disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

`(gensym)` **procedure**
`(gensym` *pretty-name*`)` **procedure**
`(gensym` *pretty-name unique-name*`)` **procedure**
returns: a unique generated symbol
libraries: `(chezscheme)`

Each call to `gensym` returns a unique generated symbol, or *gensym*. Each generated symbol has two names: a “pretty” name and a “unique” name.

In the first form above, the pretty name is formed (lazily—see below) by combining an internal prefix with the value of an internal counter. After each name is formed, the internal counter is incremented. The parameters `gensym-prefix` and `gensym-count`, described below, may be used to access and set the internal prefix and counter. By default, the prefix is the single-character string `"g"`. In the second and third forms, the pretty name of the new gensym is *pretty-name*, which must be a string. The pretty name of a gensym is returned by the procedure `symbol->string`.

In both the first and second forms, the unique name is an automatically generated globally unique name. Globally unique names are constructed (lazily—see below) from some combination of a unique machine identifier (such as the network address), the current process identifier (PID), and the time at which the Scheme session began, along with an internal counter. In the third form of gensym, the unique name of the new gensym is *unique-name*, which must be a string. The unique name of a gensym may be obtained via the procedure `gensym->unique-string`.

The unique name allows gensyms to be written in such a way that they can be read back and reliably commonized on input. The syntax for gensyms includes both the pretty name and the unique name, as shown in the example below:

```
(gensym)  ⇒  #{g0 bcsfg5eq4e9b3h9o-a}
```

When the parameter `print-gensym` is set to `pretty`, the printer prints the pretty name only, with a `#:` syntax, so

```
(parameterize ([print-gensym 'pretty])
  (write (gensym)))
```

prints `#:g0`.

When the reader sees the `#:` syntax, it produces a gensym with the given pretty name, but the original unique name is lost.

When the parameter is set to `#f`, the printer prints just the pretty name, so

```
(parameterize ([print-gensym #f])
  (write (gensym)))
```

prints `g0`. This is useful only when gensyms do not need to be read back in as gensyms.

In order to reduce construction and (when threaded) synchronization overhead when gensyms are frequently created but rarely printed or stored in an object file, generated pretty and unique names are created lazily, i.e., not until first requested, either by the printer, fasl writer, or explicitly by one of the procedures `symbol->string` or `gensym->unique-string`. In addition, a gensym is not placed into the system's internal symbol table (the oblist; see page 155) until the unique name is requested. This allows a gensym to be reclaimed by the storage manager if no references to the gensym exist and no unique name exists by which to access it, even if it has a top-level binding or a nonempty property list.

```
(define x (gensym))
x                          ⇒  #{g2 bcsfg5eq4e9b3h9o-c}
(symbol->string x)         ⇒  "g2"
(gensym->unique-string x)  ⇒  "bcsfg5eq4e9b3h9o-c"
```

Gensyms subsume the notion of *uninterned symbols* supported by earlier versions of *Chez Scheme*. Similarly, the predicate `uninterned-symbol?` has been replaced by `gensym?`.

`gensym-prefix` **thread parameter**
`gensym-count` **thread parameter**
libraries: `(chezscheme)`

The parameters `gensym-prefix` and `gensym-count` are used to access and set the internal prefix and counter from which the pretty name of a gensym is generated when `gensym` is not given an explicit string argument. `gensym-prefix` defaults to the string `"g"` and may be set to any object. `gensym-count` starts at 0 and may be set to any nonnegative integer.

As described above, *Chez Scheme* delays the creation of the pretty name until the name is first requested by the printer or by an explicit call to `symbol->string`. These parameters are not consulted until that time; setting them when `gensym` is called thus has no effect on the generated name.

```
(let ([x (parameterize ([gensym-prefix "genny"]
                        [gensym-count 17]
                        [print-gensym 'pretty])
           (gensym))])
  (format "~s" x))                        ⇒  "#{g4 bcsfg5eq4e9b3h9o-e}"
(let ([x (gensym)])
  (parameterize ([gensym-prefix "genny"]
                 [gensym-count 17]
```

```
                [print-gensym #f])
    (format "~s" (gensym))))              ⇒  "genny17"
```

(gensym->unique-string *gensym*) **procedure**
returns: the unique name of *gensym*
libraries: (chezscheme)

```
(gensym->unique-string (gensym))  ⇒  "bd3kufa7ypjcuvut-g"
```

(gensym? *obj*) **procedure**
returns: #t if *obj* is gensym, #f otherwise
libraries: (chezscheme)

```
(gensym? (string->symbol "z"))  ⇒  #f
(gensym? (gensym "z"))  ⇒  #t
(gensym? 'a)  ⇒  #f
(gensym? 3)  ⇒  #f
(gensym? (gensym))  ⇒  #t
(gensym? '#{g2 bcsfg5eq4e9b3h9o-c})  ⇒  #t
```

(putprop *symbol key value*) **procedure**
returns: unspecified
libraries: (chezscheme)

Chez Scheme associates a *property list* with each symbol, allowing multiple *key-value* pairs to be stored directly with the symbol. New key-value pairs may be placed in the property list or retrieved in a manner analogous to the use of association lists, using the procedures putprop and getprop. Property lists are often used to store information related to the symbol itself. For example, a natural language program might use symbols to represent words, using their property lists to store information about use and meaning.

putprop associates *value* with *key* on the property list of *symbol*. *key* and *value* may be any types of object, although *key* is typically a symbol.

putprop may be used to establish a new property or to change an existing property.

See the examples under getprop below.

(getprop *symbol key*) **procedure**
(getprop *symbol key default*) **procedure**
returns: the value associated with *key* on the property list of *symbol*
libraries: (chezscheme)

getprop searches the property list of *symbol* for a key identical to *key* (in the sense of eq?), and returns the value associated with this key, if any. If no value is associated with *key* on the property list of *symbol*, getprop returns *default*, or #f if the *default* argument is not supplied.

```
(putprop 'fred 'species 'snurd)
(putprop 'fred 'age 4)
(putprop 'fred 'colors '(black white))

(getprop 'fred 'species)  ⇒  snurd
(getprop 'fred 'colors)  ⇒  (black white)
(getprop 'fred 'nonkey)  ⇒  #f
(getprop 'fred 'nonkey 'unknown)  ⇒  unknown

(putprop 'fred 'species #f)
(getprop 'fred 'species 'unknown)  ⇒  #f
```

(remprop *symbol key*) **procedure**
returns: unspecified
libraries: (chezscheme)

`remprop` removes the property with key *key* from the property list of *symbol*, if such a property exists.

```
(putprop 'fred 'species 'snurd)
(getprop 'fred 'species)  ⇒  snurd

(remprop 'fred 'species)
(getprop 'fred 'species 'unknown)  ⇒  unknown
```

(property-list *symbol*) **procedure**
returns: a copy of the internal property list for *symbol*
libraries: (chezscheme)

A property list is a list of alternating keys and values, i.e., (*key value* ...).

```
(putprop 'fred 'species 'snurd)
(putprop 'fred 'colors '(black white))
(property-list 'fred)  ⇒  (colors (black white) species snurd)
```

(oblist) **procedure**
returns: a list of interned symbols
libraries: (chezscheme)

The system maintains an internal symbol table used to insure that any two occurrences of the same symbol name resolve to the same symbol object. The `oblist` procedure returns a list of the symbols currently in this symbol table.

The list of interned symbols grows when a new symbol is introduced into the system or when the unique name of a gensym (see page 152) is requested. It shrinks when the garbage collector determines that it is safe to discard a symbol. It is safe to discard a symbol only if the symbol is not accessible except through the oblist, has no top-level binding, and has no properties on its property list.

```
(if (memq 'tiger (oblist)) 'yes 'no)  ⇒  yes
(equal? (oblist) (oblist))  ⇒  #t
(= (length (oblist)) (length (oblist)))  ⇒  #t or #f
```

The first example above follows from the property that all interned symbols are in the oblist from the time they are read, which happens prior to evaluation. The second example follows from the fact that no symbols can be removed from the oblist while references to those symbols exist, in this case, within the list returned by the first call to `oblist` (whichever call is performed first). The expression in the third example can return `#f` only if a garbage collection occurs sometime between the two calls to `oblist`, and only if one or more symbols are removed from the oblist by that collection.

7.10. Void

Many Scheme operations return an unspecified result. *Chez Scheme* typically returns a special *void* object when the value returned by an operation is unspecified. The *Chez Scheme* void object is not meant to be used as a datum, and consequently does not have a reader syntax. As for other objects without a reader syntax, such as procedures and ports, *Chez Scheme* output procedures print the void object using a nonreadable representation, i.e., `#<void>`. Since the void object should be returned only by operations that do not have "interesting" values, the default waiter printer (see `waiter-write`) suppresses the printing of the void object. `set!`, `set-car!`, `load`, and `write` are examples of *Chez Scheme* operations that return the void object.

`(void)` **procedure**
returns: the void object
libraries: `(chezscheme)`

`void` is a procedure of no arguments that returns the void object. It can be used to force expressions that are used for effect or whose values are otherwise unspecified to evaluate to a consistent, trivial value. Since most *Chez Scheme* operations that are used for effect return the void object, however, it is rarely necessary to explicitly invoke the `void` procedure.

Since the void object is used explicitly as an "unspecified" value, it is a bad idea to use it for any other purpose or to count on any given expression evaluating to the void object.

The default waiter printer suppresses the void object; that is, nothing is printed for expressions that evaluate to the void object.

```
(eq? (void) #f)  ⇒  #f
(eq? (void) #t)  ⇒  #f
(eq? (void) '())  ⇒  #f
```

7.11. Sorting

(sort *predicate list*) **procedure**
(sort! *predicate list*) **procedure**
returns: a list containing the elements of *list* sorted according to *predicate*
libraries: (chezscheme)

`sort` is identical to the Revised[6] Report `list-sort`, and `sort!` is a destructive version of `sort`, i.e., it reuses pairs from the input list to form the output list.

```
(sort < '(3 4 2 1 2 5))  ⇒  (1 2 2 3 4 5)
(sort! < '(3 4 2 1 2 5))  ⇒  (1 2 2 3 4 5)
```

(merge *predicate* $list_1$ $list_2$) **procedure**
(merge! *predicate* $list_1$ $list_2$) **procedure**
returns: $list_1$ merged with $list_2$ in the order specified by *predicate*
libraries: (chezscheme)

predicate should be a procedure that expects two arguments and returns `#t` if its first argument must precede its second in the merged list. It should not have any side effects. That is, if *predicate* is applied to two objects x and y, where x is taken from the second list and y is taken from the first list, it should return true only if x should appear before y in the output list. If this constraint is met, `merge` and `merge!` are stable, in that items from $list_1$ are placed in front of equivalent items from $list_2$ in the output list. Duplicate elements are included in the merged list.

`merge!` combines the lists destructively, using pairs from the input lists to form the output list.

```
(merge char<?
       '(#\a #\c)
       '(#\b #\c #\d))  ⇒  (#\a #\b #\c #\c #\d)
(merge <
       '(1/2 2/3 3/4)
       '(0.5 0.6 0.7))  ⇒  (1/2 0.5 0.6 2/3 0.7 3/4)
```

7.12. Hashtables

Chez Scheme provides several extensions to the hashtable mechanism, including a mechanism for directly accessing a key, value pair in a hashtable, support for weak eq and eqv hashtables, and a set of procedures specialized to eq and symbol hashtables.

(hashtable-cell *hashtable key default*) **procedure**
returns: a pair (see below)
libraries: (chezscheme)

hashtable must be a hashtable. *key* and *default* may be any Scheme values.

If no value is associated with *key* in *hashtable*, `hashtable-cell` modifies *hashtable* to associate *key* with *default*. It returns a pair whose car is *key* and whose cdr is the associated value. Changing the cdr of this pair effectively updates the table to associate *key* with a new value. The *key* in the car field should not be changed. The advantage of this procedure over the Revised[6] Report procedures for manipulating hashtable entries is that the value associated with a key may be read or written many times with only a single hashtable lookup.

```
(define ht (make-eq-hashtable))
(define v (vector 'a 'b 'c))
(define cell (hashtable-cell ht v 3))
cell  ⇒  (#(a b c) . 3)
(hashtable-ref ht v 0)  ⇒  3
(set-cdr! cell 4)
(hashtable-ref ht v 0)  ⇒  4
```

`(hashtable-values` *hashtable*`)` **procedure**

returns: a vector containing the values in *hashtable*
libraries: `(chezscheme)`

Each value is the value of one of the keys in *hashtable*. Duplicate values are not removed. The values may appear in any order in the returned vector.

```
(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 "one")
(hashtable-set! ht p2 "two")
(hashtable-set! ht 'q "two")
(hashtable-values ht)  ⇒  #("one" "two" "two")
```

This procedure is equivalent to:

```
(lambda (ht)
  (let-values ([(keys values) (hashtable-entries ht)])
    values))
```

but more efficient since the separate vector of keys need not be created.

`(make-weak-eq-hashtable)` **procedure**
`(make-weak-eq-hashtable` *size*`)` **procedure**
`(make-weak-eqv-hashtable)` **procedure**
`(make-weak-eqv-hashtable` *size*`)` **procedure**

returns: a new weak eq hashtable
libraries: `(chezscheme)`

These procedures are like the Revised[6] Report procedures `make-eq-hashtable` and `make-eqv-hashtable` except the keys of the hashtable are held weakly, i.e., they are not

protected from the garbage collector. Keys reclaimed by the garbage collector are removed from the table, and their associated values are dropped the next time the table is modified, if not sooner.

Values in the hashtable are referenced normally as long as the key is not reclaimed, since keys are paired values using weak pairs. Consequently, if a value in the hashtable refers to its own key, then garbage collection is prevented from reclaiming the key. See `make-ephemeron-eq-hashtable` and `make-ephemeron-eqv-hashtable`.

A copy of a weak eq or eqv hashtable created by `hashtable-copy` is also weak. If the copy is immutable, inaccessible keys may still be dropped from the hashtable, even though the contents of the table is otherwise unchanging. The effect of this can be observed via `hashtable-keys` and `hashtable-entries`.

```
(define ht1 (make-weak-eq-hashtable))
(define ht2 (make-weak-eq-hashtable 32))
```

`(make-ephemeron-eq-hashtable)` **procedure**
`(make-ephemeron-eq-hashtable` *size*`)` **procedure**
`(make-ephemeron-eqv-hashtable)` **procedure**
`(make-ephemeron-eqv-hashtable` *size*`)` **procedure**
returns: a new ephemeron eq hashtable
libraries: `(chezscheme)`

These procedures are like `make-weak-eq-hashtable` and `make-weak-eqv-hashtable`, but a value in the hashtable can refer to a key in the hashtable (directly or indirectly) without preventing garbage collection from reclaiming the key, because keys are paired with values using ephemeron pairs.

A copy of an ephemeron eq or eqv hashtable created by `hashtable-copy` is also an ephemeron table, and an inaccesible key can be dropped from an immutable ephemeron hashtable in the same way as for an immutable weak hashtable.

```
(define ht1 (make-ephemeron-eq-hashtable))
(define ht2 (make-ephemeron-eq-hashtable 32))
```

`(hashtable-weak?` *obj*`)` **procedure**
returns: `#t` if *obj* is a weak eq or eqv hashtable, `#f` otherwise
libraries: `(chezscheme)`

```
(define ht1 (make-weak-eq-hashtable))
(define ht2 (hashtable-copy ht1))
(hashtable-weak? ht2)  ⇒  #t
```

`(hashtable-ephemeron?` *obj*`)` **procedure**
returns: `#t` if *obj* is an ephemeron eq or eqv hashtable, `#f` otherwise
libraries: `(chezscheme)`

```
(define ht1 (make-ephemeron-eq-hashtable))
(define ht2 (hashtable-copy ht1))
(hashtable-ephemeron? ht2)  ⇒  #t
```

`(eq-hashtable?` *obj*`)` **procedure**
returns: `#t` if *obj* is an eq hashtable, `#f` otherwise
libraries: `(chezscheme)`

```
(eq-hashtable? (make-eq-hashtable))  ⇒  #t
(eq-hashtable? '(not a hash table))  ⇒  #f
```

`(eq-hashtable-weak?` *hashtable*`)` **procedure**
returns: `#t` if *hashtable* is weak, `#f` otherwise
libraries: `(chezscheme)`

hashtable must be an eq hashtable.

```
(eq-hashtable-weak? (make-eq-hashtable))  ⇒  #f
(eq-hashtable-weak? (make-weak-eq-hashtable))  ⇒  #t
```

`(eq-hashtable-ephemeron?` *hashtable*`)` **procedure**
returns: `#t` if *hashtable* uses ephemeron pairs, `#f` otherwise
libraries: `(chezscheme)`

hashtable must be an eq hashtable.

```
(eq-hashtable-ephemeron? (make-eq-hashtable))  ⇒  #f
(eq-hashtable-ephemeron? (make-ephemeron-eq-hashtable))  ⇒  #t
```

`(eq-hashtable-set!` *hashtable key value*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

hashtable must be a mutable eq hashtable. *key* and *value* may be any Scheme values.

`eq-hashtable-set!` associates the value *value* with the key *key* in *hashtable*.

```
(define ht (make-eq-hashtable))
(eq-hashtable-set! ht 'a 73)
```

`(eq-hashtable-ref` *hashtable key default*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

hashtable must be an eq hashtable. *key* and *default* may be any Scheme values.

`eq-hashtable-ref` returns the value associated with *key* in *hashtable*. If no value is associated with *key* in *hashtable*, `eq-hashtable-ref` returns *default*.

```
(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(eq-hashtable-set! ht p1 73)
(eq-hashtable-ref ht p1 55)  ⇒  73
(eq-hashtable-ref ht p2 55)  ⇒  55
```

`(eq-hashtable-contains?` *hashtable key*`)` **procedure**
returns: `#t` if an association for *key* exists in *hashtable*, `#f` otherwise
libraries: `(chezscheme)`

hashtable must be an eq hashtable. *key* may be any Scheme value.

```
(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(eq-hashtable-set! ht p1 73)
(eq-hashtable-contains? ht p1)  ⇒  #t
(eq-hashtable-contains? ht p2)  ⇒  #f
```

`(eq-hashtable-update!` *hashtable key procedure default*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

hashtable must be a mutable eq hashtable. *key* and *default* may be any Scheme values. *procedure* should accept one argument, should return one value, and should not modify *hashtable*.

`eq-hashtable-update!` applies *procedure* to the value associated with *key* in *hashtable*, or to *default* if no value is associated with *key* in *hashtable*. If *procedure* returns, `eq-hashtable-update!` associates *key* with the value returned by *procedure*, replacing the old association, if any.

A version of `eq-hashtable-update!` that does not verify that it receives arguments of the proper type might be defined as follows.

```
(define eq-hashtable-update!
  (lambda (ht key proc value)
    (eq-hashtable-set! ht key
      (proc (eq-hashtable-ref ht key value)))))
```

An implementation may, however, be able to implement `eq-hashtable-update!` more efficiently by avoiding multiple hash computations and hashtable lookups.

```
(define ht (make-eq-hashtable))
(eq-hashtable-update! ht 'a
  (lambda (x) (* x 2))
  55)
(eq-hashtable-ref ht 'a 0)  ⇒  110
(eq-hashtable-update! ht 'a
  (lambda (x) (* x 2))
  0)
(eq-hashtable-ref ht 'a 0)  ⇒  220
```

`(eq-hashtable-cell` *hashtable key default*`)` **procedure**
returns: a pair (see below)
libraries: `(chezscheme)`

hashtable must be an eq hashtable. *key* and *default* may be any Scheme values.

If no value is associated with *key* in *hashtable*, `eq-hashtable-cell` modifies *hashtable* to associate *key* with *default*. It returns a pair whose car is *key* and whose cdr is the associated value. Changing the cdr of this pair effectively updates the table to associate *key* with a new value. The *key* should not be changed.

```
(define ht (make-eq-hashtable))
(define v (vector 'a 'b 'c))
(define cell (eq-hashtable-cell ht v 3))
cell  ⇒  (#(a b c) . 3)
(eq-hashtable-ref ht v 0)  ⇒  3
(set-cdr! cell 4)
(eq-hashtable-ref ht v 0)  ⇒  4
```

`(eq-hashtable-delete!` *hashtable key*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

hashtable must be a mutable eq hashtable. *key* may be any Scheme value.

`eq-hashtable-delete!` drops any association for *key* from *hashtable*.

```
(define ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(eq-hashtable-set! ht p1 73)
(eq-hashtable-contains? ht p1)  ⇒  #t
(eq-hashtable-delete! ht p1)
(eq-hashtable-contains? ht p1)  ⇒  #f
(eq-hashtable-contains? ht p2)  ⇒  #f
(eq-hashtable-delete! ht p2)
```

`(symbol-hashtable?` *obj*`)` **procedure**
returns: `#t` if *obj* is an eq hashtable, `#f` otherwise
libraries: `(chezscheme)`

```
(symbol-hashtable? (make-hashtable symbol-hash eq?))  ⇒  #t
(symbol-hashtable? (make-eq-hashtable))  ⇒  #f
```

`(symbol-hashtable-set!` *hashtable key value*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

hashtable must be a mutable symbol hashtable. (A symbol hashtable is a hashtable created with hash function `symbol-hash` and equivalence function `eq?`, `eqv?`, `equal?`, or `symbol=?`.) *key* must be a symbol, and *value* may be any Scheme value.

`symbol-hashtable-set!` associates the value *value* with the key *key* in *hashtable*.

```
(define ht (make-hashtable symbol-hash eq?))
(symbol-hashtable-ref ht 'a #f)  ⇒  #f
(symbol-hashtable-set! ht 'a 73)
(symbol-hashtable-ref ht 'a #f)  ⇒  73
```

`(symbol-hashtable-ref` *hashtable key default*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

hashtable must be a symbol hashtable. (A symbol hashtable is a hashtable created with hash function `symbol-hash` and equivalence function `eq?`, `eqv?`, `equal?`, or `symbol=?`.) *key* must be a symbol, and *default* may be any Scheme value.

`symbol-hashtable-ref` returns the value associated with *key* in *hashtable*. If no value is associated with *key* in *hashtable*, `symbol-hashtable-ref` returns *default*.

```
(define ht (make-hashtable symbol-hash eq?))
(define k1 'abcd)
(define k2 'not-abcd)
(symbol-hashtable-set! ht k1 "hi")
(symbol-hashtable-ref ht k1 "bye")  ⇒  "hi"
(symbol-hashtable-ref ht k2 "bye")  ⇒  "bye"
```

`(symbol-hashtable-contains?` *hashtable key*`)` **procedure**
returns: `#t` if an association for *key* exists in *hashtable*, `#f` otherwise
libraries: `(chezscheme)`

hashtable must be a symbol hashtable. (A symbol hashtable is a hashtable created with hash function `symbol-hash` and equivalence function `eq?`, `eqv?`, `equal?`, or `symbol=?`.) *key* must be a symbol.

```
(define ht (make-hashtable symbol-hash eq?))
(define k1 'abcd)
(define k2 'not-abcd)
(symbol-hashtable-set! ht k1 "hi")
(symbol-hashtable-contains? ht k1)  ⇒  #t
(symbol-hashtable-contains? ht k2 )  ⇒  #f
```

`(symbol-hashtable-update!` *hashtable key procedure default*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

hashtable must be a mutable symbol hashtable. (A symbol hashtable is a hashtable created with hash function `symbol-hash` and equivalence function `eq?`, `eqv?`, `equal?`, or `symbol=?`.) *key* must be a symbol, and *default* may be any Scheme value. *procedure* should accept one argument, should return one value, and should not modify *hashtable*.

`symbol-hashtable-update!` applies *procedure* to the value associated with *key* in *hashtable*, or to *default* if no value is associated with *key* in *hashtable*. If *procedure* returns, `symbol-hashtable-update!` associates *key* with the value returned by *procedure*, replacing the old association, if any.

A version of `symbol-hashtable-update!` that does not verify that it receives arguments of the proper type might be defined as follows.

```
(define symbol-hashtable-update!
  (lambda (ht key proc value)
    (symbol-hashtable-set! ht key
      (proc (symbol-hashtable-ref ht key value)))))
```

An implementation may, however, be able to implement `symbol-hashtable-update!` more efficiently by avoiding multiple hash computations and hashtable lookups.

```
(define ht (make-hashtable symbol-hash eq?))
(symbol-hashtable-update! ht 'a
  (lambda (x) (* x 2))
  55)
(symbol-hashtable-ref ht 'a 0)  ⇒  110
(symbol-hashtable-update! ht 'a
  (lambda (x) (* x 2))
  0)
(symbol-hashtable-ref ht 'a 0)  ⇒  220
```

`(symbol-hashtable-cell` *hashtable key default*`)` **procedure**
returns: a pair (see below)
libraries: `(chezscheme)`

hashtable must be a mutable symbol hashtable. (A symbol hashtable is a hashtable created with hash function `symbol-hash` and equivalence function `eq?`, `eqv?`, `equal?`, or `symbol=?`.) *key* must be a symbol, and *default* may be any Scheme value.

If no value is associated with *key* in *hashtable*, `symbol-hashtable-cell` modifies *hashtable* to associate *key* with *default*. It returns a pair whose car is *key* and whose cdr is the associated value. Changing the cdr of this pair effectively updates the table to associate *key* with a new value. The *key* should not be changed.

```
(define ht (make-hashtable symbol-hash eq?))
(define k 'a-key)
(define cell (symbol-hashtable-cell ht k 3))
cell  ⇒  (a-key . 3)
(symbol-hashtable-ref ht k 0)  ⇒  3
(set-cdr! cell 4)
(symbol-hashtable-ref ht k 0)  ⇒  4
```

`(symbol-hashtable-delete!` *hashtable key*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

hashtable must be a mutable symbol hashtable. (A symbol hashtable is a hashtable created with hash function `symbol-hash` and equivalence function `eq?`, `eqv?`, `equal?`, or `symbol=?`.) *key* must be a symbol.

`symbol-hashtable-delete!` drops any association for *key* from *hashtable*.

```
(define ht (make-hashtable symbol-hash eq?))
(define k1 (gensym))
(define k2 (gensym))
(symbol-hashtable-set! ht k1 73)
(symbol-hashtable-contains? ht k1)  ⇒  #t
(symbol-hashtable-delete! ht k1)
(symbol-hashtable-contains? ht k1)  ⇒  #f
(symbol-hashtable-contains? ht k2)  ⇒  #f
(symbol-hashtable-delete! ht k2)
```

7.13. Record Types

Chez Scheme extends the Revised[6] Report's `define-record-type` syntax in one way, which is that it allows a generative record type to be declared explicitly as such (in a double-negative sort of way) by including a `nongenerative` clause with `#f` as the uid, i.e.:

```
(nongenerative #f)
```

This can be used in conjunction with the parameter `require-nongenerative-clause` to catch the accidental use of generative record types while avoiding spurious errors for record types that must be generative. Generative record types are rarely needed and are generally less efficient since a run-time representation of the type is created each time the `define-record-clause` is evaluated, rather than once at compile (expansion) time.

`require-nongenerative-clause` **thread parameter**
libraries: `(chezscheme)`

This parameter holds a boolean value that determines whether `define-record-type` requires a nongenerative clause. The default value is `#f`. The lead-in above describes why one might want to set this to `#t`.

7.14. Record Equality and Hashing

By default, the `equal?` primitive compares record instances using `eq?`, i.e., it distinguishes non-eq? instances even if they are of the same type and have equal contents. A program can override this behavior for instances of a record type (and its subtypes that do not have their own equality procedures) by using `record-type-equal-procedure` to associate an equality procedure with the record-type descriptor (*rtd*) that describes the record type.

When comparing two eq? instances, `equal?` always returns `#t`. When comparing two non-eq? instances that share an equality procedure *equal-proc*, `equal?` uses *equal-proc* to compare the instances. Two instances x and y share an equality procedure if they inherit an equality procedure from the same point in the inheritance chain, i.e., if `(record-equal-procedure` x y`)` returns a procedure (*equal-proc*) rather than `#f`. *equal?* passes *equal-proc* three arguments: the two instances plus a *eql?* procedure that should be used for recursive comparison of values within the two instances. Use of *eql?* for recursive comparison is necessary to allow comparison of potentially cyclic structure. When comparing two non-eq? instances that do not share an equality procedure, `equal?` returns `#f`.

A default equality procedure to be used for all record types (including opaque types) can be specified via the parameter `default-record-equal-procedure`. The default equality procedure is used only if neither instance's type has or inherits a type-specific record equality procedure.

Similarly, when the `equal-hash` primitive hashes a record instance, it defaults to a value that is independent of the record type and contents of the instance. A program can override this behavior for instances of a record type by using `record-type-hash-procedure` to associate a hash procedure with the record-type descriptor (*rtd*) that describes the record type. The procedure `record-hash-procedure` can be used to find the hash procedure for a given record instance, following the inheritance chain. *equal-hash* passes the hash procedure two arguments: the instance plus a *hash* procedure that should be used for recursive hashing of values within the instance. Use of *hash* for recursive hashing is necessary to allow hashing of potentially cyclic structure and to make the hashing of shared structure more efficient.

A default hash procedure to be used for all record types (including opaque types) can be specified via the parameter `default-record-hash-procedure`. The default hash procedure is used only if an instance's type does not have or inherit a type-specific hash procedure.

The following example illustrates the setting of equality and hash procedures.

```
(define-record-type marble
  (nongenerative)
  (fields color quality))

(record-type-equal-procedure (record-type-descriptor marble))  ⇒  #f
(equal? (make-marble 'blue 'medium) (make-marble 'blue 'medium))  ⇒  #f
(equal? (make-marble 'blue 'medium) (make-marble 'blue 'high))  ⇒  #f

; Treat marbles as equal when they have the same color
(record-type-equal-procedure (record-type-descriptor marble)
  (lambda (m1 m2 eql?)
    (eql? (marble-color m1) (marble-color m2))))
(record-type-hash-procedure (record-type-descriptor marble)
  (lambda (m hash)
    (hash (marble-color m))))

(equal? (make-marble 'blue 'medium) (make-marble 'blue 'high))  ⇒  #t
(equal? (make-marble 'red 'high) (make-marble 'blue 'high))  ⇒  #f

(define ht (make-hashtable equal-hash equal?))
(hashtable-set! ht (make-marble 'blue 'medium) "glass")
(hashtable-ref ht (make-marble 'blue 'high) #f)  ⇒  "glass"

(define-record-type shooter
  (nongenerative)
  (parent marble)
  (fields size))

(equal? (make-marble 'blue 'medium) (make-shooter 'blue 'large 17))  ⇒  #t
(equal? (make-shooter 'blue 'large 17) (make-marble 'blue 'medium))  ⇒  #t
(hashtable-ref ht (make-shooter 'blue 'high 17) #f)  ⇒  "glass"
```

This example illustrates the application of equality and hash procedures to cyclic record structures.

```
(define-record-type node
  (nongenerative)
  (fields (mutable left) (mutable right)))

(record-type-equal-procedure (record-type-descriptor node)
  (lambda (x y e?)
    (and
      (e? (node-left x) (node-left y))
      (e? (node-right x) (node-right y)))))
(record-type-hash-procedure (record-type-descriptor node)
  (lambda (x hash)
    (+ (hash (node-left x)) (hash (node-right x)) 23)))

(define graph1
  (let ([x (make-node "a" (make-node #f "b"))])
    (node-left-set! (node-right x) x)
    x))
(define graph2
```

```
  (let ([x (make-node "a" (make-node (make-node "a" #f) "b"))])
    (node-right-set! (node-left (node-right x)) (node-right x))
    x))
(define graph3
  (let ([x (make-node "a" (make-node #f "c"))])
    (node-left-set! (node-right x) x)
    x))

(equal? graph1 graph2)  ⇒  #t
(equal? graph1 graph3)  ⇒  #f
(equal? graph2 graph3)  ⇒  #f

(define h (make-hashtable equal-hash equal?))
(hashtable-set! h graph1 #t)
(hashtable-ref h graph1 #f)  ⇒  #t
(hashtable-ref h graph2 #f)  ⇒  #t
(hashtable-ref h graph3 #f)  ⇒  #f
```

`(record-type-equal-procedure` *rtd equal-proc*`)` **procedure**
returns: unspecified
`(record-type-equal-procedure` *rtd*`)` **procedure**
returns: equality procedure associated with *rtd*, if any, otherwise `#f`
libraries: `(chezscheme)`

In the first form, *equal-proc* must be a procedure or `#f`. If *equal-proc* is a procedure, a new association between *rtd* and *equal-proc* is established, replacing any existing such association. If *equal-proc* is `#f`, any existing association between *rtd* and an equality procedure is dropped.

In the second form, `record-type-equal-procedure` returns the equality procedure associated with *rtd*, if any, otherwise `#f`.

When changing a record type's equality procedure, the record type's hash procedure, if any, should be updated if necessary to maintain the property that it produces the same hash value for any two instances the equality procedure considers equal.

`(record-equal-procedure` $record_1$ $record_2$`)` **procedure**
returns: the shared equality procedure for $record_1$ and $record_2$, if there is one, otherwise `#f`
libraries: `(chezscheme)`

`record-equal-procedure` traverses the inheritance chains for both record instances in an attempt to find the most specific type for each that is associated with an equality procedure, if any. If such type is found and is the same for both instances, the equality procedure associated with the type is returned. Otherwise, `#f` is returned.

`(record-type-hash-procedure` *rtd hash-proc*`)` **procedure**
returns: unspecified
`(record-type-hash-procedure` *rtd*`)` **procedure**
returns: hash procedure associated with *rtd*, if any, otherwise `#f`
libraries: `(chezscheme)`

In the first form, *hash-proc* must be a procedure or `#f`. If *hash-proc* is a procedure, a new association between *rtd* and *hash-proc* is established, replacing any existing such association. If *hash-proc* is `#f`, any existing association between *rtd* and a hash procedure is dropped.

In the second form, `record-type-hash-procedure` returns the hash procedure associated with *rtd*, if any, otherwise `#f`.

The procedure *hash-proc* should accept two arguments, the instance for which it should compute a hash value and a hash procedure to use to compute hash values for arbitrary fields of the instance, and it return a nonnegative exact integer. A record type's hash procedure should produce the same hash value for any two instances the record type's equality procedure considers equal.

`(record-hash-procedure` *record*`)` **procedure**
returns: the hash procedure for *record*, if there is one, otherwise `#f`
libraries: `(chezscheme)`

`record-hash-procedure` traverses the inheritance chain for the record instance in an attempt to find the most specific type that is associated with a hash procedure, if any. If such type is found, the hash procedure associated with the type is returned. Otherwise, `#f` is returned.

`default-record-equal-procedure` **thread parameter**
libraries: `(chezscheme)`

This parameter determines how two record instances are compared by `equal?` if neither has a type-specific equality procedure. When the parameter has the value `#f` (the default), `equal?` compares the instances with `eq?`, i.e., there is no attempt at determining structural equivalence. Otherwise, the parameter's value must be a procedure, and `equal?` invokes that procedure to compare the instances, passing it three arguments: the two instances and a procedure that should be used to recursively compare arbitrary values within the instances.

`default-record-hash-procedure` **thread parameter**
libraries: `(chezscheme)`

This parameter determines the hash procedure used when `equal-hash` is called on a record instance and the instance does not have a type-specific hash procedure. When the parameter has the value `#f` (the default), `equal-hash` returns a value that is independent of the record type and contents of the instance. Otherwise, the parameter's value must be a procedure, and `equal-hash` invokes the procedure to compute the instance's hash value,

passing it the record instance and a procedure to invoke to recursively compute hash values for arbitrary values contained within the record. The procedure should return a nonnegative exact integer, and the return value should be the same for any two instances the default equal procedure considers equivalent.

7.15. Legacy Record Types

In addition to the Revised[6] Report record-type creation and definition mechanisms, which are described in Chapter 9 of *The Scheme Programming Language, 4th Edition*, *Chez Scheme* continues to support pre-R6RS mechanisms for creating new data types, or *record types*, with fixed sets of named fields. Many of the procedures described in this section are available only when imported from the `(chezscheme csv7)` library.

Code intended to be portable should use the R6RS mechanism instead.

Records may be defined via the `define-record` syntactic form or via the `make-record-type` procedure. The underlying representation of records and record-type descriptors is the same for the Revised[6] Report mechanism and the alternative mechanism. Record types created by one can be used as parent record types for the other via the procedural mechanisms, though not via the syntactic mechanisms.

The syntactic (`define-record`) interface is the most commonly used interface. Each `define-record` form defines a constructor procedure for records of the new type, a type predicate that returns true only for records of the new type, an access procedure for each field, and an assignment procedure for each mutable field. For example,

```
(define-record point (x y))
```

creates a new `point` record type with two fields, `x` and `y`, and defines the following procedures:

`(make-point` *x y*`)`	constructor
`(point?` *obj*`)`	predicate
`(point-x` *p*`)`	accessor for field `x`
`(point-y` *p*`)`	accessor for field `y`
`(set-point-x!` *p obj*`)`	mutator for field `x`
`(set-point-y!` *p obj*`)`	mutator for field `y`

The names of these procedures follow a regular naming convention by default, but the programmer can override the defaults if desired. `define-record` allows the programmer to control which fields are arguments to the generated constructor procedure and which are explicitly initialized by the constructor procedure. Fields are mutable by default, but may be declared immutable. Fields can generally contain any Scheme value, but the internal representation of each field may be specified, which places implicit constraints on the type of value that may be stored there. These customization options are covered in the formal description of `define-record` later in this section.

The procedural (`make-record-type`) interface may be used to implement interpreters that must handle `define-record` forms. Each call to `make-record-type` returns a *record-type descriptor* representing the record type. Using this record-type descriptor, programs may

generate constructors, type predicates, field accessors, and field mutators dynamically. The following code demonstrates how the procedural interface might be used to create a similar `point` record type and associated definitions.

```
(define point (make-record-type "point" '(x y)))
(define make-point (record-constructor point))
(define point? (record-predicate point))
(define point-x (record-field-accessor point 'x))
(define point-y (record-field-accessor point 'y))
(define set-point-x! (record-field-mutator point 'x))
(define set-point-y! (record-field-mutator point 'y))
```

The procedural interface is more flexible than the syntactic interface, but this flexibility can lead to less readable programs and compromises the compiler's ability to generate efficient code. Programmers should use the syntactic interface whenever it suffices.

A record-type descriptor may also be extracted from an instance of a record type, whether the record type was produced by `define-record` or `make-record-type`, and the extracted descriptor may also be used to produce constructors, predicates, accessors, and mutators, with a few limitations noted in the description of `record-type-descriptor` below. This is a powerful feature that permits the coding of portable printers and object inspectors. For example, the printer employs this feature in its default record printer, and the inspector uses it to allow inspection and mutation of system- and user-defined records during debugging.

A parent record may be specified in the `define-record` syntax or as an optional argument to `make-record-type`. A new record inherits the parent record's fields, and each instance of the new record type is considered to be an instance of the parent type as well, so that accessors and mutators for the parent type may be used on instances of the new type.

Record type definitions may be classified as either generative or nongenerative. A new type results for each *generative* record definition, while only one type results for all occurrences of a given *nongenerative* record definition. This distinction is important semantically since record accessors and setters are applicable only to objects with the same type.

Syntactic (`define-record`) record definitions are *expand-time generative* by default, which means that a new record is created when the code is expanded. Expansion happens once for each form prior to compilation or interpretation, as when it is entered interactively, loaded from source, or compiled by `compile-file`. As a result, multiple evaluations of a single `define-record` form, e.g., in the body of a procedure called multiple times, always produce the same record type.

Separate `define-record` forms usually produce different types, even if the forms are textually identical. The only exception occurs when the name of a record is specified as a generated symbol, or *gensym* (page 152). Multiple copies of a record definition whose name is given by a gensym always produce the same record type; i.e., such definitions are nongenerative. Each copy of the record definition must contain the same fields and field modifiers in the same order; an exception is raised with condition-type `&assertion` when two differing record types with the same generated name are loaded into the same Scheme process.

Procedural (`make-record-type`) record definitions are *run-time generative* by default. That

is, each call to `make-record-type` usually produces a new record type. As with the syntactic interface, the only exception occurs when the name of the record is specified as a gensym, in which case the record type is fully nongenerative.

By default, a record is printed with the syntax

`#[`*type-name field* `...]`

where *field* `...` are the printed representations of the contents of the fields of the record, and *type-name* is a generated symbol, or *gensym* (page 152), that uniquely identifies the record type. For nongenerative records, *type-name* is the gensym provided by the program. Otherwise, it is a gensym whose "pretty" name (page 152) is the name given to the record by `define-record` or `make-record-type`.

The default printing of records of a given type may be overridden with `record-writer`.

The default syntax may be used as input to the reader as well, as long as the corresponding record type has already been defined in the Scheme session in which the read occurs. The parameter `record-reader` may be used to specify a different name to be recognized by the reader in place of the generated name. Specifying a different name in this manner also changes the name used when the record is printed. This reader extension is disabled in an input stream after `#!r6rs` has been seen by the reader, unless `#!chezscheme` has been seen more recently.

The mark (`#`*n*`=`) and reference (`#`*n*`#`) syntaxes may be used within the record syntax, with the result of creating shared or cyclic structure as desired. All cycles must be resolvable, however, without mutation of an immutable record field. That is, any cycle must contain at least one pointer through a mutable field, whether it is a mutable record field or a mutable field of a built-in object type such as a pair or vector.

When the parameter `print-record` is set to `#f`, records are printed using the simpler syntax

`#<record of type` *name*`>`

where *name* is the "pretty" name of the record (not the full gensym) or the reader name first assigned to the record type.

`(define-record` *name* `(`fld_1 `...) ((`fld_2 *init*`) ...) (`*opt* `...))` **syntax**
`(define-record` *name parent* `(`fld_1 `...) ((`fld_2 *init*`) ...) (`*opt* `...))` **syntax**
returns: unspecified
libraries: `(chezscheme)`

A `define-record` form is a definition and may appear anywhere and only where other definitions may appear.

`define-record` creates a new record type containing a specified set of named fields and defines a set of procedures for creating and manipulating instances of the record type.

name must be an identifier. If *name* is a generated symbol (gensym), the record definition is *nongenerative*, otherwise it is *expand-time generative*. (See the discussion of generativity earlier in this section.)

Each *fld* must be an identifier *field-name*, or it must take the form

(*class type field-name*)

where *class* and *type* are optional and *field-name* is an identifier. *class*, if present, must be the keyword `immutable` or the keyword `mutable`. If the `immutable` class specifier is present, the field is immutable; otherwise, the field is mutable. *type*, if present, specifies how the field is represented, as described below.

`ptr`	any Scheme object
`scheme-object`	same as `ptr`
`int`	a C `int`
`unsigned`	a C `unsigned int`
`short`	a C `short`
`unsigned-short`	a C `unsigned short`
`long`	a C `long`
`unsigned-long`	a C `unsigned long`
`iptr`	a signed integer the size of a `ptr`
`uptr`	an unsigned integer the size of a `ptr`
`float`	a C `float`
`double`	a C `double`
`integer-8`	an eight-bit signed integer
`unsigned-8`	an eight-bit unsigned integer
`integer-16`	a 16-bit signed integer
`unsigned-16`	a 16-bit unsigned integer
`integer-32`	a 32-bit signed integer
`unsigned-32`	a 32-bit unsigned integer
`integer-64`	a 64-bit signed integer
`unsigned-64`	a 64-bit unsigned integer
`single-float`	a 32-bit single floating point number
`double-float`	a 64-bit double floating point number

If a type is specified, the field can contain objects only of the specified type. If no type is specified, the field is of type `ptr`, meaning that it can contain any Scheme object.

The field identifiers name the fields of the record. The values of the n fields described by fld_1 ... are specified by the n arguments to the generated constructor procedure. The values of the remaining fields, fld_2 ..., are given by the corresponding expressions, *init* .... Each *init* is evaluated within the scope of the set of field names given by fld_1 ... and each field in fld_2 ... that precedes it, as if within a `let*` expression. Each of these field names is bound to the value of the corresponding field during initialization.

If *parent* is present, the record type named by *parent* is the parent of the record. The new record type inherits each of the parent record's fields, and records of the new type are considered records of the parent type. If *parent* is not present, the parent record type is a base record type with no fields.

The following procedures are defined by `define-record`:

- a constructor procedure whose name is `make-`*name*,

- a type predicate whose name is *name*`?`,
- an access procedure whose name is *name*`-`*fieldname* for each noninherited field, and
- an assignment procedure whose name is `set-`*name*`-`*fieldname*`!` for each noninherited mutable field.

If no parent record type is specified, the constructor behaves as if defined as

```
(define make-name
  (lambda (id1 ...)
    (let* ([id2 init] ...)
      body)))
```

where id_1 ... are the names of the fields defined by fld_1 ..., id_2 ... are the names of the fields defined by fld_2 ..., and *body* builds the record from the values of the identifiers id_1 ... and id_2 ....

If a parent record type is specified, the parent arguments appear first, and the parent fields are inserted into the record before the child fields.

The options *opt* ... control the selection of names of the generated constructor, predicate, accessors, and mutators.

```
(constructor id)
(predicate id)
(prefix string)
```

The option `(constructor` *id*`)` causes the generated constructor's name to be *id* rather than `make-`*name*. The option `(predicate` *id*`)` likewise causes the generated predicate's name to be *id* rather than *name*`?`. The option `(prefix` *string*`)` determines the prefix to be used in the generated accessor and mutator names in place of *name*`-`.

If no options are needed, the third subexpression, `(`*opt* `...)`, may be omitted. If no options and no fields other than those initialized by the arguments to the constructor procedure are needed, both the second and third subexpressions may be omitted. If options are specified, the second subexpression must be present, even if it contains no field specifiers.

Here is a simple example with no inheritance and no options.

```
(define-record marble (color quality))
(define x (make-marble 'blue 'medium))
(marble? x)  ⇒  #t
(pair? x)  ⇒  #f
(vector? x)  ⇒  #f
(marble-color x)  ⇒  blue
(marble-quality x)  ⇒  medium
(set-marble-quality! x 'low)
(marble-quality x)  ⇒  low
```

```
(define-record marble ((immutable color) (mutable quality))
  (((mutable shape) (if (eq? quality 'high) 'round 'unknown))))
(marble-shape (make-marble 'blue 'high))  ⇒  round
(marble-shape (make-marble 'blue 'low))  ⇒  unknown
```

```
(define x (make-marble 'blue 'high))
(set-marble-quality! x 'low)
(marble-shape x)  ⇒  round
(set-marble-shape! x 'half-round)
(marble-shape x)  ⇒  half-round
```

The following example illustrates inheritance.

```
(define-record shape (x y))
(define-record point shape ())
(define-record circle shape (radius))

(define a (make-point 7 -3))
(shape? a)  ⇒  #t
(point? a)  ⇒  #t
(circle? a)  ⇒  #f

(shape-x a)  ⇒  7
(set-shape-y! a (- (shape-y a) 1))
(shape-y a)  ⇒  -4

(define b (make-circle 7 -3 1))
(shape? b)  ⇒  #t
(point? b)  ⇒  #f
(circle? b)  ⇒  #t

(circle-radius b)  ⇒  1
(circle-radius a)  ⇒  exception: not of type circle

(define c (make-shape 0 0))
(shape? c)  ⇒  #t
(point? c)  ⇒  #f
(circle? c)  ⇒  #f
```

This example demonstrates the use of options:

```
(define-record pair (car cdr)
  ()
  ((constructor cons)
   (prefix "")))

(define x (cons 'a 'b))
(car x)  ⇒  a
(cdr x)  ⇒  b
(pair? x)  ⇒  #t

(pair? '(a b c))  ⇒  #f
x  ⇒  #[#{pair bdhavk1bwafxyss1-a} a b]
```

This example illustrates the use a specified reader name, immutable fields, and the graph mark and reference syntax.

```
(define-record triple ((immutable x1) (mutable x2) (immutable x3)))
(record-reader 'triple (type-descriptor triple))

(let ([t '#[triple #1=(1 2) (3 4) #1#]])
  (eq? (triple-x1 t) (triple-x3 t)))  ⇒  #t
(let ([x '(#1=(1 2) . #[triple #1# b c])])
  (eq? (car x) (triple-x1 (cdr x))))  ⇒  #t
```

```
(let ([t #[triple #1# (3 4) #1=(1 2)]])
  (eq? (triple-x1 t) (triple-x3 t)))  ⇒  #t
(let ([t '#1=#[triple a #1# c]])
  (eq? t (triple-x2 t)))  ⇒  #t
(let ([t '#1=(#[triple #1# b #1#])])
  (and (eq? t (triple-x1 (car t)))
       (eq? t (triple-x1 (car t)))))  ⇒  #t
```

Cycles established with the mark and reference syntax can be resolved only if a mutable record field or mutable location of some other object is involved the cycle, as in the last two examples above. An exception is raised with condition type `&lexical` if only immutable fields are involved.

```
'#1=#[triple #1# (3 4) #1#]  ⇒  exception
```

The following example demonstrates the use of nongenerative record definitions.

```
(module A (point-disp)
  (define-record #{point bdhavk1bwafxyss1-b} (x y))
  (define square (lambda (x) (* x x)))
  (define point-disp
    (lambda (p1 p2)
      (sqrt (+ (square (- (point-x p1) (point-x p2)))
               (square (- (point-y p1) (point-y p2))))))))

(module B (base-disp)
  (define-record #{point bdhavk1bwafxyss1-b} (x y))
  (import A)
  (define base-disp
    (lambda (p)
      (point-disp (make-point 0 0) p))))

(let ()
  (import B)
  (define-record #{point bdhavk1bwafxyss1-b} (x y))
  (base-disp (make-point 3 4)))  ⇒  5
```

This works even if the different program components are loaded from different source files or are compiled separately and loaded from different object files.

`predicate`	**syntax**
`prefix`	**syntax**
`constructor`	**syntax**

libraries: `(chezscheme)`

These identifiers are auxiliary keywords for `define-record`. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords. `mutable` and `immutable` are also auxiliary keywords for `define-record`, shared with the Revised[6] Report `define-record-type`.

`(type-descriptor` *name*`)` **syntax**
returns: the record-type descriptor associated with *name*
libraries: `(chezscheme)`

name must name a record type defined by `define-record` or `define-record-type`.

This form is equivalent to the Revised[6] Report `record-type-descriptor` form.

The record-type descriptor is useful for overriding the default read and write syntax using `record-reader` and `record-writer` and may also be used with the procedural interface routines described later in this section.

```
(define-record frob ())
(type-descriptor frob)  ⇒  #<record type frob>
```

`(record-reader` *name*`)` **procedure**
returns: the record-type descriptor associated with *name*
`(record-reader` *rtd*`)` **procedure**
returns: the first name associated with *rtd*
`(record-reader` *name rtd*`)` **procedure**
returns: unspecified
`(record-reader` *name* `#f)` **procedure**
returns: unspecified
`(record-reader` *rtd* `#f)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

name must be a symbol, and *rtd* must be a record-type descriptor.

With one argument, `record-reader` is used to retrieve the record type associated with a name or name associated with a record type. If no association has been created, `record-reader` returns `#f`

With arguments *name* and *rtd*, `record-reader` registers *rtd* as the record-type descriptor to be used whenever the `read` procedure encounters a record named by *name* and printed in the default record syntax.

With arguments *name* and `#f`, `record-reader` removes any association for *name* to a record-type descriptor. Similarly, with arguments *rtd* and `#f`, `record-reader` removes any association for *rtd* to a name.

```
(define-record marble (color quality))
(define m (make-marble 'blue 'perfect))
m  ⇒  #[#{marble bdhavk1bwafxyss1-c} blue perfect]

(record-reader (type-descriptor marble))  ⇒  #f
(record-reader 'marble)  ⇒  #f

(record-reader 'marble (type-descriptor marble))
(marble-color '#[marble red miserable])  ⇒  red
```

```
(record-reader (type-descriptor marble))  ⇒  marble
(record-reader 'marble)  ⇒  #<record type marble>

(record-reader (type-descriptor marble) #f)
(record-reader (type-descriptor marble))  ⇒  #f
(record-reader 'marble)  ⇒  #f

(record-reader 'marble (type-descriptor marble))
(record-reader 'marble #f)
(record-reader (type-descriptor marble))  ⇒  #f
(record-reader 'marble)  ⇒  #f
```

The introduction of a record reader also changes the default printing of records. The printer always chooses the reader name first assigned to the record, if any, in place of the unique record name, as this continuation of the example above demonstrates.

```
(record-reader 'marble (type-descriptor marble))
(make-marble 'pink 'splendid)  ⇒  #[marble pink splendid]
```

`(record-writer` *rtd*`)` **procedure**
returns: the record writer associated with *rtd*
`(record-writer` *rtd procedure*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

rtd must be a record-type descriptor, and *procedure* should accept three arguments, as described below.

When passed only one argument, `record-writer` returns the record writer associated with *rtd*, which is initially the default record writer for all records. The default print method prints all records in a uniform syntax that includes the generated name for the record and the values of each of the fields, as described in the introduction to this section.

When passed two arguments, `record-writer` establishes a new association between *rtd* and *procedure* so that *procedure* will be used by the printer in place of the default printer for records of the given type. The printer passes *procedure* three arguments: the record *r*, a port *p*, and a procedure *wr* that should be used to write out the values of arbitrary Scheme objects that the print method chooses to include in the printed representation of the record, e.g., values of the record's fields.

```
(define-record marble (color quality))
(define m (make-marble 'blue 'medium))

m  ⇒  #[#{marble bdhavk1bwafxyss1-d} blue medium]

(record-writer (type-descriptor marble)
  (lambda (r p wr)
    (display "#<" p)
    (wr (marble-quality r) p)
    (display " quality " p)
```

```
    (wr (marble-color r) p)
    (display " marble>" p)))

m  ⇒  #<medium quality blue marble>
```

The record writer is used only when `print-record` is true (the default). When the parameter `print-record` is set to `#f`, records are printed using a compressed syntax that identifies only the type of record.

```
(parameterize ([print-record #f])
  (format "~s" m))  ⇒  "#<record of type marble>"
```

A print method may be called more than once during the printing of a single record to support cycle detection and graph printing (see `print-graph`), so print methods that perform side effects other than printing to the given port are discouraged. Whenever a print method is called more than once during the printing of a single record, in all but one call, a generic "bit sink" port is used to suppress output automatically so that only one copy of the object appears on the actual port. In order to avoid confusing the cycle detection and graph printing algorithms, a print method should always produce the same printed representation for each object. Furthermore, a print method should normally use the supplied procedure *wr* to print subobjects, though atomic values, such as strings or numbers, may be printed by direct calls to `display` or `write` or by other means.

```
(let ()
  (define-record ref () ((contents 'nothing)))
  (record-writer (type-descriptor ref)
    (lambda (r p wr)
      (display "<" p)
      (wr (ref-contents r) p)
      (display ">" p)))
  (let ([ref-lexive (make-ref)])
    (set-ref-contents! ref-lexive ref-lexive)
    ref-lexive))  ⇒  #0=<#0#>
```

Print methods need not be concerned with handling nonfalse values of the parameters `print-level`. The printer handles `print-level` automatically even when user-defined print procedures are used. Since records typically contain a small, fixed number of fields, it is usually possible to ignore nonfalse values of `print-length` as well.

```
(print-level 3)
(let ()
  (define-record ref () ((contents 'nothing)))
  (record-writer (type-descriptor ref)
    (lambda (r p wr)
      (display "<" p)
      (wr (ref-contents r) p)
      (display ">" p)))
  (let ([ref-lexive (make-ref)])
    (set-ref-contents! ref-lexive ref-lexive)
    ref-lexive))  ⇒  <<<<#[...]>>>>
```

print-record **thread parameter**
libraries: (chezscheme)

This parameter controls the printing of records. If set to true (the default) the record writer associated with a record type is used to print records of that type. If set to false, all records are printed with the syntax `#<record of type` *name*`>`, where *name* is the name of the record type as returned by `record-type-name`.

(`make-record-type` *type-name fields*) **procedure**
(`make-record-type` *parent-rtd type-name fields*) **procedure**
returns: a record-type descriptor for a new record type
libraries: (chezscheme)

`make-record-type` creates a new data type and returns a record-type descriptor, a value representing the new data type. The new type is disjoint from all others.

If present, *parent-rtd* must be a record-type descriptor.

type-name must be a string or gensym. If *type-name* is a string, a new record type is generated. If *type-name* is a gensym, a new record type is generated only if one with the same gensym has not already been defined. If one has already been defined, the parent and fields must be identical to those of the existing record type, and the existing record type is used. If the parent and fields are not identical, an exception is raised with condition-type `&assertion`.

fields must be a list of field descriptors, each of which describes one field of instances of the new record type. A field descriptor is either a symbol or a list in the following form:

(*class type field-name*)

where *class* and *type* are optional. *field-name* must be a symbol. *class*, if present, must be the symbol `immutable` or the symbol `mutable`. If the `immutable` class-specifier is present, the field is immutable; otherwise, the field is mutable. *type*, if present, specifies how the field is represented. The types are the same as those given in the description of `define-record` on page 173.

If a type is specified, the field can contain objects only of the specified type. If no type is specified, the field is of type `ptr`, meaning that it can contain any Scheme object.

The behavior of a program that modifies the string *type-name* or the list *fields* or any of its sublists is unspecified.

The record-type descriptor may be passed as an argument to any of the Revised[6] Report procedures

- `record-constructor`,
- `record-predicate`,
- `record-accessor`, and
- `record-mutator`,

or to the *Chez Scheme* variants

- `record-constructor`,
- `record-field-accessor`, and
- `record-field-mutator`

to obtain procedures for creating and manipulating records of the new type.

```
(define marble
  (make-record-type "marble"
    '(color quality)
    (lambda (r p wr)
      (display "#<" p)
      (wr (marble-quality r) p)
      (display " quality " p)
      (wr (marble-color r) p)
      (display " marble>" p))))
(define make-marble
  (record-constructor marble))
(define marble?
  (record-predicate marble))
(define marble-color
  (record-field-accessor marble 'color))
(define marble-quality
  (record-field-accessor marble 'quality))
(define set-marble-quality!
  (record-field-mutator marble 'quality))
(define x (make-marble 'blue 'high))
(marble? x)  ⇒  #t
(marble-quality x)  ⇒  high
(set-marble-quality! x 'low)
(marble-quality x)  ⇒  low
x  ⇒  #<low quality blue marble>
```

The order in which the fields appear in *fields* is important. While field names are generally distinct, it is permissible for one field name to be the same as another in the list of fields or the same as an inherited name. In this case, field ordinals must be used to select fields in calls to `record-field-accessor` and `record-field-mutator`. Ordinals range from zero through one less than the number of fields. Parent fields come first, if any, followed by the fields in *fields*, in the order given.

```
(define r1 (make-record-type "r1" '(t t)))
(define r2 (make-record-type r1 "r2" '(t)))
(define r3 (make-record-type r2 "r3" '(t t t)))

(define x ((record-constructor r3) 'a 'b 'c 'd 'e 'f))
((record-field-accessor r3 0) x)  ⇒  a
((record-field-accessor r3 2) x)  ⇒  c
((record-field-accessor r3 4) x)  ⇒  e
((record-field-accessor r3 't) x)  ⇒  unspecified
```

`(record-constructor` *rcd*`)` **procedure**
`(record-constructor` *rtd*`)` **procedure**
returns: a constructor for records of the type represented by *rtd*
libraries: `(chezscheme)`

Like the Revised[6] Report version of this procedure, this procedure may be passed a record-constructor descriptor, *rcd*, which determines the behavior of the constructor. It may also be passed a record-type descriptor, *rtd*, in which case the constructor accepts as many arguments as there are fields in the record; these arguments are the initial values of the fields in the order given when the record-type descriptor was created.

`(record-field-accessor` *rtd field-id*`)` **procedure**
returns: an accessor for the identified field
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor, *field-id* must be a symbol or field ordinal, i.e., a nonnegative exact integer less than the number of fields of the given record type. The specified field must be accessible.

The generated accessor expects one argument, which must be a record of the type represented by *rtd*. It returns the contents of the specified field of the record.

`(record-field-accessible?` *rtd field-id*`)` **procedure**
returns: `#t` if the specified field is accessible, otherwise `#f`
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor, *field-id* must be a symbol or field ordinal, i.e., a nonnegative exact integer less than the number of fields of the given record type.

The compiler is free to eliminate a record field if it can prove that the field is not accessed. In making this determination, the compiler is free to ignore the possibility that an accessor might be created from a record-type descriptor obtained by calling `record-type-descriptor` on an instance of the record type.

`(record-field-mutator` *rtd field-id*`)` **procedure**
returns: a mutator for the identified field
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor, *field-id* must be a symbol or field ordinal, i.e., a nonnegative exact integer less than the number of fields of the given record type. The specified field must be mutable.

The mutator expects two arguments, *r* and *obj*. *r* must be a record of the type represented by *rtd*. *obj* must be a value that is compatible with the type declared for the specified field when the record-type descriptor was created. *obj* is stored in the specified field of the record.

`(record-field-mutable?` *rtd field-id*`)` **procedure**
returns: `#t` if the specified field is mutable, otherwise `#f`
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor, *field-id* must be a symbol or field ordinal, i.e., a nonnegative exact integer less than the number of fields of the given record type.

Any field declared immutable is immutable. In addition, the compiler is free to treat a field as immutable if it can prove that the field is never assigned. In making this determination, the compiler is free to ignore the possibility that a mutator might be created from a record-type descriptor obtained by calling `record-type-descriptor` on an instance of the record type.

`(record-type-name` *rtd*`)` **procedure**
returns: the name of the record-type represented by *rtd*
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor.

The name is a always a string. If a gensym is provided as the record-type name in a `define-record` form or `make-record-type` call, the result is the "pretty" name of the gensym (see 7.9).

```
(record-type-name (make-record-type "empty" '()))  ⇒  "empty"

(define-record #{point bdhavk1bwafxyss1-b} (x y))
(define p (type-descriptor #{point bdhavk1bwafxyss1-b}))
(record-type-name p)  ⇒  "point"
```

`(record-type-symbol` *rtd*`)` **procedure**
returns: the generated symbol associated with *rtd*
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor.

```
(define e (make-record-type "empty" '()))
(record-type-symbol e)  ⇒  #{empty bdhavk1bwafxyss1-e}

(define-record #{point bdhavk1bwafxyss1-b} (x y))
(define p (type-descriptor #{point bdhavk1bwafxyss1-b}))
(record-type-symbol p)  ⇒  #{point bdhavk1bwafxyss1-b}
```

`(record-type-field-names` *rtd*`)` **procedure**
returns: a list of field names of the type represented by *rtd*
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor. The field names are symbols.

```
(define-record triple ((immutable x1) (mutable x2) (immutable x3)))
(record-type-field-names (type-descriptor triple))  ⇒  (x1 x2 x3)
```

`(record-type-field-decls` *rtd*`)` **procedure**
returns: a list of field declarations of the type represented by *rtd*
libraries: `(chezscheme csv7)`

rtd must be a record-type descriptor. Each field declaration has the following form:

(*class type field-name*)

where *class*, *type*, and *field-name* are as described under `make-record-type`.

```
(define-record shape (x y))
(define-record circle shape (radius))

(record-type-field-decls
  (type-descriptor circle))  ⇒  ((mutable ptr x)
                                 (mutable ptr y)
                                 (mutable ptr radius))
```

`(record?` *obj*`)` **procedure**
returns: `#t` if *obj* is a record, otherwise `#f`
`(record?` *obj rtd*`)` **procedure**
returns: `#t` if *obj* is a record of the given type, otherwise `#f`
libraries: `(chezscheme)`

If present, *rtd* must be a record-type descriptor.

A record is "of the given type" if it is an instance of the record type or one of its ancestors. The predicate generated by `record-predicate` for a record-type descriptor *rtd* is equivalent to the following.

```
(lambda (x) (record? x rtd))
```

`(record-type-descriptor` *rec*`)` **procedure**
returns: the record-type descriptor of *rec*
libraries: `(chezscheme csv7)`

rec must be a record. This procedure is intended for use in the definition of portable printers and debuggers. For records created with `make-record-type`, it may not be the same as the descriptor returned by `make-record-type`. See the comments about field accessibility and mutability under `record-field-accessible?` and `record-field-mutable?` above.

This procedure is equivalent to the Revised[6] Report `record-rtd` procedure.

```
(define rtd (make-record-type "frob" '(blit blat)))
rtd  ⇒  #<record type frob>
(define x ((record-constructor rtd) 1 2))
(record-type-descriptor x)  ⇒  #<record type frob>
(eq? (record-type-descriptor x) rtd)  ⇒  unspecified
```

7.16. Procedures

(procedure-arity-mask *proc*) **procedure**

returns: an exact integer bitmask identifying the accepted argument counts of *proc*

libraries: (chezscheme)

The bitmask is represented as two's complement number with the bit at each index n set if and only if *proc* accepts n arguments.

The two's complement encoding implies that if *proc* accepts n or more arguments, the encoding is a negative number, since all the bits from n and up are set. For example, if *proc* accepts any number of arguments, the two's complement encoding of all bits set is -1.

```
(procedure-arity-mask (lambda () 'none))  ⇒  1
(procedure-arity-mask car)  ⇒  2
(procedure-arity-mask (case-lambda [() 'none] [(x) x]))  ⇒  3
(procedure-arity-mask (lambda x x))  ⇒  -1
(procedure-arity-mask (case-lambda [() 'none] [(x y . z) x]))  ⇒  -3
(procedure-arity-mask (case-lambda))  ⇒  0
(logbit? 1 (procedure-arity-mask pair?))  ⇒  #t
(logbit? 2 (procedure-arity-mask pair?))  ⇒  #f
(logbit? 2 (procedure-arity-mask cons))  ⇒  #t
```

8. Numeric Operations

This chapter describes *Chez Scheme* extensions to the standard set of operations on numbers. See Chapter 6 of *The Scheme Programming Language, 4th Edition* or the Revised[6] Report on Scheme for a description of standard operations on numbers.

Chez Scheme supports the full set of Scheme numeric datatypes, including exact and inexact integer, rational, real, and complex numbers. A variety of representations are used to support these datatypes:

Fixnums represent exact integers in the fixnum range (see `most-negative-fixnum` and `most-positive-fixnum`). The length of a string, vector, or fxvector is constrained to be a fixnum.

Bignums represent arbitrary-precision exact integers outside of the fixnum range.

Ratnums represent arbitrary-precision exact rational numbers. Each ratnum contains an exact integer (fixnum or bignum) numerator and an exact integer denominator. Ratios are always reduced to lowest terms and never have a denominator of one or a numerator of zero.

Flonums represent inexact real numbers. Flonums are IEEE 64-bit floating-point numbers. (Since flonums cannot represent irrational numbers, all inexact real numbers are actually rational, although they may approximate irrational quantities.)

Exact complexnums represent exact complex numbers. Each exact complexnum contains an exact rational (fixnum, bignum, or ratnum) real part and an exact rational imaginary part.

Inexact complexnums represent inexact complex numbers. Each inexact complexnum contains a flonum real part and a flonum imaginary part.

Most numbers can be represented in only one way; however, real numbers are sometimes represented as inexact complex numbers with imaginary component equal to zero.

Chez Scheme extends the syntax of numbers with arbitrary radixes from two through 36, nondecimal floating-point and scientific notation, and printed representations for IEEE infinities and NANs. (NAN stands for "not-a-number.")

Arbitrary radixes are specified with the prefix `#`*n*`r`, where *n* ranges from 2 through 36. Digits beyond 9 are specified with the letters (in either upper or lower case) `a` through `z`. For example, `#2r101` is 5_{10}, and `#36rZ` is 35_{10}.

For higher radixes, an ambiguity arises between the interpretation of certain letters, e.g., `e`, as digits or exponent specifiers; in such cases, the letter is assumed to be a digit. For

example, the `e` in `#x3.2e5` is interpreted as a digit, not as an exponent marker, whereas in `3.2e5` it is treated as an exponent marker.

IEEE infinities are printed as `+inf.0` and `-inf.0`, while IEEE NANs are printed as `+nan.0` or `-nan.0`. (+nan.0 is used on output for all NANs.)

```
(/ 1.0 0.0)  ⇒  +inf.0
(/ 1.0 -0.0)  ⇒  -inf.0
(/ 0.0 0.0)  ⇒  +nan.0
(/ +inf.0 -inf.0)  ⇒  +nan.0
```

The first section of this chapter describes type-specific numeric type predicates. Sections 8.2 through 8.4 describe fast, type-specific numeric operations on fixnums, flonums, and inexact complex numbers (flonums and/or inexact complexnums). The fixnum-specific versions should be used only when the programmer is certain that the operands and results (where appropriate) will be fixnums, i.e., integers in the range `(most-negative-fixnum)` to `(most-positive-fixnum)`, inclusive. The flonum-specific versions should be used only when the inputs and outputs (where appropriate) are certain to be flonums. The mixed flonum/complexnum versions should be used only when the inputs are certain to be either flonums or inexact complexnums. Section 8.5 describes operations, both arbitrary precision and fixnum-specific, that allow exact integers to be treated as sets or sequences of bits. Random number generation is covered Section 8.6, and miscellaneous numeric operations are covered in the Section 8.7.

8.1. Numeric Type Predicates

The Revised[6] Report distinguishes two types of special numeric objects: fixnums and flonums. *Chez Scheme* additionally distinguishes *bignums* (exact integers outside of the fixnum range) and *ratnums* (ratios of exact integers). It also provides a predicate for recognizing *cflonums*, which are flonums or inexact complex numbers.

`(bignum?` *obj*`)` **procedure**
returns: `#t` if *obj* is a bignum, otherwise `#f`
libraries: `(chezscheme)`

```
(bignum? 0)  ⇒  #f
(bignum? (most-positive-fixnum))  ⇒  #f
(bignum? (most-negative-fixnum))  ⇒  #f
(bignum? (* (most-positive-fixnum) 2))  ⇒  #t
(bignum? 3/4)  ⇒  #f
(bignum? 'a)  ⇒  #f
```

(ratnum? *obj*) **procedure**
returns: `#t` if *obj* is a ratnum, otherwise `#f`
libraries: `(chezscheme)`

```
(ratnum? 0)  ⇒  #f
(ratnum? (* (most-positive-fixnum) 2))  ⇒  #f
(ratnum? 3/4)  ⇒  #t
(ratnum? -10/2)  ⇒  #f
(ratnum? -11/2)  ⇒  #t
(ratnum? 'a)  ⇒  #f
```

(cflonum? *obj*) **procedure**
returns: `#t` if *obj* is an inexact complexnum or flonum, otherwise `#f`
libraries: `(chezscheme)`

```
(cflonum? 0)  ⇒  #f
(cflonum? 0.0)  ⇒  #t
(cflonum? 3+4i)  ⇒  #f
(cflonum? 3.0+4i)  ⇒  #t
(cflonum? +i)  ⇒  #f
(cflonum? +1.0i)  ⇒  #t
```

8.2. Fixnum Operations

Fixnum-specific procedures normally check their inputs and outputs (where appropriate), but at optimization level 3 the compiler generates, in most cases, code that does not perform these checks.

(most-positive-fixnum) **procedure**
returns: the most positive fixnum supported by the system
(most-negative-fixnum) **procedure**
returns: the most negative fixnum supported by the system
libraries: `(chezscheme)`

These procedures are identical to the Revised[6] Report `greatest-fixnum` and `least-fixnum` procedures.

(fx= *fixnum*$_1$ *fixnum*$_2$ ...) **procedure**
(fx< *fixnum*$_1$ *fixnum*$_2$ ...) **procedure**
(fx> *fixnum*$_1$ *fixnum*$_2$ ...) **procedure**
(fx<= *fixnum*$_1$ *fixnum*$_2$ ...) **procedure**
(fx>= *fixnum*$_1$ *fixnum*$_2$ ...) **procedure**
returns: `#t` if the relation holds, `#f` otherwise
libraries: `(chezscheme)`

The predicate `fx=` returns `#t` if its arguments are equal. The predicate `fx<` returns `#t` if its

arguments are monotonically increasing, i.e., each argument is greater than the preceding ones, while `fx>` returns `#t` if its arguments are monotonically decreasing. The predicate `fx<=` returns `#t` if its arguments are monotonically nondecreasing, i.e., each argument is not less than the preceding ones, while `fx>=` returns `#t` if its arguments are monotonically nonincreasing. When passed only one argument, each of these predicates returns `#t`.

These procedures are similar to the Revised[6] Report procedures `fx=?`, `fx<?`, `fx>?`, `fx<=?`, and `fx>=?` except that the Revised[6] Report procedures require two or more arguments, and their names have the "`?`" suffix.

```
(fx= 0)  ⇒  #t
(fx= 0 0)  ⇒  #t
(fx< (most-negative-fixnum) 0 (most-positive-fixnum))  ⇒  #t
(let ([x 3]) (fx<= 0 x 9))  ⇒  #t
(fx<= 0 3 3)  ⇒  #t
(fx>= 0 0 (most-negative-fixnum))  ⇒  #t
```

`(fxnonpositive?` *fixnum*`)` **procedure**
returns: `#t` if *fixnum* is not greater than zero, `#f` otherwise
`(fxnonnegative?` *fixnum*`)` **procedure**
returns: `#t` if *fixnum* is not less than zero, `#f` otherwise
libraries: `(chezscheme)`

`fxnonpositive?` is equivalent to `(lambda (x) (fx<= x 0))`, and `fxnonnegative?` is equivalent to `(lambda (x) (fx>= x 0))`.

```
(fxnonpositive? 128)  ⇒  #f
(fxnonpositive? 0)  ⇒  #t
(fxnonpositive? -1)  ⇒  #t

(fxnonnegative? -65)  ⇒  #f
(fxnonnegative? 0)  ⇒  #t
(fxnonnegative? 1)  ⇒  #t
```

`(fx+` *fixnum* `...)` **procedure**
returns: the sum of the arguments *fixnum* `...`
libraries: `(chezscheme)`

When called with no arguments, `fx+` returns `0`.

```
(fx+)  ⇒  0
(fx+ 1 2)  ⇒  3
(fx+ 3 4 5)  ⇒  12
(apply fx+ '(1 2 3 4 5))  ⇒  15
```

`(fx-` *fixnum*$_1$ *fixnum*$_2$ `...)` **procedure**
returns: a fixnum
libraries: `(chezscheme)`

When called with one argument, `fx-` returns the negative of *fixnum*$_1$. Thus, `(fx-` *fixnum*$_1$`)` is an idiom for `(fx- 0` *fixnum*$_1$`)`.

When called with two or more arguments, `fx-` returns the result of subtracting the sum of the numbers *fixnum*$_2$ `...` from *fixnum*$_1$.

```
(fx- 3)  ⇒  -3
(fx- 4 3)  ⇒  1
(fx- 4 3 2 1)  ⇒  -2
```

`(fx*` *fixnum* `...)` **procedure**
returns: the product of the arguments *fixnum* `...`
libraries: `(chezscheme)`

When called with no arguments, `fx*` returns `1`.

```
(fx*)  ⇒  1
(fx* 1 2)  ⇒  2
(fx* 3 -4 5)  ⇒  -60
(apply fx* '(1 -2 3 -4 5))  ⇒  120
```

`(fx/` *fixnum*$_1$ *fixnum*$_2$ `...)` **procedure**
returns: see explanation
libraries: `(chezscheme)`

When called with one argument, `fx/` returns the reciprocal of *fixnum*$_1$. That is, `(fx/` *fixnum*$_1$`)` is an idiom for `(fx/ 1` *fixnum*$_1$`)`.

When called with two or more arguments, `fx/` returns the result of dividing *fixnum*$_1$ by the product of the remaining arguments *fixnum*$_2$ `...`.

```
(fx/ 1)  ⇒  1
(fx/ -17)  ⇒  0
(fx/ 8 -2)  ⇒  -4
(fx/ -9 2)  ⇒  -4
(fx/ 60 5 3 2)  ⇒  2
```

`(fx1+` *fixnum*`)` **procedure**
`(fx1-` *fixnum*`)` **procedure**
returns: *fixnum* plus 1 or *fixnum* minus 1
libraries: `(chezscheme)`

```
(define fxplus
  (lambda (x y)
    (if (fxzero? x)
        y
        (fxplus (fx1- x) (fx1+ y)))))

(fxplus 7 8)  ⇒  15
```

`fx1+` and `fx1-` can be defined as follows:

```
(define fx1+ (lambda (x) (fx+ x 1)))
(define fx1- (lambda (x) (fx- x 1)))
```

(`fxquotient` *fixnum*$_1$ *fixnum*$_2$ ...) **procedure**
returns: see explanation
libraries: (`chezscheme`)

`fxquotient` is identical to `fx/`. See the description of `fx/` above.

(`fxremainder` *fixnum*$_1$ *fixnum*$_2$) **procedure**
returns: the fixnum remainder of *fixnum*$_1$ divided by *fixnum*$_2$
libraries: (`chezscheme`)

The result of `fxremainder` has the same sign as *fixnum*$_1$.

```
(fxremainder 16 4)  ⇒  0
(fxremainder 5 2)  ⇒  1
(fxremainder -45 7)  ⇒  -3
(fxremainder 10 -3)  ⇒  1
(fxremainder -17 -9)  ⇒  -8
```

(`fxmodulo` *fixnum*$_1$ *fixnum*$_2$) **procedure**
returns: the fixnum modulus of *fixnum*$_1$ and *fixnum*$_2$
libraries: (`chezscheme`)

The result of `fxmodulo` has the same sign as *fixnum*$_2$.

```
(fxmodulo 16 4)  ⇒  0
(fxmodulo 5 2)  ⇒  1
(fxmodulo -45 7)  ⇒  4
(fxmodulo 10 -3)  ⇒  -2
(fxmodulo -17 -9)  ⇒  -8
```

(`fxabs` *fixnum*) **procedure**
returns: the absolute value of *fixnum*
libraries: (`chezscheme`)

```
(fxabs 1)  ⇒  1
```

```
(fxabs -1)  ⇒  1
(fxabs 0)  ⇒  0
```

8.3. Flonum Operations

Inexact real numbers are normally represented by *flonums*. A flonum is a single 64-bit double-precision floating point number. This section describes operations on flonums, most of which accept flonum arguments and return flonum values. In most cases, the operations are inline-coded or coded as machine language subroutines at optimize-level 3 with no argument type checking; full type checking is performed at lower optimize levels. Flonum-specific procedure names begin with the prefix "`fl`" to set them apart from their generic counterparts.

Inexact real numbers may also be represented by inexact complexnums with imaginary parts equal to zero, which cannot be used as input to the flonum-specific operators. Such numbers are produced, however, only from operations involving complex numbers with nonzero imaginary parts, by explicit calls to `fl-make-rectangular`, `make-rectangular`, or `make-polar`, or by numeric input in either polar or rectangular format.

`(flonum->fixnum` *flonum*`)` **procedure**
returns: the fixnum representation of *flonum*, truncated
libraries: `(chezscheme)`

The truncated value of *flonum* must fall within the fixnum range. `flonum->fixnum` is a restricted version of `exact`, which converts any numeric representation to its exact equivalent.

```
(flonum->fixnum 0.0)  ⇒  0
(flonum->fixnum 3.9)  ⇒  3
(flonum->fixnum -2.2)  ⇒  -2
```

`(fl=` $flonum_1$ $flonum_2$ `...)` **procedure**
`(fl<` $flonum_1$ $flonum_2$ `...)` **procedure**
`(fl>` $flonum_1$ $flonum_2$ `...)` **procedure**
`(fl<=` $flonum_1$ $flonum_2$ `...)` **procedure**
`(fl>=` $flonum_1$ $flonum_2$ `...)` **procedure**
returns: `#t` if the relation holds, `#f` otherwise
libraries: `(chezscheme)`

The predicate `fl=` returns `#t` if its arguments are equal. The predicate `fl<` returns `#t` if its arguments are monotonically increasing, i.e., each argument is greater than the preceding ones, while `fl>` returns `#t` if its arguments are monotonically decreasing. The predicate `fl<=` returns `#t` if its arguments are monotonically nondecreasing, i.e., each argument is not less than the preceding ones, while `fl>=` returns `#t` if its arguments are monotonically nonincreasing. When passed only one argument, each of these predicates returns `#t`.

IEEE NANs are not comparable, i.e., comparisons involving NANs always return `#f`.

These procedures are similar to the Revised[6] Report procedures `fl=?`, `fl<?`, `fl>?`, `fl<=?`, and `fl>=?` except that the Revised[6] Report procedures require two or more arguments, and their names have the "`?`" suffix.

```
(fl= 0.0)  ⇒  #t
(fl= 0.0 0.0)  ⇒  #t
(fl< -1.0 0.0 1.0)  ⇒  #t
(fl> -1.0 0.0 1.0)  ⇒  #f
(fl<= 0.0 3.0 3.0)  ⇒  #t
(fl>= 4.0 3.0 3.0)  ⇒  #t
(fl< 7.0 +inf.0)  ⇒  #t
(fl= +nan.0 0.0)  ⇒  #f
(fl= +nan.0 +nan.0)  ⇒  #f
(fl< +nan.0 +nan.0)  ⇒  #f
(fl> +nan.0 +nan.0)  ⇒  #f
```

`(flnonpositive?` *fl*`)` **procedure**
returns: `#t` if *fl* is not greater than zero, `#f` otherwise
`(flnonnegative?` *fl*`)` **procedure**
returns: `#t` if *fl* is not less than zero, `#f` otherwise
libraries: `(chezscheme)`

`flnonpositive?` is equivalent to `(lambda (x) (fl<= x 0.0))`, and `flnonnegative?` is equivalent to `(lambda (x) (fl>= x 0.0))`.

Even if the flonum representation distinguishes -0.0 from +0.0, both are considered nonpositive and nonnegative.

```
(flnonpositive? 128.0)  ⇒  #f
(flnonpositive? 0.0)  ⇒  #t
(flnonpositive? -0.0)  ⇒  #t
(flnonpositive? -1.0)  ⇒  #t

(flnonnegative? -65.0)  ⇒  #f
(flnonnegative? 0.0)  ⇒  #t
(flnonnegative? -0.0)  ⇒  #t
(flnonnegative? 1.0)  ⇒  #t

(flnonnegative? +nan.0)  ⇒  #f
(flnonpositive? +nan.0)  ⇒  #f

(flnonnegative? +inf.0)  ⇒  #t
(flnonnegative? -inf.0)  ⇒  #f
```

`(decode-float` *x*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

x must be a flonum. `decode-float` returns a vector with three integer elements, *m*, *e*, and *s*, such that $x = sm2^e$. It is useful primarily in the printing of floating-point numbers.

```
(decode-float 1.0)  ⇒  #(4503599627370496 -52 1)
(decode-float -1.0)  ⇒  #(4503599627370496 -52 -1)
```

```
(define slow-identity
  (lambda (x)
    (inexact
      (let ([v (decode-float x)])
        (let ([m (vector-ref v 0)]
              [e (vector-ref v 1)]
              [s (vector-ref v 2)])
          (* s m (expt 2 e)))))))
```

```
(slow-identity 1.0)  ⇒  1.0
(slow-identity -1e20)  ⇒  -1e20
```

`(fllp` *flonum*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

`fllp` returns the 12-bit integer consisting of the exponent plus highest order represented bit of a flonum (ieee 64-bit floating-point number). It can be used to compute a fast approximation of the logarithm of the number.

```
(fllp 0.0)  ⇒  0
(fllp 1.0)  ⇒  2046
(fllp -1.0)  ⇒  2046

(fllp 1.5)  ⇒  2047

(fllp +inf.0)  ⇒  4094
(fllp -inf.0)  ⇒  4094

(fllp #b1.0e-1111111111)  ⇒  1
(fllp #b1.0e-10000000000)  ⇒  0
```

8.4. Inexact Complex Operations

The procedures described in this section provide mechanisms for creating and operating on inexact complex numbers. Inexact complex numbers with nonzero imaginary parts are represented as *inexact complexnums*. An inexact complexnum contains two 64-bit double-precision floating point numbers. Inexact complex numbers with imaginary parts equal to zero (in other words, inexact real numbers) may be represented as either inexact

complexnums or flonums. The operations described in this section accept any mix of inexact complexnum and flonum arguments (collectively, "cflonums").

In most cases, the operations are performed with minimal type checking at optimize-level 3; full type checking is performed at lower optimize levels. Inexact complex procedure names begin with the prefix "`cfl`" to set them apart from their generic counterparts.

`(fl-make-rectangular` *flonum*$_1$ *flonum*$_2$`)` **procedure**
returns: an inexact complexnum
libraries: `(chezscheme)`

The inexact complexnum produced by fl-make-rectangular has real part equal to *flonum*$_1$ and imaginary part equal to *flonum*$_2$.

```
(fl-make-rectangular 2.0 -3.0)  ⇒  2.0-3.0i
(fl-make-rectangular 2.0 0.0)  ⇒  2.0+0.0i
(fl-make-rectangular 2.0 -0.0)  ⇒  2.0-0.0i
```

`(cfl-real-part` *cflonum*`)` **procedure**
returns: the real part of *cflonum*
`(cfl-imag-part` *cflonum*`)` **procedure**
returns: the imaginary part of *cflonum*
libraries: `(chezscheme)`

```
(cfl-real-part 2.0-3.0i)  ⇒  2.0
(cfl-imag-part 2.0-3.0i)  ⇒  -3.0
(cfl-imag-part 2.0-0.0i)  ⇒  -0.0
(cfl-imag-part 2.0-inf.0i)  ⇒  -inf.0
```

`(cfl=` *cflonum* ...`)` **procedure**
returns: `#t` if its arguments are equal, `#f` otherwise
libraries: `(chezscheme)`

```
(cfl= 7.0+0.0i 7.0)  ⇒  #t
(cfl= 1.0+2.0i 1.0+2.0i)  ⇒  #t
(cfl= 1.0+2.0i 1.0-2.0i)  ⇒  #f
```

`(cfl+` *cflonum* ...`)` **procedure**
`(cfl*` *cflonum* ...`)` **procedure**
`(cfl-` *cflonum*$_1$ *cflonum*$_2$ ...`)` **procedure**
`(cfl/` *cflonum*$_1$ *cflonum*$_2$ ...`)` **procedure**
returns: a cflonum
libraries: `(chezscheme)`

These procedures compute the sum, difference, product, or quotient of inexact complex quantities, whether these quantities are represented by flonums or inexact complexnums.

For example, if `cfl+` receives two flonum arguments a and b, it returns the sum $a + b$; in this case, it behaves the same as `fl+`. With two inexact complexnum arguments $a + bi$ and $c + di$, it returns the sum $(a + c) + (b + d)i$. If one argument is a flonum a and the other an inexact complexnum $c + di$, `cfl+` returns $(a + c) + di$.

When passed zero arguments, `cfl+` returns 0.0 and `cfl*` returns 1.0. When passed one argument, `cfl-` returns the additive inverse of the argument, and `cfl/` returns the multiplicative inverse of the argument. When passed three or more arguments, `cfl-` returns the difference between its first and the sum of its remaining arguments, and `cfl/` returns the quotient of its first and the product of its remaining arguments.

```
(cfl+)  ⇒  0.0
(cfl*)  ⇒  1.0
(cfl- 5.0+1.0i)  ⇒  -5.0-1.0i
(cfl/ 2.0+2.0i)  ⇒  0.25-0.25i

(cfl+ 1.0+2.2i -3.7+5.3i)  ⇒  -2.7+7.5i
(cfl+ 1.0 -5.3)  ⇒  -4.3
(cfl+ 1.0 2.0 -5.3i)  ⇒  3.0-5.3i
(cfl- 1.0+2.5i -3.7)  ⇒  4.7+2.5i
(cfl* 1.0+2.0i 3.0+4.0i)  ⇒  -5.0+10.0i
(cfl/ -5.0+10.0i 1.0+2.0i 2.0)  ⇒  1.5+2.0i
```

`(cfl-conjugate` *cflonum*`)` **procedure**

returns: complex conjugate of *cflonum*
libraries: `(chezscheme)`

The procedure `cfl-conjugate`, when passed an inexact complex argument $a + bi$, returns its complex conjugate $a + (-b)i$.

See also `conjugate`, which is a generic version of this operator that returns the complex conjugate of any valid representation for a complex number.

```
(cfl-conjugate 3.0)  ⇒  3.0
(cfl-conjugate 3.0+4.0i)  ⇒  3.0-4.0i
(cfl-conjugate 1e-20-2e-30i)  ⇒  1e-20+2e-30i
```

`(cfl-magnitude-squared` *cflonum*`)` **procedure**

returns: magnitude of *cflonum* squared
libraries: `(chezscheme)`

The procedure `cfl-magnitude-squared`, when passed an inexact complex argument $a + bi$ returns a flonum representing the magnitude of the argument squared, i.e., $a^2 + b^2$.

See also `magnitude-squared`, which is a generic version of this operator that returns the magnitude squared of any valid representation for a complex number. Both operations are similar to the `magnitude` procedure, which returns the magnitude, $sqrt(a^2 + b^2)$, of its generic complex argument.

```
(cfl-magnitude-squared 3.0)  ⇒  9.0
(cfl-magnitude-squared 3.0-4.0i)  ⇒  25.0
```

8.5. Bitwise and Logical Operators

Chez Scheme provides a set of logical operators that allow exact integers (fixnums and bignums) to be treated as sets or sequences of bits. These operators include `logand` (bitwise logical `and`), `logior` (bitwise logical `or`), `logxor` (bitwise logical exclusive `or`), `lognot` (bitwise logical `not`), `logtest` (test multiple bits), `logbit?` (test single bit), `logbit0` (reset single bit), `logbit1` (set single bit), and `ash` (arithmetic shift). Each of these operators treats its arguments as two's complement integers, regardless of the underlying representation. This treatment can be exploited to represent infinite sets: a negative number represents an infinite number of one bits beyond the leftmost zero, and a nonnegative number represents an infinite number of zero bits beyond the leftmost one bit.

Fixnum equivalents of the logical operators are provided, as `fxlogand`, `fxlogior`, `fxlogxor`, `fxlognot`, `fxlogtest`, `fxlogbit?`, `fxlogbit0`, and `fxlogbit1`. Three separate fixnum operators are provided for shifting: `fxsll` (shift-left logical), `fxsrl` (shift-right logical), `fxsra` (shift-right arithmetic). Logical and arithmetic shifts differ only for right shifts. Shift-right logical shifts in zero bits on the left end, and shift-right arithmetic replicates the sign bit.

Logical shifts do not make sense for arbitrary-precision integers, since these have no "left end" into which bits must be shifted.

`(logand` *int* `...)` **procedure**
returns: the logical "and" of the arguments *int* `...`
libraries: `(chezscheme)`

The arguments must be exact integers (fixnums or bignums) and are treated as two's complement integers, regardless of the underlying representation. With no arguments, `logand` returns -1, i.e., all bits set.

```
(logand)  ⇒  -1
(logand 15)  ⇒  15
(logand -1 -1)  ⇒  -1
(logand -1 0)  ⇒  0
(logand 5 3)  ⇒  1
(logand #x173C8D95 7)  ⇒  5
(logand #x173C8D95 -8)  ⇒  #x173C8D90
(logand #b1100 #b1111 #b1101)  ⇒  #b1100
```

`(logior` *int* `...)` **procedure**
`(logor` *int* `...)` **procedure**
returns: the logical "or" of the arguments *int* `...`
libraries: `(chezscheme)`

The arguments must be exact integers (fixnums or bignums) and are treated as two's

complement integers, regardless of the underlying representation. With no arguments, `logior` returns 0, i.e., all bits reset.

```
(logior)  ⇒  0
(logior 15)  ⇒  15
(logior -1 -1)  ⇒  -1
(logior -1 0)  ⇒  -1
(logior 5 3)  ⇒  7
(logior #b111000 #b101010)  ⇒  #b111010
(logior #b1000 #b0100 #b0010)  ⇒  #b1110
(apply logior '(1 2 4 8 16))  ⇒  31
```

`(logxor` *int* `...)` **procedure**

returns: the logical "exclusive or" of the arguments *int* ...
libraries: `(chezscheme)`

The arguments must be exact integers (fixnums or bignums) and are treated as two's complement integers, regardless of the underlying representation. With no arguments, `logxor` returns 0, i.e., all bits reset.

```
(logxor)  ⇒  0
(logxor 15)  ⇒  15
(logxor -1 -1)  ⇒  0
(logxor -1 0)  ⇒  -1
(logxor 5 3)  ⇒  6
(logxor #b111000 #b101010)  ⇒  #b010010
(logxor #b1100 #b0100 #b0110)  ⇒  #b1110
```

`(lognot` *int*`)` **procedure**

returns: the logical "not" of *int*
libraries: `(chezscheme)`

The argument must be an exact integer (fixnum or bignum) and is treated as a two's complement integer, regardless of the underlying representation.

```
(lognot -1)  ⇒  0
(lognot 0)  ⇒  -1
(lognot 7)  ⇒  -8
(lognot -8)  ⇒  7
```

`(logbit?` *index int*`)` **procedure**

returns: `#t` if the specified bit is set, otherwise `#f`
libraries: `(chezscheme)`

index must be a nonnegative exact integer. *int* must be an exact integer (fixnum or bignum) and is treated as a two's complement integer, regardless of the underlying representation.

`logbit?` returns `#t` if the bit at index *index* of *int* is set (one) and `#f` otherwise. The index is zero-based, counting from the lowest-order toward higher-order bits. There is no upper limit on the index; for nonnegative values of *int*, the bits above the highest order set bit are all considered to be zero, and for negative values, the bits above the highest order reset bit are all considered to be one.

`logbit?` is equivalent to

```
(lambda (k n) (not (zero? (logand n (ash 1 k)))))
```

but more efficient.

```
(logbit? 0 #b1110)  ⇒  #f
(logbit? 1 #b1110)  ⇒  #t
(logbit? 2 #b1110)  ⇒  #t
(logbit? 3 #b1110)  ⇒  #t
(logbit? 4 #b1110)  ⇒  #f
(logbit? 100 #b1110)  ⇒  #f

(logbit? 0 -6)  ⇒  #f  ; the two's complement of -6 is 1...1010
(logbit? 1 -6)  ⇒  #t
(logbit? 2 -6)  ⇒  #f
(logbit? 3 -6)  ⇒  #t
(logbit? 100 -6)  ⇒  #t

(logbit? (random 1000000) 0)  ⇒  #f
(logbit? (random 1000000) -1)  ⇒  #t

(logbit? 20000 (ash 1 20000))  ⇒  #t
```

(logtest int_1 int_2**)** **procedure**

returns: `#t` if any common bits are set, otherwise `#f`
libraries: `(chezscheme)`

The arguments must be exact integers (fixnums or bignums) and are treated as two's complement integers, regardless of the underlying representation.

`logtest` returns `#t` if any bit set in one argument is also set in the other. It returns `#f` if the two arguments have no set bits in common.

`logtest` is equivalent to

```
(lambda (n1 n2) (not (zero? (logand n1 n2))))
```

but more efficient.

```
(logtest #b10001 #b1110)  ⇒  #f
(logtest #b10101 #b1110)  ⇒  #t
(logtest #b111000 #b110111)  ⇒  #t

(logtest #b101 -6)  ⇒  #f  ; the two's complement of -6 is 1...1010
(logtest #b1000 -6)  ⇒  #t
```

```
(logtest 100 -6)  ⇒  #t

(logtest (+ (random 1000000) 1) 0)  ⇒  #f
(logtest (+ (random 1000000) 1) -1)  ⇒  #t

(logtest (ash #b101 20000) (ash #b111 20000))  ⇒  #t
```

`(logbit0` *index int*`)` **procedure**
returns: the result of clearing bit *index* of *int*
libraries: `(chezscheme)`

index must be a nonnegative exact integer. *int* must be an exact integer (fixnum or bignum) and is treated as a two's complement integer, regardless of the underlying representation.

The index is zero-based, counting from the lowest-order toward higher-order bits. As with `logbit?`, there is no upper limit on the index.

`logbit0` is equivalent to

```
(lambda (i n) (logand (lognot (ash 1 i)) n))
```

but more efficient.

```
(logbit0 3 #b10101010)  ⇒  #b10100010
(logbit0 4 #b10101010)  ⇒  #b10101010
(logbit0 0 -1)  ⇒  -2
```

`(logbit1` *index int*`)` **procedure**
returns: the result of setting bit *index* of *int*
libraries: `(chezscheme)`

index must be a nonnegative exact integer. *int* must be an exact integer (fixnum or bignum) and is treated as a two's complement integer, regardless of the underlying representation.

The index is zero-based, counting from the lowest-order toward higher-order bits. As with `logbit?`, there is no upper limit on the index.

`logbit1` is equivalent to

```
(lambda (i n) (logor (ash 1 i) n))
```

but more efficient.

```
(logbit1 3 #b10101010)  ⇒  #b10101010
(logbit1 4 #b10101010)  ⇒  #b10111010
(logbit1 4 0)  ⇒  #b10000
(logbit1 0 -2)  ⇒  -1
```

(ash *int count*) **procedure**
returns: *int* shifted left arithmetically by *count*.
libraries: (chezscheme)

Both arguments must be exact integers. The first argument is treated as a two's complement integer, regardless of the underlying representation. If *count* is negative, *int* is shifted right by $-count$ bits.

```
(ash 8 0)  ⇒  8
(ash 8 2)  ⇒  32
(ash 8 -2)  ⇒  2
(ash -1 2)  ⇒  -4
(ash -1 -2)  ⇒  -1
```

(fxlogand *fixnum* ...) **procedure**
returns: the logical "and" of the arguments *fixnum* ...
libraries: (chezscheme)

The arguments are treated as two's complement integers, regardless of the underlying representation. With no arguments, `fxlogand` returns -1, i.e., all bits set.

```
(fxlogand)  ⇒  -1
(fxlogand 15)  ⇒  15
(fxlogand -1 -1)  ⇒  -1
(fxlogand -1 0)  ⇒  0
(fxlogand 5 3)  ⇒  1
(fxlogand #b111000 #b101010)  ⇒  #b101000
(fxlogand #b1100 #b1111 #b1101)  ⇒  #b1100
```

(fxlogior *fixnum* ...) **procedure**
(fxlogor *fixnum* ...) **procedure**
returns: the logical "or" of the arguments *fixnum* ...
libraries: (chezscheme)

The arguments are treated as two's complement integers, regardless of the underlying representation. With no arguments, `fxlogior` returns 0, i.e., all bits reset.

```
(fxlogior)  ⇒  0
(fxlogior 15)  ⇒  15
(fxlogior -1 -1)  ⇒  -1
(fxlogior -1 0)  ⇒  -1
(fxlogior #b111000 #b101010)  ⇒  #b111010
(fxlogior #b1000 #b0100 #b0010)  ⇒  #b1110
(apply fxlogior '(1 2 4 8 16))  ⇒  31
```

`(fxlogxor` *fixnum* `...)` **procedure**
returns: the logical "exclusive or" of the arguments *fixnum* `...`
libraries: `(chezscheme)`

The arguments are treated as two's complement integers, regardless of the underlying representation. With no arguments, `fxlogxor` returns 0, i.e., all bits reset.

```
(fxlogxor)  ⇒  0
(fxlogxor 15)  ⇒  15
(fxlogxor -1 -1)  ⇒  0
(fxlogxor -1 0)  ⇒  -1
(fxlogxor 5 3)  ⇒  6
(fxlogxor #b111000 #b101010)  ⇒  #b010010
(fxlogxor #b1100 #b0100 #b0110)  ⇒  #b1110
```

`(fxlognot` *fixnum*`)` **procedure**
returns: the logical "not" of *fixnum*
libraries: `(chezscheme)`

The argument is treated as a two's complement integer, regardless of the underlying representation.

```
(fxlognot -1)  ⇒  0
(fxlognot 0)  ⇒  -1
(fxlognot 1)  ⇒  -2
(fxlognot -2)  ⇒  1
```

`(fxlogbit?` *index fixnum*`)` **procedure**
returns: `#t` if the specified bit is set, otherwise `#f`
libraries: `(chezscheme)`

index must be a nonnegative fixnum. *fixnum* is treated as a two's complement integer, regardless of the underlying representation.

`fxlogbit?` returns `#t` if the bit at index *index* of *fixnum* is set (one) and `#f` otherwise. The index is zero-based, counting from the lowest-order toward higher-order bits. The index is limited only by the fixnum range; for nonnegative values of *fixnum*, the bits above the highest order set bit are all considered to be zero, and for negative values, the bits above the highest order reset bit are all considered to be one.

```
(fxlogbit? 0 #b1110)  ⇒  #f
(fxlogbit? 1 #b1110)  ⇒  #t
(fxlogbit? 2 #b1110)  ⇒  #t
(fxlogbit? 3 #b1110)  ⇒  #t
(fxlogbit? 4 #b1110)  ⇒  #f
(fxlogbit? 100 #b1110)  ⇒  #f
```

```
(fxlogbit? 0 -6)  ⇒  #f  ; the two's complement of -6 is 1...1010
(fxlogbit? 1 -6)  ⇒  #t
(fxlogbit? 2 -6)  ⇒  #f
(fxlogbit? 3 -6)  ⇒  #t
(fxlogbit? 100 -6)  ⇒  #t

(fxlogbit? (random 1000000) 0)  ⇒  #f
(fxlogbit? (random 1000000) -1)  ⇒  #t
```

(fxlogtest *fixnum*$_1$ *fixnum*$_2$**)** **procedure**
returns: `#t` if any common bits are set, otherwise `#f`
libraries: `(chezscheme)`

The arguments are treated as two's complement integers, regardless of the underlying representation.

`fxlogtest` returns `#t` if any bit set in one argument is also set in the other. It returns `#f` if the two arguments have no set bits in common.

```
(fxlogtest #b10001 #b1110)  ⇒  #f
(fxlogtest #b10101 #b1110)  ⇒  #t
(fxlogtest #b111000 #b110111)  ⇒  #t

(fxlogtest #b101 -6)  ⇒  #f  ; the two's complement of -6 is 1...1010
(fxlogtest #b1000 -6)  ⇒  #t
(fxlogtest 100 -6)  ⇒  #t

(fxlogtest (+ (random 1000000) 1) 0)  ⇒  #f
(fxlogtest (+ (random 1000000) 1) -1)  ⇒  #t
```

(fxlogbit0 *index fixnum*) **procedure**
returns: the result of clearing bit *index* of *fixnum*
libraries: `(chezscheme)`

fixnum is treated as a two's complement integer, regardless of the underlying representation. *index* must be nonnegative and less than the number of bits in a fixnum, excluding the sign bit, i.e., less than `(integer-length (most-positive-fixnum))`. The index is zero-based, counting from the lowest-order toward higher-order bits.

`fxlogbit0` is equivalent to

```
(lambda (i n) (fxlogand (fxlognot (fxsll 1 i)) n))
```

but more efficient.

```
(fxlogbit0 3 #b10101010)  ⇒  #b10100010
(fxlogbit0 4 #b10101010)  ⇒  #b10101010
(fxlogbit0 0 -1)  ⇒  -2
```

`(fxlogbit1` *index fixnum*`)` **procedure**
returns: the result of setting bit *index* of *fixnum*
libraries: `(chezscheme)`

fixnum is treated as a two's complement integer, regardless of the underlying representation. *index* must be nonnegative and less than the number of bits in a fixnum, excluding the sign bit, i.e., less than `(integer-length (most-positive-fixnum))`. The index is zero-based, counting from the lowest-order toward higher-order bits.

`fxlogbit1` is equivalent to

```
(lambda (i n) (fxlogor (fxsll 1 i) n))
```

but more efficient.

```
(fxlogbit1 3 #b10101010)  ⇒  #b10101010
(fxlogbit1 4 #b10101010)  ⇒  #b10111010
(fxlogbit1 4 0)  ⇒  #b10000
(fxlogbit1 0 -2)  ⇒  -1
```

`(fxsll` *fixnum count*`)` **procedure**
returns: *fixnum* shifted left by *count*
libraries: `(chezscheme)`

fixnum is treated as a two's complement integer, regardless of the underlying representation. *count* must be nonnegative and not more than the number of bits in a fixnum, i.e., `(+ (integer-length (most-positive-fixnum)) 1)`. An exception is raised with condition-type `&implementation-restriction` if the result cannot be represented as a fixnum.

```
(fxsll 1 2)  ⇒  4
(fxsll -1 2)  ⇒  -4
```

`(fxsrl` *fixnum count*`)` **procedure**
returns: *fixnum* logically shifted right by *count*
libraries: `(chezscheme)`

fixnum is treated as a two's complement integer, regardless of the underlying representation. *count* must be nonnegative and not more than the number of bits in a fixnum, i.e., `(+ (integer-length (most-positive-fixnum)) 1)`.

```
(fxsrl 4 2)  ⇒  1
(= (fxsrl -1 1) (most-positive-fixnum))  ⇒  #t
```

`(fxsra` *fixnum count*`)` **procedure**
returns: *fixnum* arithmetically shifted right by *count*
libraries: `(chezscheme)`

fixnum is treated as a two's complement integer, regardless of the underlying representa-

tion. *count* must be nonnegative and not more than the number of bits in a fixnum, i.e., `(+ (integer-length (most-positive-fixnum)) 1)`.

```
(fxsra 64 3)  ⇒  8
(fxsra -1 1)  ⇒  -1
(fxsra -64 3)  ⇒  -8
```

8.6. Random Number Generation

`(random` *real*`)` **procedure**

returns: a nonnegative pseudo-random number less than *real*
libraries: `(chezscheme)`

real must be a positive integer or positive inexact real number.

```
(random 1)  ⇒  0
(random 1029384535235)  ⇒  1029384535001, every now and then
(random 1.0)  ⇒  0.5, every now and then
```

`random-seed` **thread parameter**

libraries: `(chezscheme)`

The random number generator allows the current random seed to be obtained and modified via the parameter `random-seed`.

When called without arguments, `random-seed` returns the current random seed. When called with one argument, which must be a nonnegative exact integer ranging from 1 through $2^{32} - 1$, `random-seed` sets the current random seed to the argument.

```
(let ([s (random-seed)])
  (let ([r1 (random 1.0)])
    (random-seed s)
    (eqv? (random 1.0) r1)))  ⇒  #t
```

8.7. Miscellaneous Numeric Operations

`(=` num_1 num_2 num_3 `...)` **procedure**
`(<` $real_1$ $real_2$ $real_3$ `...)` **procedure**
`(>` $real_1$ $real_2$ $real_3$ `...)` **procedure**
`(<=` $real_1$ $real_2$ $real_3$ `...)` **procedure**
`(>=` $real_1$ $real_2$ $real_3$ `...)` **procedure**

returns: `#t` if the relation holds, `#f` otherwise
libraries: `(chezscheme)`

These predicates are identical to the Revised[6] Report counterparts, except they are extended to accept one or more rather than two or more arguments. When passed one argument, each of these predicates returns `#t`.

```
(> 3/4)  ⇒  #t
(< 3/4)  ⇒  #t
(= 3/4)  ⇒  #t
```

`(1+` *num*`)` **procedure**
`(add1` *num*`)` **procedure**
`(1-` *num*`)` **procedure**
`(-1+` *num*`)` **procedure**
`(sub1` *num*`)` **procedure**

returns: *num* plus 1 or *num* minus 1
libraries: `(chezscheme)`

`1+` and `add1` are equivalent to `(lambda (x) (+ x 1))`; `1-`, `-1+`, and `sub1` are equivalent to `(lambda (x) (- x 1))`.

```
(define plus
 ; x should be a nonnegative integer
  (lambda (x y)
    (if (zero? x)
        y
        (plus (1- x) (1+ y)))))

(plus 7 8)  ⇒  15

(define double
 ; x should be a nonnegative integer
  (lambda (x)
    (if (zero? x)
        0
        (add1 (add1 (double (sub1 x)))))))

(double 7)  ⇒  14
```

`(expt-mod` int_1 int_2 int_3`)` **procedure**

returns: int_1 raised to the int_2 power, modulo int_3
libraries: `(chezscheme)`

int_1, int_2 and int_3 must be nonnegative integers. `expt-mod` performs its computation in such a way that the intermediate results are never much larger than int_3. This means that when int_2 is large, `expt-mod` is more efficient than the equivalent procedure `(lambda (x y z) (modulo (expt x y) z))`.

```
(expt-mod 2 4 3)  ⇒  1
(expt-mod 2 76543 76543)  ⇒  2
```

`(isqrt` *n*`)` **procedure**
returns: the integer square root of *n*
libraries: `(chezscheme)`

n must be a nonnegative integer. The integer square root of *n* is defined to be $\lfloor\sqrt{n}\rfloor$.

```
(isqrt 0)  ⇒  0
(isqrt 16)  ⇒  4
(isqrt 16.0)  ⇒  4.0
(isqrt 20)  ⇒  4
(isqrt 20.0)  ⇒  4.0
(isqrt (* 2 (expt 10 20)))  ⇒  14142135623
```

`(integer-length` *n*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

The procedure `integer-length` returns the length in bits of the smallest two's complement representation for *n*, with an assumed leading 1 (sign) bit for negative numbers. For zero, `integer-length` returns 0.

```
(integer-length 0)  ⇒  0
(integer-length 1)  ⇒  1
(integer-length 2)  ⇒  2
(integer-length 3)  ⇒  2
(integer-length 4)  ⇒  3
(integer-length #b10000000)  ⇒  8
(integer-length #b11111111)  ⇒  8
(integer-length -1)  ⇒  0
(integer-length -2)  ⇒  1
(integer-length -3)  ⇒  2
(integer-length -4)  ⇒  2
```

`(nonpositive?` *real*`)` **procedure**
returns: `#t` if *real* is not greater than zero, `#f` otherwise
libraries: `(chezscheme)`

`nonpositive?` is equivalent to `(lambda (x) (<= x 0))`.

```
(nonpositive? 128)  ⇒  #f
(nonpositive? 0.0)  ⇒  #t
(nonpositive? 1.8e-15)  ⇒  #f
(nonpositive? -2/3)  ⇒  #t
```

`(nonnegative?` *real*`)` **procedure**
returns: `#t` if *real* is not less than zero, `#f` otherwise
libraries: `(chezscheme)`

`nonnegative?` is equivalent to `(lambda (x) (>= x 0))`.

```
(nonnegative? -65)  ⇒  #f
(nonnegative? 0)  ⇒  #t
(nonnegative? -0.0121)  ⇒  #f
(nonnegative? 15/16)  ⇒  #t
```

`(conjugate` *num*`)` **procedure**
returns: complex conjugate of *num*
libraries: `(chezscheme)`

The procedure `conjugate`, when passed a complex argument $a + bi$, returns its complex conjugate $a + (-b)i$.

```
(conjugate 3.0+4.0i)  ⇒  3.0-4.0i
(conjugate 1e-20-2e-30i)  ⇒  1e-20+2e-30i
(conjugate 3)  ⇒  3
```

`(magnitude-squared` *num*`)` **procedure**
returns: magnitude of *num* squared
libraries: `(chezscheme)`

The procedure `magnitude-squared`, when passed a complex argument $a + bi$ returns its magnitude squared, i.e., $a^2 + b^2$.

```
(magnitude-squared 3.0-4.0i)  ⇒  25.0
(magnitude-squared 3.0)  ⇒  9.0
```

`(sinh` *num*`)` **procedure**
`(cosh` *num*`)` **procedure**
`(tanh` *num*`)` **procedure**
returns: the hyperbolic sine, cosine, or tangent of *num*
libraries: `(chezscheme)`

```
(sinh 0.0)  ⇒  0.0
(cosh 0.0)  ⇒  1.0
(tanh -0.0)  ⇒  -0.0
```

(asinh *num*) **procedure**
(acosh *num*) **procedure**
(atanh *num*) **procedure**
returns: the hyperbolic arc sine, arc cosine, or arc tangent of *num*
libraries: (chezscheme)

```
(acosh 0.0)  ⇒  0.0+1.5707963267948966i
(acosh 1.0)  ⇒  0.0
(atanh -1.0)  ⇒  -inf.0
```

(string->number *string*) **procedure**
(string->number *string radix*) **procedure**
returns: the number represented by *string*, or #f
libraries: (chezscheme)

This procedure is identical to the Revised[6] Report version except that radix may be any exact integer between 2 and 36, inclusive. The Revised[6] Report version requires radix to be in the set $\{2, 8, 10, 16\}$.

```
(string->number "211012" 3)  ⇒  559
(string->number "tobeornottobe" 36)  ⇒  14061368915981283669 8
```

(number->string *num*) **procedure**
(number->string *num radix*) **procedure**
(number->string *num radix precision*) **procedure**
returns: an external representation of *num* as a string
libraries: (chezscheme)

This procedure is identical to the Revised[6] Report version except that radix may be any exact integer between 2 and 36, inclusive. The Revised[6] Report version requires radix to be in the set $\{2, 8, 10, 16\}$.

```
(number->string 10000 4)  ⇒  "2130100"
(number->string 10000 27)  ⇒  "DJA"
```

9. Input/Output Operations

This chapter describes *Chez Scheme*'s generic port facility, operations on ports, and various *Chez Scheme* extensions to the standard set of input/output operations. See Chapter 7 of *The Scheme Programming Language, 4th Edition* or the Revised[6] Report on Scheme for a description of standard input/output operations. Definitions of a few sample generic ports are given in Section 9.17.

Chez Scheme closes file ports automatically after they become inaccessible to the program or when the Scheme program exits, but it is best to close ports explicitly whenever possible.

9.1. Generic Ports

Chez Scheme's "generic port" facility allows the programmer to add new types of textual ports with arbitrary input/output semantics. It may be used, for example, to define any of the built-in Common Lisp [30] stream types, i.e., synonym streams, broadcast streams, concatenated streams, two-way streams, echo streams, and string streams. It may also be used to define more exotic ports, such as ports that represent windows on a bit-mapped display or ports that represent processes connected to the current process via pipes or sockets.

Each port has an associated *port handler*. A port handler is a procedure that accepts messages in an object-oriented style. Each message corresponds to one of the low-level Scheme operations on ports, such as `read-char` and `close-input-port` (but not `read`, which is defined in terms of the lower-level operations). Most of these operations simply call the handler immediately with the corresponding message.

Standard messages adhere to the following conventions: the message name is the first argument to the handler. It is always a symbol, and it is always the name of a primitive Scheme operation on ports. The additional arguments are the same as the arguments to the primitive procedure and occur in the same order. (The port argument to some of the primitive procedures is optional; in the case of the messages passed to a handler, the port argument is always supplied.) The following messages are defined for built-in ports:

`block-read` *port string count*
`block-write` *port string count*
`char-ready?` *port*
`clear-input-port` *port*
`clear-output-port` *port*

`close-port` *port*
`file-position` *port*
`file-position` *port position*
`file-length` *port*
`flush-output-port` *port*
`peek-char` *port*
`port-name` *port*
`read-char` *port*
`unread-char` *char port*
`write-char` *char port*

Additional messages may be accepted by user-defined ports.

Chez Scheme input and output is normally buffered for efficiency. To support buffering, each input port contains an input buffer and each output port contains an output buffer. Bidirectional ports, ports that are both input ports and output ports, contain both input and output buffers. Input is not buffered if the input buffer is the empty string, and output is not buffered if the output buffer is the empty string. In the case of unbuffered input and output, calls to `read-char`, `write-char`, and similar messages cause the handler to be invoked immediately with the corresponding message. For buffered input and output, calls to these procedures cause the buffer to be updated, and the handler is not called under normal circumstances until the buffer becomes empty (for input) or full (for output). Handlers for buffered ports must *not* count on the buffer being empty or full when `read-char`, `write-char`, and similar messages are received, however, due to the possibility that (a) the handler is invoked through some other mechanism, or (b) the call to the handler is interrupted.

In the presence of keyboard, timer, and other interrupts, it is possible for a call to a port handler to be interrupted or for the handler itself to be interrupted. If the port is accessible outside of the interrupted code, there is a possibility that the interrupt handler will cause input or output to be performed on the port. This is one reason, as stated above, that port handlers must not count on the input buffer being empty or output buffer being full when a `read-char`, `write-char`, or similar message is received. In addition, port handlers may need to manipulate the buffers only with interrupts disabled (using `with-interrupts-disabled`).

Generic ports are created via one of the port construction procedures `make-input-port`, `make-output-port`, and `make-input/output-port` defined later in this chapter. Ports have seven accessible fields:

handler, accessed with `port-handler`;
output-buffer, accessed with `port-output-buffer`,
output-size, accessed with `port-output-size`,
output-index, accessed with `port-output-index`,
input-buffer, accessed with `port-input-buffer`,
input-size, accessed with `port-input-size`, and
input-index, accessed with `port-input-index`.

The output-size and output-index fields are valid only for output ports, and the input-size

and input-index fields are valid only for input ports. The output and input size and index fields may be updated as well using the corresponding "`set-`*field*`!`" procedure.

A port's output size determines how much of the port's output buffer is actually available for writing by `write-char`. The output size is often the same as the string length of the port's output buffer, but it can be set to less (but no less than zero) at the discretion of the programmer. The output index determines to which position in the port's buffer the next character will be written. The output index should be between 0 and the output size, inclusive. If no output has occurred since the buffer was last flushed, the output index should be 0. If the index is less than the size, `write-char` stores its character argument into the specified character position within the buffer and increments the index. If the index is equal to the size, `write-char` leaves the fields of the port unchanged and invokes the handler.

A port's input size determines how much of the port's input buffer is actually available for reading by `read-char`. A port's input size and input index are constrained in the same manner as output size and index, i.e., the input size must be between 0 and the string length of the input buffer (inclusive), and the input index must be between 0 and the input size (inclusive). Often, the input size is less than the length of the input buffer because there are fewer characters available to read than would fit in the buffer. The input index determines from which position in the input buffer the next character will be read. If the index is less than the size, `read-char` extracts the character in this position, increments the index, and returns the character. If the index is equal to the size, `read-char` leaves the fields of the port unchanged and invokes the handler.

The operation of `peek-char` is similar to that of `read-char`, except that it does not increment the input index. `unread-char` decrements the input index if it is greater than 0, otherwise it invokes the handler. `char-ready?` returns `#t` if the input index is less than the input size, otherwise it invokes the handler.

Although the fields shown and discussed above are logically present in a port, actual implementation details may differ. The current *Chez Scheme* implementation uses a different representation that allows `read-char`, `write-char`, and similar operations to be open-coded with minimal overhead. The access and assignment operators perform the conversion between the actual representation and the one shown above.

Port handlers receiving a message must return a value appropriate for the corresponding operation. For example, a handler receiving a `read-char` message must return a character or eof object (if it returns). For operations that return unspecified values, such as `close-port`, the handler is not required to return any particular value.

9.2. File Options

The Revised[6] Report requires that the universe of a file-options enumeration set must include `no-create`, `no-fail`, and `no-truncate`, whose meanings are described within the description of the `file-options` syntax in Section 7.2 of *The Scheme Programming Language, 4th Edition*. *Chez Scheme* defines a number of additional file options:

compressed: An output file should be compressed when written; and a compressed input file should be decompressed when read.

replace: For output files only, replace (remove and recreate) the existing file if it exists.

exclusive: For output files only, lock the file for exclusive access. On some systems the lock is advisory, i.e., it inhibits access by other processes only if they also attempt to open exclusively.

append: For output files only, position the output port at the end of the file before each write so that output to the port is always appended to the file.

perm-set-user-id: For newly created output files under Unix-based systems only, set user-id bit.

perm-set-group-id: For newly created output files under Unix-based systems only, set group-id bit.

perm-sticky: For newly created output files under Unix-based systems only, set sticky bit.

perm-no-user-read: For newly created output files under Unix-based systems only, do not set user read bit. (User read bit is set by default, unless masked by the process umask.)

perm-no-user-write: For newly created output files under Unix-based systems only, do not set user write bit. (User write bit is set by default, unless masked by the process umask.)

perm-user-execute: For newly created output files under Unix-based systems only, set user execute bit unless masked by process umask. (User execute bit is not set by default.)

perm-no-group-read: For newly created output files under Unix-based systems only, do not set group read bit. (Group read bit is set by default, unless masked by the process umask.)

perm-no-group-write: For newly created output files under Unix-based systems only, do not set group write bit. (Group write bit is set by default, unless masked by the process umask.)

perm-group-execute: For newly created output files under Unix-based systems only, set group execute bit unless masked by process umask. (Group execute bit is not set by default.)

perm-no-other-read: For newly created output files under Unix-based systems only, do not set other read bit. (Other read bit is set by default, unless masked by the process umask.)

perm-no-other-write: For newly created output files under Unix-based systems only, do not set other write bit. (Other write bit is set by default, unless masked by the process umask.)

perm-other-execute: For newly created output files under Unix-based systems only, set other execute bit unless masked by process umask. (Other execute bit is not set by default.)

9.3. Transcoders

The language of the Revised[6] Report provides three built-in codecs: a latin-1 codec, a utf-8 codec, and a utf-16 codec. *Chez Scheme* provides three additional codecs: a utf-16le codec, utf-16be codec, and an "iconv" codec for non-Unicode character sets. It also provides an alternative to the standard utf-16 codec that defaults to little-endian format rather than the default big-endian format. This section describes these codecs, plus a `current-transcoder` parameter that allows the programmer to determine the transcoder used for a textual port whenever the transcoder is implicit, as for `open-input-file` or `load`, along with the predicate `transcoder?`, which should be standard but is not.

`(utf-16-codec)` **procedure**
`(utf-16-codec` *endianness*`)` **procedure**
`(utf-16le-codec)` **procedure**
`(utf-16be-codec)` **procedure**
returns: a codec
libraries: `(chezscheme)`

endianness must be the symbol `big` or the symbol `little`.

The codec returned by `utf-16-codec` can be used to create process data written UFT-16 format. When called without the *endianness* argument or with *endianness* `big`, `utf-16-codec` returns a codec for standard UTF-16 data, i.e., one that defaults to big-endian format if no byte-order mark (BOM) is found.

When output is transcoded with a transcoder based on this codec, a BOM is emitted just before the first character written, and each character is written as a UTF-16 character in big-endian format. For input, a BOM is looked for at the start of the input and, if present, controls the byte order of the remaining UTF-16 characters. If no BOM is present, big-endian order is assumed. For input-output ports, the BOM is not emitted if the file is read before written, and a BOM is not looked for if the file is written before read.

For textual ports created via `transcoded-port`, a BOM written or read via the transcoder appears at the beginning of the underlying data stream or file only if the binary port passed to `transcoded-port` is positioned at the start of the data stream or file. When the transcoder can determine this is the case, it sets a flag that causes `set-port-position!`) to position the port beyond the BOM if an attempt is made to reposition the port to the start of the data stream or file, so that the BOM is preserved.

When called with *endianness* `little`, `utf-16-codec` returns a codec that defaults to the little-endian format both for reading and for writing. For output-only streams or input/output streams that are written before read, the result is standard UTF-16, with a BOM that specifies little-endian format followed by characters in little-endian byte order. For input-only streams or input/output streams that are read before written, this codec allows programs to read from input streams that either begin with a BOM or are encoded in UTF-16LE format. This is particularly useful for handling files that might have been produced by older Windows applications that claim to produce UTF-16 files but actually produce UTF-16LE files.

The Revised[6] Report version of `utf-16-codec` lacks the optional *endianness* argument.

The codecs returned by `utf-16le-codec` and `utf-16be-codec` are used to read and write data in the UTF-16LE and UTF-16BE formats, i.e., UTF-16 with little-endian or big-endian byte order and no BOM. For output, these codecs are useful for controlling whether and where the BOM is emitted, since no BOM is emitted implicitly and a BOM can be emitted explicitly as an ordinary character. For input, these codecs are useful for processing files known to be in little-endian or big-endian format with no BOM.

`(iconv-codec` *code-page*`)` **procedure**
returns: a codec
libraries: `(chezscheme)`

code-page must be a string and should identify a codec accepted by the `iconv` library installed on the target machine. The codec returned by this procedure can be used to convert from the non-Unicode single- and multiple-byte character sets supported by `iconv`. When used in the input direction, the codec converts byte sequences into Scheme strings, and when used in the output direction, it converts Scheme strings to byte sequences.

The set of supported code pages depends on the version of `iconv` available; consult the `iconv` documentation or use the shell command `iconv --list` to obtain a list of supported code pages.

While the Windows operating system does not supply an `iconv` library, it is possible to use `iconv-codec` on Windows systems by supplying an `iconv` dynamic-link library (named `iconv.dll`, `libiconv.dll`, or `libiconv-2.dll`) that provides Posix-conformant `iconv_open`, `iconv`, and `iconv_close` entry points either under those names or under the alternative names `libiconv_open`, `libiconv`, and `libiconv_close`. The dll must be located in a standard location for dlls or in the current directory of the process the first time `iconv-codec` is called.

`current-transcoder` **thread parameter**
libraries: `(chezscheme)`

The transcoder value of the `current-transcoder` parameter is used whenever a textual file is opened with an implicit transcoder, e.g., by `open-input-file` and other convenience I/O procedures, `compile-file` `include`, `load`, and `pretty-file`. Its initial value is the value of the `native-transcoder` procedure.

`(transcoder?` *obj*`)` **procedure**
returns: `#t` if *obj* is a transcoder, `#f` otherwise
libraries: `(chezscheme)`

9.4. Port Operations

The procedures used to create, access, and alter ports directly are described in this section. Also described are several nonstandard operations on ports.

Unless otherwise specified, procedures requiring either input ports or output ports as arguments accept input/output ports as well, i.e., an input/output port is both an input port and an output port.

`(make-input-port` *handler input-buffer*`)` **procedure**
`(make-output-port` *handler output-buffer*`)` **procedure**
`(make-input/output-port` *handler input-buffer output-buffer*`)` **procedure**
returns: a new textual port
libraries: `(chezscheme)`

handler must be a procedure, and *input-buffer* and *output-buffer* must be strings. Each procedure creates a generic port. The handler associated with the port is *handler*, the input buffer is *input-buffer*, and the output buffer is *output-buffer*. For `make-input-port`, the output buffer is undefined, and for `make-output-port`, the input buffer is undefined.

The input size of an input or input/output port is initialized to the string length of the input buffer, and the input index is set to 0. The output size and index of an output or input/output port are initialized similarly.

The length of an input or output buffer may be zero, in which case buffering is effectively disabled.

`(port-handler` *port*`)` **procedure**
returns: a procedure
libraries: `(chezscheme)`

For generic ports, `port-handler` returns the handler passed to one of the generic port creation procedures described above. For ports created by `open-input-file` and similar procedures, `port-handler` returns an internal handler that may be invoked in the same manner as any other handler.

`(port-input-buffer` *input-port*`)` **procedure**
`(port-input-size` *input-port*`)` **procedure**
`(port-input-index` *input-port*`)` **procedure**
`(textual-port-input-buffer` *textual-input-port*`)` **procedure**
`(textual-port-input-size` *textual-input-port*`)` **procedure**
`(textual-port-input-index` *textual-input-port*`)` **procedure**
`(binary-port-input-buffer` *binary-input-port*`)` **procedure**
`(binary-port-input-size` *binary-input-port*`)` **procedure**
`(binary-port-input-index` *binary-input-port*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

These procedures return the input buffer, size, or index of the input port. The variants specialized to textual or binary ports are slightly more efficient than their generic counterparts.

(set-port-input-index! *input-port n*) **procedure**
(set-port-input-size! *input-port n*) **procedure**
(set-port-input-buffer! *input-port x*) **procedure**
(set-textual-port-input-index! *textual-input-port n*) **procedure**
(set-textual-port-input-size! *textual-input-port n*) **procedure**
(set-textual-port-input-buffer! *textual-input-port string*) **procedure**
(set-binary-port-input-index! *binary-input-port n*) **procedure**
(set-binary-port-input-size! *binary-input-port n*) **procedure**
(set-binary-port-input-buffer! *binary-input-port bytevector*) **procedure**
returns: unspecified
libraries: (chezscheme)

The procedure `set-port-input-index!` sets the input index field of *input-port* to n, which must be a nonnegative integer less than or equal to the port's input size.

The procedure `set-port-input-size!` sets the input size field of *input-port* to n, which must be a nonnegative integer less than or equal to the string length of the port's input buffer. It also sets the input index to 0.

The procedure `set-port-input-buffer!` sets the input buffer field of *input-port* to x, which must be a string for textual ports and a bytevector for binary ports. It also sets the input size to the length of the string or bytevector and the input index to 0.

The variants specialized to textual or binary ports are slightly more efficient than their generic counterparts.

(port-input-count *input-port*) **procedure**
(textual-port-input-count *textual-input-port*) **procedure**
(binary-port-input-count *binary-input-port*) **procedure**
returns: see below
libraries: (chezscheme)

These procedures return an exact integer representing the number of characters or bytes left to be read from the port's input buffer, i.e., the difference between the buffer size and index.

The variants specialized to textual or binary ports are slightly more efficient than their generic counterpart.

(port-input-empty? *input-port*) **procedure**
returns: `#t` if the port's input buffer contains no more data, otherwise `#f`
libraries: (chezscheme)

This procedure determines whether the port's input count is zero without computing or returning the actual count.

`(port-output-buffer` *output-port*`)` **procedure**
`(port-output-size` *output-port*`)` **procedure**
`(port-output-index` *output-port*`)` **procedure**
`(textual-port-output-buffer` *output-port*`)` **procedure**
`(textual-port-output-size` *output-port*`)` **procedure**
`(textual-port-output-index` *output-port*`)` **procedure**
`(binary-port-output-buffer` *output-port*`)` **procedure**
`(binary-port-output-size` *output-port*`)` **procedure**
`(binary-port-output-index` *output-port*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

These procedures return the output buffer, size, or index of the output port. The variants specialized to textual or binary ports are slightly more efficient than their generic counterparts.

`(set-port-output-index!` *output-port n*`)` **procedure**
`(set-port-output-size!` *output-port n*`)` **procedure**
`(set-port-output-buffer!` *output-port x*`)` **procedure**
`(set-textual-port-output-index!` *textual-output-port n*`)` **procedure**
`(set-textual-port-output-size!` *textual-output-port n*`)` **procedure**
`(set-textual-port-output-buffer!` *textual-output-port string*`)` **procedure**
`(set-binary-port-output-index!` *output-port n*`)` **procedure**
`(set-binary-port-output-size!` *output-port n*`)` **procedure**
`(set-binary-port-output-buffer!` *binary-output-port bytevector*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

The procedure `set-port-output-index!` sets the output index field of the output port to n, which must be a nonnegative integer less than or equal to the port's output size.

The procedure `set-port-output-size!` sets the output size field of the output port to n, which must be a nonnegative integer less than or equal to the string length of the port's output buffer. It also sets the output index to 0.

The procedure `set-port-output-buffer!` sets the output buffer field of *output-port* to x, which must be a string for textual ports and a bytevector for binary ports. It also sets the output size to the length of the string or bytevector and the output index to 0.

The variants specialized to textual or binary ports are slightly more efficient than their generic counterparts.

`(port-output-count` *output-port*`)` **procedure**
`(textual-port-output-count` *textual-output-port*`)` **procedure**
`(binary-port-output-count` *binary-output-port*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

These procedures return an exact integer representing the amount of space in characters or bytes available to be written in the port's output buffer, i.e., the difference between the buffer size and index.

The variants specialized to textual or binary ports are slightly more efficient than their generic counterpart.

`(port-output-full?` *output-port*`)` **procedure**
returns: `#t` if the port's input buffer has no more room, otherwise `#f`
libraries: `(chezscheme)`

This procedure determines whether the port's output count is zero without computing or returning the actual count.

`(mark-port-closed!` *port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedure directly marks the port closed so that no further input or output operations are allowed on it. It is typically used by handlers upon receipt of a `close-port` message.

`(port-closed?` *port*`)` **procedure**
returns: `#t` if *port* is closed, `#f` otherwise
libraries: `(chezscheme)`

```
(let ([p (open-output-string)])
  (port-closed? p))  ⇒  #f

(let ([p (open-output-string)])
  (close-port p)
  (port-closed? p))  ⇒  #t
```

`(set-port-bol!` *output-port obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

When *obj* is `#f`, the port's beginning-of-line (BOL) flag is cleared; otherwise, the port's BOL flag is set.

The BOL flag is consulted by `fresh-line` (page 240) to determine if it needs to emit a newline. This flag is maintained automatically for file output ports, string output ports,

and transcript ports. The flag is set for newly created file and string output ports, except for file output ports created with the `append` option, for which the flag is reset. The BOL flag is clear for newly created generic ports and never set automatically, but may be set explicitly using `set-port-bol!`. The port is always flushed immediately before the flag is consulted, so it need not be maintained on a per-character basis for buffered ports.

`(port-bol?` *port*`)` **procedure**
returns: `#t` if *port*'s BOL flag is set, `#f` otherwise
libraries: `(chezscheme)`

`(set-port-eof!` *input-port obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

When *obj* is not `#f`, `set-port-eof!` marks *input-port* so that, once its buffer is empty, the port is treated as if it were at eof even if more data is available in the underlying byte or character stream. Once this artificial eof has been read, the eof mark is cleared, making any additional data in the stream available beyond the eof. This feature can be used by a generic port to simulate a stream consisting of multiple input files.

When *obj* is `#f`, the eof mark is cleared.

The following example assumes /dev/zero provides an infinite stream of zero bytes.

```
(define p
  (parameterize ([file-buffer-size 3])
    (open-file-input-port "/dev/zero")))
(set-port-eof! p #t)
(get-u8 p)  ⇒  #!eof
(get-u8 p)  ⇒  0
(set-port-eof! p #t)
(get-u8 p)  ⇒  0
(get-u8 p)  ⇒  0
(get-u8 p)  ⇒  #!eof
(get-u8 p)  ⇒  0
```

`(port-name` *port*`)` **procedure**
returns: the name associated with *port*
libraries: `(chezscheme)`

The name may be any object but is usually a string or `#f` (denoting no name). For file ports, the name is typically a string naming the file.

```
(let ([p (open-input-file "myfile.ss")])
  (port-name p))  ⇒  "myfile.ss"

(let ([p (open-output-string)])
  (port-name p))  ⇒  "string"
```

`(set-port-name!` *port obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedure sets *port*'s name to *obj*, which should be a string or `#f` (denoting no name).

`(port-length` *port*`)` **procedure**
`(file-length` *port*`)` **procedure**
returns: the length of the file or other object to which *port* refers
`(port-has-port-length?` *port*`)` **procedure**
returns: `#t` if the port supports `port-length`, `#f` otherwise
libraries: `(chezscheme)`

A port may allow the length of the underlying stream of characters or bytes to be determined. If so, the procedure `port-has-port-length?` returns `#t` and `port-length` returns the current length. For binary ports, the length is always an exact nonnegative integer byte count. For textual ports, the representation of a length is unspecified; it may not be an exact nonnegative integer and, even if it is, it may not represent either a byte or character count. The length may be used at some later time to reset the length if the port supports `set-port-length!`. If `port-length` is called on a port that does not support it, an exception with condition type `&assertion` is raised.

File lengths beyond 2^{32} might not be reported property for compressed files on 32-bit versions of the system.

`file-length` is identical to `port-length`.

`(set-port-length!` *port len*`)` **procedure**
returns: unspecified
`(port-has-set-port-length!?` *port*`)` **procedure**
returns: `#t` if the port supports `set-port-length!`, `#f` otherwise
libraries: `(chezscheme)`

A port may allow the length of the underlying stream of characters or bytes to be set, i.e., extended or truncated. If so, the procedure `port-has-set-port-length!?` returns `#t` and `set-port-length!` changes the length. For binary ports, the length *len* must be an exact nonnegative integer byte count. For textual ports, the representation of a length is unspecified, as described in the entry for `port-length` above, but *len* must be an appropriate length for the textual port, which is usually guaranteed to be the case only if it was obtained from a call to `port-length` on the same port. If `set-port-length!` is called on a port that does not support it, an exception with condition type `&assertion` is raised.

It is not possible to set the length of a port opened with compression to an arbitrary position, and the result of an attempt to set the length of a compressed file beyond 2^{32} on 32-bit versions of the system is undefined.

`(port-nonblocking?` *port*`)` **procedure**
returns: `#t` if the port is in nonblocking mode, `#f` otherwise
`(port-has-port-nonblocking??` *port*`)` **procedure**
returns: `#t` if the port supports `port-nonblocking?`, `#f` otherwise
libraries: `(chezscheme)`

A port may allow the nonblocking status of the port to be determined. If so, the procedure `port-has-port-nonblocking??` returns `#t` and `port-nonblocking?` returns a boolean value reflecting whether the port is in nonblocking mode.

`(set-port-nonblocking!` *port obj*`)` **procedure**
returns: unspecified
`(port-has-set-port-nonblocking!?` *port*`)` **procedure**
returns: `#t` if the port supports `set-port-nonblocking!`, `#f` otherwise
libraries: `(chezscheme)`

A port may allow reads or writes to be performed in a "nonblocking" fashion. If so, the procedure `port-has-set-port-nonblocking!?` returns `#t` and `set-port-nonblocking!` sets the port to nonblocking mode (if *obj* is a true value) or blocking mode (if *obj* is `#f`). If `set-port-nonblocking!` is called on a port that does not support it, an exception with condition type `&assertion` is raised.

Ports created by the standard Revised[6] port opening procedures are initially set in blocking mode by default. The same is true for most of the procedures described in this document. A generic port based on a nonblocking source may be nonblocking initially. A port returned by `open-fd-input-port`, `open-fd-output-port`, or `open-fd-input/output-port` is initially in nonblocking mode if the file-descriptor passed in is in nonblocking mode. Similarly, a port returned by `standard-input-port`, `standard-output-port`, or `standard-error-port` is initially in nonblocking mode if the underlying stdin, stdout, or stderr file descriptor is in nonblocking mode.

Although `get-bytevector-some` and `get-string-some` normally cannot return an empty bytevector or empty string, they can if the port is in nonblocking mode and no input is available. Also, `get-bytevector-some!` and `get-string-some!` may not read any data if the port is in nonblocking mode and no data is available. Similarly, `put-bytevector-some` and `put-string-some` may not write any data if the port is in nonblocking mode and no room is available.

Nonblocking mode is not supported under Windows.

`(file-position` *port*`)` **procedure**
`(file-position` *port pos*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

When the second argument is omitted, this procedure behaves like the R6RS `port-position` procedure, and when present, like the R6RS `set-port-position!` procedure.

For compressed files opened with the `compressed` flag, `file-position` returns the position in the uncompressed stream of data. Changing the position of a compressed input file opened with the `compressed` flag generally requires rewinding and rereading the file and might thus be slow. The position of a compressed output file opened with the `compressed` flag can be moved forward only; this is accomplished by writing a (compressed) sequence of zeros. File positions beyond 2^{32} might not be reported property for compressed files on 32-bit versions of the system.

`(clear-input-port)` **procedure**
`(clear-input-port` *input-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

If *input-port* is not supplied, it defaults to the current input port. This procedure discards any data in the buffer associated with *input-port*. This may be necessary, for example, to clear any type-ahead from the keyboard in preparation for an urgent query.

`(clear-output-port)` **procedure**
`(clear-output-port` *output-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

If *output-port* is not supplied, it defaults to the current output port. This procedure discards any data in the buffer associated with *output-port*. This may be necessary, for example, to clear any pending output on an interactive port in preparation for an urgent message.

`(flush-output-port)` **procedure**
`(flush-output-port` *output-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

If *output-port* is not supplied, it defaults to the current output port. This procedure forces any data in the buffer associated with *output-port* to be printed immediately. The console output port is automatically flushed after a newline and before input from the console input port; all ports are automatically flushed when they are closed. `flush-output-port` may be necessary, however, to force a message without a newline to be sent to the console output port or to force output to appear on a file without delay.

`(port-file-compressed!` *port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

port must be an input or an output port, but not an input/output port. It must be a file port pointing to a regular file, i.e., a file on disk rather than, e.g., a socket. The port can be a binary or textual port. If the port is an output port, subsequent output sent to the port

will be compressed. If the port is an input port, subsequent input will be decompressed if and only if the port is currently pointing at compressed data. This procedure has no effect if the port is already set for compression.

9.5. String Ports

String ports allow the creation and manipulation of strings via port operations. The procedure `open-input-string` converts a string into a textual input port, allowing the characters in the string to be read in sequence via input operations such as `read-char` or `read`. The procedure `open-output-string` allows new strings to be built up with output operations such as `write-char` and `write`.

While string ports could be defined as generic ports, they are instead supported as primitive by the implementation.

`(open-input-string` *string*`)` **procedure**
returns: a new string input port
libraries: `(chezscheme)`

A string input port is similar to a file input port, except that characters and objects drawn from the port come from *string* rather than from a file.

A string port is at "end of file" when the port reaches the end of the string. It is not necessary to close a string port, although it is okay to do so.

```
(let ([p (open-input-string "hi mom!")])
  (let ([x (read p)])
    (list x (read p))))  ⇒  (hi mom!)
```

`(with-input-from-string` *string thunk*`)` **procedure**
returns: the values returned by *thunk*
libraries: `(chezscheme)`

thunk must be a procedure and should accept zero arguments. `with-input-from-string` parameterizes the current input port to be the result of opening *string* for input during the application of *thunk*.

```
(with-input-from-string "(cons 3 4)"
  (lambda ()
    (eval (read))))  ⇒  (3 . 4)
```

`(open-output-string)` **procedure**
returns: a new string output port
libraries: `(chezscheme)`

A string output port is similar to a file output port, except that characters and objects

written to the port are placed in a string (which grows as needed) rather than to a file. The string built by writing to a string output port may be obtained with `get-output-string`. See the example given for `get-output-string` below. It is not necessary to close a string port, although it is okay to do so.

`(get-output-string` *string-output-port*`)` **procedure**
returns: the string associated with *string-output-port*
libraries: `(chezscheme)`

string-output-port must be an port returned by `open-output-string`.

As a side effect, `get-output-string` resets *string-output-port* so that subsequent output to *string-output-port* is placed into a fresh string.

```
(let ([p (open-output-string)])
  (write 'hi p)
  (write-char #\space p)
  (write 'mom! p)
  (get-output-string p))  ⇒  "hi mom!"
```

An implementation of `format` (Section 9.13) might be written using string-output ports to produce string output.

`(with-output-to-string` *thunk*`)` **procedure**
returns: a string containing the output
libraries: `(chezscheme)`

thunk must be a procedure and should accept zero arguments. `with-output-to-string` parameterizes the current output port to a new string output port during the application of *thunk*. If *thunk* returns, the string associated with the new string output port is returned, as with `get-output-string`.

```
(with-output-to-string
  (lambda ()
    (display "Once upon a time ...")
    (newline)))  ⇒  "Once upon a time ...\n"
```

9.6. File Ports

`file-buffer-size` **thread parameter**
libraries: `(chezscheme)`

`file-buffer-size` is a parameter that determines the size of each buffer created when the buffer mode is not `none` for a port created by one of the file open operations, e.g., `open-input-file` or `open-file-output-port`. When called with no arguments, the parameter returns the current buffer size. When called with a positive fixnum k, it sets the current buffer size to k.

`(file-port?` *port*`)` **procedure**
returns: `#t` if *port* is a file port, `#f` otherwise
libraries: `(chezscheme)`

A file port is any port based directly on an O/S file descriptor, e.g., one created by `open-file-input-port`, `open-output-port`, `open-fd-input-port`, etc., but not a string, bytevector, or custom port.

`(port-file-descriptor` *port*`)` **procedure**
returns: the file descriptor associated with *port*
libraries: `(chezscheme)`

port must be a file port, i.e., a port for which *file-port?* returns `#t`.

9.7. Custom Ports

`custom-port-buffer-size` **thread parameter**
libraries: `(chezscheme)`

`custom-port-buffer-size` is a parameter that determines the sizes of the buffers associated with newly created custom ports. When called with no arguments, the parameter returns the current buffer size. When called with a positive fixnum k, it sets the current buffer size to k.

9.8. Input Operations

`console-input-port` **global parameter**
libraries: `(chezscheme)`

`console-input-port` is a parameter that determines the input port used by the waiter and interactive debugger. When called with no arguments, it returns the console input port. When called with one argument, which must be a textual input port, it changes the value of the console input port. The initial value of this parameter is a port tied to the standard input (stdin) stream of the Scheme process.

`current-input-port` **thread parameter**
libraries: `(chezscheme)`

`current-input-port` is a parameter that determines the default port argument for most input procedures, including `read-char`, `peek-char`, and `read`, When called with no arguments, `current-input-port` returns the current input port. When called with one argument, which must be a textual input port, it changes the value of the current input port. The Revised[6] Report version of `current-input-port` accepts only zero arguments, i.e., it cannot be used to change the current input port. The initial value of this parameter is the same port as the initial value of `console-input-port`.

`(open-input-file` *path*`)` **procedure**
`(open-input-file` *path options*`)` **procedure**
returns: a new input port
libraries: `(chezscheme)`

path must be a string. `open-input-file` opens a textual input port for the file named by *path*. An exception is raised with condition type `&i/o-filename` if the file does not exist or cannot be opened for input.

options, if present, is a symbolic option name or option list. Possible symbolic option names are `compressed`, `uncompressed`, `buffered`, and `unbuffered`. An option list is a list containing zero or more symbolic option names.

The mutually exclusive `compressed` and `uncompressed` options determine whether the input file should be decompressed if it is compressed. (See `open-output-file`.) The default is `uncompressed`, so the `uncompressed` option is useful only as documentation.

The mutually exclusive `buffered` and `unbuffered` options determine whether input is buffered. When input is buffered, it is read in large blocks and buffered internally for efficiency to reduce the number of operating system requests. When the `unbuffered` option is specified, input is unbuffered, but not fully, since one character of buffering is required to support `peek-char` and `unread-char`. Input is buffered by default, so the `buffered` option is useful only as documentation.

For example, the call

```
(open-input-file "frob" '(compressed))
```

opens the file frob with decompression enabled.

The Revised[6] Report version of `open-input-file` does not support the optional *options* argument.

`(call-with-input-file` *path procedure*`)` **procedure**
`(call-with-input-file` *path procedure options*`)` **procedure**
returns: the values returned by *procedure*
libraries: `(chezscheme)`

path must be a string. *procedure* should accept one argument.

`call-with-input-file` creates a new input port for the file named by *path*, as if with `open-input-file`, and passes this port to *procedure*. If *procedure* returns normally, `call-with-input-file` closes the input port and returns the values returned by *procedure*.

`call-with-input-file` does not automatically close the input port if a continuation created outside of *procedure* is invoked, since it is possible that another continuation created inside of *procedure* will be invoked at a later time, returning control to *procedure*. If *procedure* does not return, an implementation is free to close the input port only if it can prove that the input port is no longer accessible. As shown in Section 5.6 of *The Scheme Programming Language, 4th Edition*, `dynamic-wind` may be used to ensure that the port is closed if a continuation created outside of *procedure* is invoked.

See `open-input-file` above for a description of the optional *options* argument.

The Revised[6] Report version of `call-with-input-file` does not support the optional *input* argument.

`(with-input-from-file` *path thunk*`)` **procedure**
`(with-input-from-file` *path thunk options*`)` **procedure**
returns: the values returned by *thunk*
libraries: `(chezscheme)`

path must be a string. *thunk* must be a procedure and should accept zero arguments.

`with-input-from-file` temporarily changes the current input port to be the result of opening the file named by *path*, as if with `open-input-file`, during the application of *thunk*. If *thunk* returns, the port is closed and the current input port is restored to its old value.

The behavior of `with-input-from-file` is unspecified if a continuation created outside of *thunk* is invoked before *thunk* returns. An implementation may close the port and restore the current input port to its old value—but it may not.

See `open-input-file` above for a description of the optional *options* argument.

The Revised[6] Report version of `with-input-from-file` does not support the optional *options* argument.

`(open-fd-input-port` *fd*`)` **procedure**
`(open-fd-input-port` *fd b-mode*`)` **procedure**
`(open-fd-input-port` *fd b-mode ?transcoder*`)` **procedure**
returns: a new input port for the file descriptor *fd*
libraries: `(chezscheme)`

fd must be a nonnegative exact integer and should be a valid open file descriptor. If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure returns a textual input port whose transcoder is *?transcoder*. Otherwise, this procedure returns a binary input port. See the lead-in to Section 7.2 of *The Scheme Programming Language, 4th Edition* for a description of the constraints on and effects of the other arguments.

The file descriptor is closed when the port is closed.

`(standard-input-port)` **procedure**
`(standard-input-port` *b-mode*`)` **procedure**
`(standard-input-port` *b-mode ?transcoder*`)` **procedure**
returns: a new input port connected to the process's standard input
libraries: `(chezscheme)`

If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure returns a textual input port whose transcoder is *?transcoder*. Otherwise, this procedure returns a binary input port. The buffer mode *b-mode* defaults to `block`, which differs from `block` in *Chez Scheme* only for textual output ports. See the lead-in to Section 7.2 of *The Scheme*

Programming Language, 4th Edition for a description of the constraints on and effects of the other arguments.

The Revised[6] Report version of this procedure does not accept the optional *b-mode* and *?transcoder* arguments, which limits it to an implementation-dependent buffering mode (`block` in *Chez Scheme*) and binary output.

`(get-string-some` *textual-input-port*`)` **procedure**
returns: a nonempty string or the eof object
libraries: `(chezscheme)`

If *textual-input-port* is at end of file, the eof object is returned. Otherwise, `get-string-some` reads (as if with `get-u8`) at least one character and possibly more, and returns a string containing these characters. The port's position is advanced past the characters read. The maximum number of characters read by this operation is implementation-dependent.

An exception to the "at least one character" guarantee occurs if the port is in nonblocking mode (see `set-port-nonblocking!`) and no input is ready. In this case, an empty string is returned.

`(get-string-some!` *textual-input-port string start n*`)` **procedure**
returns: the count of characters read, as an exact nonnegative integer, or the eof object
libraries: `(chezscheme)`

start and *n* must be exact nonnegative integers, and the sum of *start* and *n* must not exceed the length of *string*.

If *n* is 0, this procedure returns zero without attempting to read from *textual-input-port* and without modifying *string*.

Otherwise, if *textual-input-port* is at end of file, this procedure returns the eof object, except it returns zero when the port is in nonblocking mode (see `set-port-nonblocking!`) and the port cannot be determined to be at end of file without blocking. In either case, *string* is not modified.

Otherwise, this procedure reads (as if with `get-char`) up to *n* characters from the port, stores the characters in consecutive locations of *string* starting at *start*, advances the port's position just past the characters read, and returns the count of characters read.

If the port is in nonblocking mode, this procedure reads no more than it can without blocking and thus might read zero characters; otherwise, it reads at least one character but no more than are available when the first character becomes available.

`(get-bytevector-some!` *binary-input-port bytevector start n*`)` **procedure**
returns: the count of bytes read, as an exact nonnegative integer, or the eof object
libraries: `(chezscheme)`

start and *n* must be exact nonnegative integers, and the sum of *start* and *n* must not exceed the length of *bytevector*.

If *n* is 0, this procedure returns zero without attempting to read from *binary-input-port* and without modifying *bytevector*.

Otherwise, if *binary-input-port* is at end of file, this procedure returns the eof object, except it returns zero when the port is in nonblocking mode (see `set-port-nonblocking!`) and the port cannot be determined to be at end of file without blocking. In either case, *bytevector* is not modified.

Otherwise, this procedure reads (as if with `get-u8`) up to *n* bytes from the port, stores the bytes in consecutive locations of *bytevector* starting at *start*, advances the port's position just past the bytes read, and returns the count of bytes read.

If the port is in nonblocking mode, this procedure reads no more than it can without blocking and thus might read zero bytes; otherwise, it reads at least one byte but no more than are available when the first byte becomes available.

`(unread-char` *char*`)` **procedure**
`(unread-char` *char textual-input-port*`)` **procedure**
`(unget-char` *textual-input-port char*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

For `unread-char`, if *textual-input-port* is not supplied, it defaults to the current input port. These procedures "unread" the last character read from `textual-input-port`. *char* may or may not be ignored, depending upon the implementation. In any case, *char* should be last character read from the port. A character should not be unread twice on the same port without an intervening call to `read-char` or `get-char`.

`unread-char` and `unget-char` are provided for applications requiring one character of lookahead and may be used in place of, or even in combination with, `peek-char` or `lookahead-char`. One character of lookahead is required in the procedure `read-word`, which is defined below in terms of `unread-char`. `read-word` returns the next word from a textual input port as a string, where a word is defined to be a sequence of alphabetic characters. Since it does not know until it reads one character too many that it has read the entire word, `read-word` uses `unread-char` to return the character to the input port.

```
(define read-word
  (lambda (p)
    (list->string
      (let f ([c (read-char p)])
        (cond
          [(eof-object? c) '()]
          [(char-alphabetic? c)
           (cons c (f (read-char p)))]
          [else
           (unread-char c p)
           '()])))))
```

In the alternate version below, `peek-char` is used instead of `unread-char`.

```
(define read-word
  (lambda (p)
    (list->string
      (let f ([c (peek-char p)])
        (cond
          [(eof-object? c) '()]
          [(char-alphabetic? c)
           (read-char p)
           (cons c (f (peek-char p)))]
          [else '()])))))
```

The advantage of `unread-char` in this situation is that only one call to `unread-char` per word is required, whereas one call to `peek-char` is required for each character in the word plus the first character beyond. In many cases, `unread-char` and `unget-char` do not enjoy this advantage, and `peek-char` or `lookahead-char` should be used instead.

`(unget-u8` *binary-input-port octet*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedures "unreads" the last byte read from `binary-input-port`. *octet* may or may not be ignored, depending upon the implementation. In any case, *octet* should be last byte read from the port. A byte should not be unread twice on the same port without an intervening call to `get-u8`.

`(input-port-ready?` *input-port*`)` **procedure**
returns: `#t` if data is available on *input-port*, `#f` otherwise
libraries: `(chezscheme)`

`input-port-ready?` allows a program to check to see if input is available on a textual or binary input port without hanging. If input is available or the port is at end of file, `input-port-ready?` returns `#t`. If it cannot determine from the port whether input is ready, `input-port-ready?` raises an exception with condition type `&i/o-read-error`. Otherwise, it returns `#f`.

`(char-ready?)` **procedure**
`(char-ready?` *textual-input-port*`)` **procedure**
returns: `#t` if a character is available on *textual-input-port*, `#f` otherwise
libraries: `(chezscheme)`

If *textual-input-port* is not supplied, it defaults to the current input port. `char-ready?` is like `input-port-ready?` except it is restricted to textual input ports.

`(block-read` *textual-input-port string*`)` **procedure**
`(block-read` *textual-input-port string count*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

count must be a nonnegative fixnum less than or equal to the length of *string*. If not provided, it defaults to the length of *string*.

If *textual-input-port* is at end-of-file, an eof object is returned. Otherwise, *string* is filled with as many characters as are available for reading from *textual-input-port* up to *count*, and the number of characters placed in the string is returned.

If *textual-input-port* is buffered and the buffer is nonempty, the buffered input or a portion thereof is returned; otherwise `block-read` bypasses the buffer entirely.

`(read-token)` **procedure**
`(read-token` *textual-input-port*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

Parsing of a Scheme datum is conceptually performed in two steps. First, the sequence of characters that form the datum are grouped into `tokens`, such as symbols, numbers, left parentheses, and double quotes. During this first step, whitespace and comments are discarded. Second, these tokens are grouped into data.

`read` performs both of these steps and creates an internal representation of each datum it parses. `read-token` may be used to perform the first step only, one token at a time. `read-token` is intended to be used by editors and program formatters that must be able to parse a program or datum without actually reading it.

If *textual-input-port* is not supplied, it defaults to the current input port. One token is read from the input port and returned as four values:

type: a symbol describing the type of token read,

value: the token value,

start: the position of the first character of the token, relative to the starting position of the input port, and

end: the first position beyond the token, relative to the starting position of the input port.

When the token type fully specifies the token, `read-token` returns `#f` for the value. The token types are listed below with the corresponding *value* in parentheses.

`atomic` (*atom*) an atomic value, i.e., a symbol, boolean, number, character, `#!eof`, or `#!bwp`

`box` (`#f`) box prefix, i.e., `#&`

`dot` (`#f`) dotted pair separator, i.e., `.`

`eof` (`#!eof`) end of file

`fasl` (`#f`) fasl prefix, i.e., `#@`

`insert` (*n*) graph reference, i.e., `#`*n*`#`

`lbrack` (`#f`) open square bracket

`lparen` (`#f`) open parenthesis

`mark` (n) graph mark, i.e., `#`n`=`

`quote` (`quote`, `quasiquote`, `syntax`, `unquote`, `unquote-splicing`, or `datum-comment`) an abbreviation mark, e.g., `'` or `,@` or datum-comment prefix

`rbrack` (`#f`) close square bracket

`record-brack` (`#f`) record open bracket, i.e., `#[`

`rparen` (`#f`) close parenthesis

`vfxnparen` (n) fxvector prefix, i.e., `#`n`vfx(`

`vfxparen` (`#f`) fxvector prefix, i.e., `#vfx(`

`vnparen` (n) vector prefix, i.e., `#`n`(`

`vparen` (`#f`) vector prefix, i.e., `#(`

`vu8nparen` (n) bytevector prefix, i.e., `#`n`vu8(`

`vu8paren` (`#f`) bytevector prefix, i.e., `#vu8(`

The set of token types is likely to change in future releases of the system; check the release notes for details on such changes.

The input port is left pointing to the first character position beyond the token, i.e., *end* characters from the starting position.

```
(define s (open-input-string "(a b c)"))
(read-token s)  ⇒  lparen
                   #f
                   0
                   1
(define s (open-input-string "abc 123"))
(read-token s)  ⇒  atomic
                   abc
                   0
                   3
(define s (open-input-string ""))
(read-token s)  ⇒  eof
                   #!eof
                   0
                   0
(define s (open-input-string "#7=#7#"))
(read-token s)  ⇒  mark
                   7
                   0
                   3
(read-token s)  ⇒  insert
                   7
                   3
                   6
```

The information `read-token` returns is not always sufficient for reconstituting the exact sequence of characters that make up a token. For example, `1.0` and `1e0` both return *type* `atomic` with *value* `1.0`. The exact sequence of characters may be obtained only by repositioning the port and reading a block of characters of the appropriate length, using the relative positions given by *start* and *end*.

9.9. Output Operations

`console-output-port` **global parameter**
libraries: `(chezscheme)`

`console-output-port` is a parameter that determines the output port used by the waiter and interactive debugger. When called with no arguments, it returns the console output port. When called with one argument, which must be a textual output port, it changes the value of the console output port. The initial value of this parameter is a port tied to the standard output (stdout) stream of the Scheme process.

`current-output-port` **thread parameter**
libraries: `(chezscheme)`

`current-output-port` is a parameter that determines the default port argument for most output procedures, including `write-char`, `newline`, `write`, `display`, and `pretty-print`. When called with no arguments, `current-output-port` returns the current output port. When called with one argument, which must be a textual output port, it changes the value of the current output port. The Revised[6] Report version of `current-output-port` accepts only zero arguments, i.e., it cannot be used to change the current output port. The initial value of this parameter is the same port as the initial value of `console-output-port`.

`console-error-port` **thread parameter**
libraries: `(chezscheme)`

`console-error-port` is a parameter that can be used to set or obtain the console error port, which determines the port to which conditions and other messages are printed by the default exception handler. When called with no arguments, `console-error-port` returns the console error port. When called with one argument, which must be a textual output port, it changes the value of the console error port.

If the system determines that the standard output (stdout) and standard error (stderr) streams refer to the same file, socket, pipe, virtual terminal, device, etc., this parameter is initially set to the same value as the parameter `console-output-port`. Otherwise, this parameter is initially set to a different port tied to the standard error (stderr) stream of the Scheme process.

`current-error-port` **thread parameter**
libraries: `(chezscheme)`

`current-error-port` is a parameter that can be used to set or obtain the current error port. When called with no arguments, `current-error-port` returns the current error port. When called with one argument, which must be a textual output port, it changes the value of the current error port. The Revised[6] Report version of `current-error-port` accepts only zero arguments, i.e., it cannot be used to change the current error port. The initial value of this parameter is the same port as the initial value of `console-error-port`.

`(open-output-file` *path*`)` **procedure**
`(open-output-file` *path options*`)` **procedure**
returns: a new output port
libraries: `(chezscheme)`

path must be a string. `open-output-file` opens a textual output port for the file named by *path*.

options, if present, is a symbolic option name or option list. Possible symbolic option names are `error`, `truncate`, `replace`, `append`, `compressed`, `uncompressed`, `buffered`, `unbuffered`, `exclusive`, and `nonexclusive`. An option list is a list containing zero or more symbolic option names and possibly the two-element option `mode` *mode*.

The mutually exclusive `error`, `truncate`, `replace`, and `append` options are used to direct what happens when the file to be opened already exists.

`error`: An exception is raised with condition-type `&i/o-filename`.

`replace`: The existing file is deleted before the new file is opened.

`truncate`: The existing file is opened and truncated to zero length.

`append`: The existing file is opened and the output port is positioned at the end of the file before each write so that output to the port is always appended to the file.

The default behavior is to raise an exception.

The mutually exclusive `compressed` and `uncompressed` options determine whether the output file is to be compressed. Compression is performed with the use of the zlib compression library developed by Jean-loup Gailly and Mark Adler. It is therefore compatible with the `gzip` program, which means that `gzip` may be used to uncompress files produced by *Chez Scheme* and visa versa. Files are uncompressed by default, so the `uncompressed` option is useful only as documentation.

The mutually exclusive `buffered` and `unbuffered` options determine whether output is buffered. Unbuffered output is sent immediately to the file, whereas buffered output not written until the port's output buffer is filled or the port is flushed (via `flush-output-port`) or closed (via `flush-output-port` or by the storage management system when the port becomes inaccessible). Output is buffered by default for efficiency, so the `buffered` option is useful only as documentation.

The mutually exclusive `exclusive` and `nonexclusive` options determine whether access to the file is "exclusive." When the exclusive option is specified, the file is locked until the port

is closed to prevent access by other processes. On some systems the lock is advisory, i.e., it inhibits access by other processes only if they also attempt to open exclusively. Nonexclusive access is the default, so the `nonexclusive` option is useful only as documentation.

The `mode` option determines the permission bits on Unix systems when the file is created by the operation, subject to the process umask. The subsequent element in the options list must be an exact integer specifying the permissions in the manner of the Unix `open` function. The mode option is ignored under Windows.

For example, the call

```
(open-output-file "frob" '(compressed truncate mode #o644))
```

opens the file frob with compression enabled. If frob already exists it is truncated. On Unix-based systems, if frob does not already exist, the permission bits on the newly created file are set to logical and of `#o644` and the process's umask.

The Revised[6] Report version of `open-output-file` does not support the optional *options* argument.

`(call-with-output-file` *path procedure*`)` **procedure**
`(call-with-output-file` *path procedure options*`)` **procedure**
returns: the values returned by *procedure*
libraries: `(chezscheme)`

path must be a string. *procedure* should accept one argument.

`call-with-output-file` creates a new output port for the file named by *path*, as if with `open-output-file`, and passes this port to *procedure*. If *procedure* returns, `call-with-output-file` closes the output port and returns the values returned by *procedure*.

`call-with-output-file` does not automatically close the output port if a continuation created outside of *procedure* is invoked, since it is possible that another continuation created inside of *procedure* will be invoked at a later time, returning control to *procedure*. If *procedure* does not return, an implementation is free to close the output port only if it can prove that the output port is no longer accessible. As shown in Section 5.6 of *The Scheme Programming Language, 4th Edition*, `dynamic-wind` may be used to ensure that the port is closed if a continuation created outside of *procedure* is invoked.

See `open-output-file` above for a description of the optional *options* argument.

The Revised[6] Report version of `call-with-output-file` does not support the optional *options* argument.

`(with-output-to-file` *path thunk*`)` **procedure**
`(with-output-to-file` *path thunk options*`)` **procedure**
returns: the value returned by *thunk*
libraries: `(chezscheme)`

path must be a string. *thunk* must be a procedure and should accept zero arguments.

`with-output-to-file` temporarily rebinds the current output port to be the result of opening the file named by *path*, as if with `open-output-file`, during the application of *thunk*. If *thunk* returns, the port is closed and the current output port is restored to its old value.

The behavior of `with-output-to-file` is unspecified if a continuation created outside of *thunk* is invoked before *thunk* returns. An implementation may close the port and restore the current output port to its old value—but it may not.

See `open-output-file` above for a description of the optional *options* argument.

The Revised[6] Report version of `with-output-to-file` does not support the optional *options* argument.

`(open-fd-output-port` *fd*`)` **procedure**
`(open-fd-output-port` *fd b-mode*`)` **procedure**
`(open-fd-output-port` *fd b-mode ?transcoder*`)` **procedure**
returns: a new output port for the file descriptor *fd*
libraries: `(chezscheme)`

fd must be a nonnegative exact integer and should be a valid open file descriptor. If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure returns a textual output port whose transcoder is *?transcoder*. Otherwise, this procedure returns a binary output port. See the lead-in to Section 7.2 of *The Scheme Programming Language, 4th Edition* for a description of the constraints on and effects of the other arguments.

The file descriptor is closed when the port is closed.

`(standard-output-port)` **procedure**
`(standard-output-port` *b-mode*`)` **procedure**
`(standard-output-port` *b-mode ?transcoder*`)` **procedure**
returns: a new output port connected to the process's standard output
libraries: `(chezscheme)`

If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure returns a textual output port whose transcoder is *?transcoder*. Otherwise, this procedure returns a binary output port. The buffer mode *b-mode* defaults to `line`, which differs from `block` in *Chez Scheme* only for textual output ports. See the lead-in to Section 7.2 of *The Scheme Programming Language, 4th Edition* for a description of the constraints on and effects of the other arguments.

The Revised[6] Report version of this procedure does not accept the optional *b-mode* and *?transcoder* arguments, which limits it to an implementation-dependent buffering mode (`line` in *Chez Scheme*) and binary output.

`(standard-error-port)` **procedure**
`(standard-error-port` *b-mode*`)` **procedure**
`(standard-error-port` *b-mode ?transcoder*`)` **procedure**
returns: a new output port connected to the process's standard error
libraries: `(chezscheme)`

If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure returns a textual output port whose transcoder is *?transcoder*. Otherwise, this procedure returns a binary output port. The buffer mode *b-mode* defaults to `none`. See the lead-in to Section 7.2 of *The Scheme Programming Language, 4th Edition* for a description of the constraints on and effects of the other arguments.

The Revised[6] Report version of this procedure does not accept the optional *b-mode* and *?transcoder* arguments, which limits it to an implementation-dependent buffering mode (`none` in *Chez Scheme*) and binary output.

`(put-bytevector-some` *binary-output-port bytevector*`)` **procedure**
`(put-bytevector-some` *binary-output-port bytevector start*`)` **procedure**
`(put-bytevector-some` *binary-output-port bytevector start n*`)` **procedure**
returns: the number of bytes written
libraries: `(chezscheme)`

start and *n* must be nonnegative exact integers, and the sum of *start* and *n* must not exceed the length of *bytevector*. If not supplied, *start* defaults to zero and *n* defaults to the difference between the length of *bytevector* and *start*.

This procedure normally writes the *n* bytes of *bytevector* starting at *start* to the port and advances the its position past the end of the bytes written. If the port is in nonblocking mode (see `set-port-nonblocking!`), however, the number of bytes written may be less than *n*, if the system would have to block to write more bytes.

`(put-string-some` *textual-output-port string*`)` **procedure**
`(put-string-some` *textual-output-port string start*`)` **procedure**
`(put-string-some` *textual-output-port string start n*`)` **procedure**
returns: the number of characters written
libraries: `(chezscheme)`

start and *n* must be nonnegative exact integers, and the sum of *start* and *n* must not exceed the length of *string*. If not supplied, *start* defaults to zero and *n* defaults to the difference between the length of *string* and *start*.

This procedure normally writes the *n* characters of *string* starting at *start* to the port and advances the its position past the end of the characters written. If the port is in nonblocking mode (see `set-port-nonblocking!`), however, the number of characters written may be less than *n*, if the system would have to block to write more characters.

`(display-string` *string*`)` **procedure**
`(display-string` *string textual-output-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`display-string` writes the characters contained within *string* to *textual-output-port* or to the current-output port if `textual-output-port` is not specified. The enclosing string quotes are not printed, and special characters within the string are not escaped. `display-string` is a more efficient alternative to `display` for displaying strings.

`(block-write` *textual-output-port string*`)` **procedure**
`(block-write` *textual-output-port string count*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

count must be a nonnegative fixnum less than or equal to the length of *string*. If not provided, it defaults to the length of *string*.

`block-write` writes the first *count* characters of *string* to *textual-output-port*. If the port is buffered and the buffer is nonempty, the buffer is flushed before the contents of *string* are written. In any case, the contents of *string* are written immediately, without passing through the buffer.

`(truncate-port` *output-port*`)` **procedure**
`(truncate-port` *output-port pos*`)` **procedure**
`(truncate-file` *output-port*`)` **procedure**
`(truncate-file` *output-port pos*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`truncate-port` and `truncate-file` are identical.

pos must be an exact nonnegative integer. It defaults to 0.

These procedures truncate the file or other object associated with *output-port* to *pos* and repositions the port to that position, i.e., it combines the functionality of `set-port-length!` and `set-port-position!` and can be called on a port only if `port-has-set-port-length!?` and `port-has-set-port-position!?` are both true of the port.

`(fresh-line)` **procedure**
`(fresh-line` *textual-output-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

If *textual-output-port* is not supplied, it defaults to the current output port.

This procedure behaves like `newline`, i.e., sends a newline character to *textual-output-port*, unless it can determine that the port is already positioned at the start of a line. It does

this by flushing the port and consulting the "beginning-of-line" (BOL) flag associated with the port. (See page 220.)

9.10. Input/Output Operations

`(open-input-output-file` *path*`)` **procedure**
`(open-input-output-file` *path options*`)` **procedure**
returns: a new input-output port
libraries: `(chezscheme)`

path must be a string. `open-input-output-file` opens a textual input-output port for the file named by *path*.

The port may be used to read from or write to the named file. The file is created if it does not already exist.

options, if present, is a symbolic option name or option list. Possible symbolic option names are `buffered`, `unbuffered`, `exclusive`, and `nonexclusive`. An option list is a list containing zero or more symbolic option names and possibly the two-element option `mode` *mode*. See the description of `open-output-file` for an explanation of these options.

Input/output files are usually closed using `close-port` but may also be closed with either `close-input-port` or `close-output-port`.

`(open-fd-input/output-port` *fd*`)` **procedure**
`(open-fd-input/output-port` *fd b-mode*`)` **procedure**
`(open-fd-input/output-port` *fd b-mode ?transcoder*`)` **procedure**
returns: a new input/output port for the file descriptor *fd*
libraries: `(chezscheme)`

fd must be a nonnegative exact integer and should be a valid open file descriptor. If *?transcoder* is present and not `#f`, it must be a transcoder, and this procedure returns a textual input/output port whose transcoder is *?transcoder*. Otherwise, this procedure returns a binary input/output port. See the lead-in to Section 7.2 of *The Scheme Programming Language, 4th Edition* for a description of the constraints on and effects of the other arguments.

The file descriptor is closed when the port is closed.

9.11. Non-Unicode Bytevector/String Conversions

The procedures described in this section convert bytevectors containing single- and multiple-byte sequences in non-Unicode character sets to and from Scheme strings. They are available only under Windows. Under other operating systems, and when an `iconv` DLL is available under Windows, `bytevector->string` and `string->bytevector` can be used with a transcoder based on a codec constructed via `iconv-codec` to achieve the same results, with more control over the handling of invalid characters and line endings.

(`multibyte->string` *code-page bytevector*) **procedure**
returns: a string containing the characters encoded in *bytevector*
(`string->multibyte` *code-page string*) **procedure**
returns: a bytevector containing the encodings of the characters in *string*
libraries: (`chezscheme`)

These procedures are available only under Windows. The procedure `multibyte->string` is a wrapper for the Windows API `MultiByteToWideChar` function, and `string->multibyte` is a wrapper for the Windows API `WideCharToMultiByte` function.

code-page declares the encoding of the byte sequences in the input or output bytevectors. It must be an exact nonnegative integer identifying a code page or one of the symbols `cp-acp`, `cp-maccp`, `cp-oemcp`, `cp-symbol`, `cp-thread-acp`, `cp-utf7`, or `cp-utf8`, which have the same meanings as the API function meanings for the like-named constants.

9.12. Pretty Printing

The pretty printer is a version of the `write` procedure that produces more human-readable output via introduced whitespace, i.e., line breaks and indentation. The pretty printer is the default printer used by the read-eval-print loop (waiter) to print the output(s) of each evaluated form. The pretty printer may also be invoked explicitly by calling the procedure `pretty-print`.

The pretty printer's operation can be controlled via the `pretty-format` procedure described later in this section, which allows the programmer to specify how specific forms are to be printed, various pretty-printer controls, also described later in this section, and also by the generic input/output controls described in Section 9.14.

(`pretty-print` *obj*) **procedure**
(`pretty-print` *obj textual-output-port*) **procedure**
returns: unspecified
libraries: (`chezscheme`)

If *textual-output-port* is not supplied, it defaults to the current output port.

`pretty-print` is similar to `write` except that it uses any number of spaces and newlines in order to print *obj* in a style that is pleasing to look at and which shows the nesting level via indentation. For example,

```
(pretty-print '(define factorial (lambda (n) (let fact ((i n) (a 1))
  (if (= i 0) a (fact (- i 1) (* a i)))))))
```

might produce

```
(define factorial
  (lambda (n)
    (let fact ([i n] [a 1])
      (if (= i 0) a (fact (- i 1) (* a i))))))
```

`(pretty-file` *ifn ofn*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

ifn and *ofn* must be strings. `pretty-file` reads each object in turn from the file named by *ifn* and pretty prints the object to the file named by *ofn*. Comments present in the input are discarded by the reader and so do not appear in the output file. If the file named by *ofn* already exists, it is replaced.

`(pretty-format` *sym*`)` **procedure**
returns: see below
`(pretty-format` *sym fmt*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

By default, the pretty printer uses a generic algorithm for printing each form. This procedure is used to override this default and guide the pretty-printers treatment of specific forms. The symbol *sym* names a syntactic form or procedure. With just one argument, `pretty-format` returns the current format associated with *sym*, or `#f` if no format is associated with *sym*.

In the two-argument case, the format *fmt* is associated with *sym* for future invocations of the pretty printer. *fmt* must be in the formatting language described below.

⟨fmt⟩ ⟶ `(quote` *symbol*`)`
| `var`
| *symbol*
| `(read-macro` *string symbol*`)`
| `(meta)`
| `(bracket .` *fmt-tail*`)`
| `(alt` *fmt fmt*`*)`
| *fmt-tail*

fmt-tail ⟶ `()`
| `(`*tab fmt* `...)`
| `(`*fmt tab* `...)`
| `(`*tab fmt* `.` *fmt-tail*`)`
| `(`*fmt* `...)`
| `(`*fmt* `.` *fmt-tail*`)`
| `(fill` *tab fmt* `...)`

tab ⟶ *int*
| `#f`

Some of the format forms are used for matching when there are multiple alternatives, while others are used for matching and control indentation or printing. A description of each *fmt* is given below.

`(quote` *symbol*`)`: This matches only the symbol *symbol*.

`var`: This matches any symbol.

symbol: This matches any input.

`(read-macro` *string symbol*`)`: This is used for read macros like `quote` and *syntax*. It matches any input of the form (*symbol subform*). For forms that match, the pretty printer prints *string* immediately followed by *subform*.

`(meta)`: This is a special case used for the `meta` keyword (Section 11.8) which is used as a keyword prefix of another form.

`(alt` *fmt fmt*`*)`: This compares the input against the specified formats and uses the one that is the closest match. Most often, one of the formats will match exactly, but in other cases, as when input is malformed or appears in abstract form in the template of a syntactic abstraction, none of the formats will match exactly.

`(bracket .` *fmt-tail*`)`: This matches any list-structured input and prints the input enclosed in square brackets, i.e., `[` and `]`, rather than parentheses.

fmt-tail: This matches any list-structured input.

Indentation of list-structured forms is determined via the *fmt-tail* specifier used to the last two cases above. A description of each *fmt-tail* is given below.

`()`: This matches an empty list tail.

(*tab fmt* `...`): This matches the tail of any proper list; if the tail is nonempty and the list does not fit entirely on the current line, a line break is inserted before the first subform of the tail and *tab* (see below) determines the amount by which this and all subsequent subforms are indented.

(*fmt tab* `...`): This matches the tail of any proper list; if the tail is nonempty and the list does not fit entirely on the current line, a line break is inserted after the first subform of the tail and *tab* (see below) determines the amount by which all subsequent subforms are indented.

(*tab fmt . fmt-tail*): This matches a nonempty tail if the tail of the tail matches *fmt-tail*. If the list does not fit entirely on the current line, a line break is inserted before the first subform of the tail and *tab* (see below) determines the amount by which the subform is indented.

(*fmt* `...`): This matches the tail of any proper list and specified that no line breaks are to be inserted before or after the current or subsequent subforms.

(*fmt . fmt-tail*): This matches a nonempty tail if the tail of the tail matches *fmt-tail* and specifies that no line break is to be inserted before or after the current subform.

`(fill` *tab fmt* `...`): This matches the tail of any proper list and invokes a fill mode in which the forms are packed with as many as will fit on each line.

A *tab* determines the amount by which a list subform is indented. If *tab* is a nonnegative exact integer *int*, the subform is indented *int* spaces in from the character position just after the opening parenthesis or bracket of the parent form. If *tab* is `#f`, the standard indentation is used. The standard indentation can be determined or changed via the parameter `pretty-standard-indent`, which is described later in this section.

In cases where a format is given that doesn't quite match, the pretty printer tries to use the given format as far as it can. For example, if a format matches a list-structured form with a specific number of subforms, but more or fewer subform are given, the pretty printer will discard or replicate subform formats as necessary.

Here is an example showing the formatting of *let* might be specified.

```
(pretty-format 'let
  '(alt (let ([bracket var x] 0 ...) #f e #f e ...)
        (let var ([bracket var x] 0 ...) #f e #f e ...)))
```

Since `let` comes in two forms, named and unnamed, two alternatives are specified. In either case, the `bracket` *fmt* is used to enclose the bindings in square brackets, with all bindings after the first appearing just below the first (and just after the enclosing opening parenthesis), if they don't all fit on one line. Each body form is indented by the standard indentation.

`pretty-line-length` **thread parameter**
`pretty-one-line-limit` **thread parameter**
libraries: `(chezscheme)`

The value of each of these parameters must be a positive fixnum.

The parameters `pretty-line-length` and `pretty-one-line-limit` control the output produced by `pretty-print`. `pretty-line-length` determines after which character position (starting from the first) on a line the pretty printer attempts to cut off output. This is a soft limit only; if necessary, the pretty-printer will go beyond `pretty-line-length`.

`pretty-one-line-limit` is similar to `pretty-line-length`, except that it is relative to the first nonblank position on each line of output. It is also a soft limit.

`pretty-initial-indent` **thread parameter**
libraries: `(chezscheme)`

The value of this parameter must be a nonnegative fixnum.

The parameter `pretty-initial-indent` is used to tell `pretty-print` where on an output line it has been called. If `pretty-initial-indent` is zero (the default), `pretty-print` assumes that the first line of output it produces will start at the beginning of the line. If set to a nonzero value n, `pretty-print` assumes that the first line will appear at character position n and will adjust its printing of subsequent lines.

`pretty-standard-indent` **thread parameter**
libraries: `(chezscheme)`

The value of this parameter must be a nonnegative fixnum.

This determines the amount by which `pretty-print` indents subexpressions of most forms, such as `let` expressions, from the form's keyword or first subexpression.

`pretty-maximum-lines` **thread parameter**
libraries: `(chezscheme)`

The parameter `pretty-maximum-lines` controls how many lines `pretty-print` prints when it is called. If set to `#f` (the default), no limit is imposed; if set to a nonnegative fixnum n, at most n lines are printed.

9.13. Formatted Output

`(format` *format-string obj ...*`)` **procedure**
`(format #f` *format-string obj ...*`)` **procedure**
`(format #t` *format-string obj ...*`)` **procedure**
`(format` *textual-output-port format-string obj ...*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

When the first argument to format is a string or `#f` (first and second forms above), `format` constructs an output string from *format-string* and the objects *obj* .... Characters are copied from *format-string* to the output string from left to right, until *format-string* is exhausted. The format string may contain one or more *format directives*, which are multi-character sequences prefixed by a tilde (`~`). Each directive is replaced by some other text, often involving one or more of the *obj* ... arguments, as determined by the semantics of the directive.

When the first argument is `#t`, output is sent to the current output port instead, as with `printf`. When the first argument is a port, output is sent to that port, as with `fprintf`. `printf` and `fprintf` are described later in this section.

Chez Scheme's implementation of `format` supports all of the Common Lisp [30] format directives except for those specific to the Common Lisp pretty printer. Please consult a Common Lisp reference or the Common Lisp Hyperspec, for complete documentation. A few of the most useful directives are described below.

Absent any format directives, `format` simply displays its string argument.

```
(format "hi there")  ⇒  "hi there"
```

The `~s` directive is replaced by the printed representation of the next *obj*, which may be any object, in machine-readable format, as with `write`.

```
(format "hi ~s" 'mom)  ⇒  "hi mom"
(format "hi ~s" "mom")  ⇒  "hi \"mom\""
(format "hi ~s~s" 'mom #\!)  ⇒  "hi mom#\\!"
```

The general form of a `~s` directive is actually `~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`s`, and the `s` can be preceded by an at sign (`@`) modifier. These additional parameters are used to control padding in the output, with at least *minpad* copies of *padchar* plus an integer multiple of *colinc* copies of *padchar* to make the total width, including the written object, *mincol* characters wide. The padding is placed on the left if the @ modifier is present,

otherwise on the right. *mincol* and *minpad* default to 0, *colinc* defaults to 1, and *padchar* defaults to space. If specified, *padchar* is prefixed by a single quote mark.

```
(format "~10s" 'hello)  ⇒  "hello     "
(format "~10@s" 'hello)  ⇒  "     hello"
(format "~10,,,'*@s" 'hello)  ⇒  "*****hello"
```

The `~a` directive is similar, but prints the object as with `display`.

```
(format "hi ~s~s" "mom" #\!)  ⇒  "hi \"mom\"#\\!"
(format "hi ~a~a" "mom" #\!)  ⇒  "hi mom!"
```

A tilde may be inserted into the output with `~~`, and a newline may be inserted with `~%` (or embedded in the string with `\n`).

```
(format "~~line one,~%line two.~~")  ⇒  "~line one,\nline two.~"
(format "~~line one,\nline two.~~")  ⇒  "~line one,\nline two.~"
```

Real numbers may be printed in floating-point notation with `~f`.

```
(format "~f" 3.14159)  ⇒  3.14159
```

Exact numbers may printed as well as inexact numbers in this manner; they are simply converted to inexact first as if with `exact->inexact`.

```
(format "~f" 1/3)  ⇒  "0.3333333333333333"
```

The general form is actually `~`*w*`,`*d*`,`*k*`,`*overflowchar*`,`*padchar*`f`. If specified, *w* determines the overall width of the output, and *d* the number of digits to the right of the decimal point. *padchar*, which defaults to space, is the pad character used if padding is needed. Padding is always inserted on the left. The number is scaled by 10^k when printed; *k* defaults to zero. The entire *w*-character field is filled with copies of *overflowchar* if *overflowchar* is specified and the number cannot be printed in *w* characters. *k* defaults to 1 If an `@` modifier is present, a plus sign is printed before the number for nonnegative inputs; otherwise, a sign is printed only if the number is negative.

```
(format "~,3f" 3.14159)  ⇒  "3.142"
(format "~10f" 3.14159)  ⇒  "   3.14159"
(format "~10,,,'#f" 1e20)  ⇒  "##########"
```

Real numbers may also be printed with `~e` for scientific notation or with `~g`, which uses either floating-point or scientific notation based on the size of the input.

```
(format "~e" 1e23)  ⇒  "1.0e+23"
(format "~g" 1e23)  ⇒  "1.0e+23"
```

A real number may also be printed with `~$`, which uses monetary notation defaulting to two digits to the right of the decimal point.

```
(format "$~$" (* 39.95 1.06))  ⇒  "$42.35"
(format "~$USD" 1/3)  ⇒  "0.33USD"
```

Words can be pluralized automatically using `p`.

```
(format "~s bear~:p in ~s den~:p" 10 1)  ⇒  "10 bears in 1 den"
```

Numbers may be printed out in words or roman numerals using variations on `~r`.

```
(format "~r" 2599)  ⇒   "two thousand five hundred ninety-nine"
(format "~:r" 99)  ⇒   "ninety-ninth"
(format "~@r" 2599)  ⇒  "MMDXCIX"
```

Case conversions can be performed by bracketing a portion of the format string with the `~@(` and `~)` directives.

```
(format "~@(~r~)" 2599)  ⇒   "Two thousand five hundred ninety-nine"
(format "~@:(~a~)" "Ouch!")  ⇒   "OUCH!"
```

Some of the directives shown above have more options and parameters, and there are other directives as well, including directives for conditionals, iteration, indirection, and justification. Again, please consult a Common Lisp reference for complete documentation.

An implementation of a greatly simplified version of `format` appears in Section 12.6 of *The Scheme Programming Language, 4th Edition*.

`(printf` *format-string obj ...*`)` **procedure**
`(fprintf` *textual-output-port format-string obj ...*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

These procedures are simple wrappers for `format`. `printf` prints the formatted output to the current output, as with a first-argument of `#t` to `format`, and `fprintf` prints the formatted output to the *textual-output-port*, as when the first argument to `format` is a port.

9.14. Input/Output Control Operations

The I/O control operations described in this section are used to control how the reader reads and printer writes, displays, or pretty-prints characters, symbols, gensyms, numbers, vectors, long or deeply nested lists or vectors, and graph-structured objects.

`(char-name` *obj*`)` **procedure**
returns: see below
`(char-name` *name char*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`char-name` is used to associate names (symbols) with characters or to retrieve the most recently associated name or character for a given character or name. A name can map to only one character, but more than one name can map to the same character. The name

most recently associated with a character determines how that character prints, and each name associated with a character may be used after the `#\` character prefix to name that character on input.

Character associations created by `char-name` are ignored by the printer unless the parameter `print-char-name` is set to a true value. The reader recognizes character names established by `char-name` except after `#!r6rs`, which is implied within a library or R6RS top-level program.

In the one-argument form, *obj* must be a symbol or character. If it is a symbol and a character is associated with the symbol, `char-name` returns that character. If it is a symbol and no character is associated with the symbol, `char-name` returns `#f`. Similarly, if *obj* is a character, `char-name` returns the most recently associated symbol for the character or `#f` if no name is associated with the character. For example, with the default set of character names:

```
(char-name #\space)  ⇒  space
(char-name 'space)  ⇒  #\space
(char-name 'nochar)  ⇒  #f
(char-name #\a)  ⇒  #f
```

When passed two arguments, *name* is added to the set of names associated with *char*, and any other association for *name* is dropped. *char* may be `#f`, in which case any other association for *name* is dropped and no new association is formed. In either case, any other names associated with *char* remain associated with *char*.

The following interactive session demonstrates the use of `char-name` to establish and remove associations between characters and names, including the association of more than one name with a character.

```
(print-char-name #t)
(char-name 'etx)  ⇒  #f
(char-name 'etx #\x3)
(char-name 'etx)  ⇒  #\etx
(char-name #\x3)  ⇒  etx
#\etx  ⇒  #\etx
(eq? #\etx #\x3)  ⇒  #t
#!r6rs #\etx  ⇒  exception: invalid character name etx
#!chezscheme #\etx  ⇒  #\etx
(char-name 'etx #\space)
(char-name #\x3)  ⇒  #f
(char-name 'etx)  ⇒  #\etx
#\space  ⇒  #\etx
(char-name 'etx #f)
#\etx  ⇒  exception: invalid character name etx
#\space  ⇒  #\space
```

(When using the expression editor, it is necessary to type Control-J to force the editor to read the erroneous `#\etx` input on the two inputs above that result in read errors, since typing Enter causes the expression editor to read the input only if the input is well-formed.)

The reader does not recognize hex scalar value escapes in character names, as it does in symbols, so `#\new\x6c;ine` is not equivalent to `#\newline`. In general, programmers should avoid the use of character name symbols that cannot be entered without the use of hex scalar value escapes or other symbol-name escape mechanisms, since such character names will not be readable.

`print-char-name` **thread parameter**
libraries: `(chezscheme)`

When `print-char-name` is set to `#f` (the default), associations created by `char-name` are ignored by `write`, `put-datum`, `pretty-print`, and the `format` "`~s`" directive. Otherwise, these procedures use the names established by `char-name` when printing character objects.

```
(char-name 'etx #\x3)
(format "~s" #\x3)  ⇒  "#\\x3"
(parameterize ([print-char-name #t])
  (format "~s" #\x3))  ⇒  "#\\etx"
```

`case-sensitive` **thread parameter**
libraries: `(chezscheme)`

The `case-sensitive` parameter determines whether the reader is case-sensitive with respect to symbol and character names. When set to true (the default, as required by the Revised[6] Report) the case of alphabetic characters within symbol names is significant. When set to `#f`, case is insignificant. More precisely, when set to `#f`, symbol and character names are folded (as if by `string-foldcase`); otherwise, they are left as they appear in the input.

The value of the `case-sensitive` matters only if neither `#!fold-case` nor `#!no-fold-case` has appeared previously in the same input stream. That is, symbol and character name are folded if `#!fold-case` has been seen. They are not folded if `#!no-fold-case` has been seen. If neither has been seen, they are folded if and only if `(case-sensitive)` is `#f`.

```
(case-sensitive)  ⇒  #t
(eq? 'abc 'ABC)  ⇒  #f
'ABC  ⇒  ABC
(case-sensitive #f)
'ABC  ⇒  abc
(eq? 'abc 'ABC)  ⇒  #t
```

`print-graph` **thread parameter**
libraries: `(chezscheme)`

When `print-graph` is set to a true value, `write` and `pretty-print` locate and print objects with shared structure, including cycles, in a notation that may be read subsequently with `read`. This notation employs the syntax "`#`*n*`=`*obj*," where *n* is a nonnegative integer and *obj* is the printed representation of an object, to label the first occurrence of *obj* in the

output. The syntax "`#`*n*`#`" is used to refer to the object labeled by *n* thereafter in the output. `print-graph` is set to `#f` by default.

If graph printing is not enabled, the settings of `print-length` and `print-level` are insufficient to force finite output, and `write` or `pretty-print` detects a cycle in an object it is given to print, a warning is issued (an exception with condition type `&warning` is raised) and the object is printed as if `print-graph` were enabled.

Since objects printed through the `~s` option in the format control strings of `format`, `printf`, and `fprintf` are printed as with `write`, the printing of such objects is also affected by `print-graph`.

```
(parameterize ([print-graph #t])
  (let ([x (list 'a 'b)])
    (format "~s" (list x x))))  ⇒  "(#0=(a b) #0#)"

(parameterize ([print-graph #t])
  (let ([x (list 'a 'b)])
    (set-car! x x)
    (set-cdr! x x)
    (format "~s" x)))  ⇒  "#0=(#0# . #0#)"
```

The graph syntax is understood by the procedure `read`, allowing graph structures to be printed and read consistently.

`print-level` **thread parameter**
`print-length` **thread parameter**
libraries: `(chezscheme)`

These parameters can be used to limit the extent to which nested or multiple-element structures are printed. When called without arguments, `print-level` returns the current print level and `print-length` returns the current print length. When called with one argument, which must be a nonnegative fixnum or `#f`, `print-level` sets the current print level and `print-length` sets the current print length to the argument.

When `print-level` is set to a nonnegative integer *n*, `write` and `pretty-print` traverse only *n* levels deep into nested structures. If a structure being printed exceeds *n* levels of nesting, the substructure beyond that point is replaced in the output by an ellipsis (`...`). `print-level` is set to `#f` by default, which places no limit on the number of levels printed.

When `print-length` is set to a nonnegative integer *n*, the procedures `write` and `pretty-print` print only *n* elements of a list or vector, replacing the remainder of the list or vector with an ellipsis (`...`). `print-length` is set to `#f` by default, which places no limit on the number of elements printed.

Since objects printed through the `~s` option in the format control strings of `format`, `printf`, and `fprintf` are printed as with `write`, the printing of such objects is also affected by `print-level` and `print-length`.

The parameters `print-level` and `print-length` are useful for controlling the volume of output in contexts where only a small portion of the output is needed to identify the

object being printed. They are also useful in situations where circular structures may be printed (see also `print-graph`).

```
(format "~s" '((((a) b) c) d e f g))  ⇒  "((((a) b) c) d e f g)"

(parameterize ([print-level 2])
  (format "~s" '((((a) b) c) d e f g)))  ⇒  "(((...) c) d e f g)"

(parameterize ([print-length 3])
  (format "~s" '((((a) b) c) d e f g)))  ⇒  "((((a) b) c) d e ...)"

(parameterize ([print-level 2]
               [print-length 3])
  (format "~s" '((((a) b) c) d e f g)))  ⇒  "(((...) c) d e ...)"
```

`print-radix` **thread parameter**
libraries: `(chezscheme)`

The `print-radix` parameter determines the radix in which numbers are printed by `write`, `pretty-print`, and `display`. Its value should be an integer between 2 and 36, inclusive. Its default value is 10.

When the value of `print-radix` is not 10, `write` and `pretty-print` print a radix prefix before the number (`#b` for radix 2, `#o` for radix 8, `#x` for radix 16, and `#`*n*`r` for any other radix *n*).

Since objects printed through the `~s` and `~a` options in the format control strings of `format`, `printf`, and `fprintf` are printed as with `write` and `display`, the printing of such objects is also affected by `print-radix`.

```
(format "~s" 11242957)  ⇒  "11242957"

(parameterize ([print-radix 16])
  (format "~s" 11242957))  ⇒  "#xAB8DCD"

(parameterize ([print-radix 16])
  (format "~a" 11242957))  ⇒  "AB8DCD"
```

`print-gensym` **thread parameter**
libraries: `(chezscheme)`

When `print-gensym` is set to `#t` (the default) gensyms are printed with an extended symbol syntax that includes both the pretty name and the unique name of the gensym: `#{`*pretty-name unique-name*`}`. When set to `pretty`, the pretty name only is shown, with the prefix `#:`. When set to `pretty/suffix`, the printer prints the gensym's "pretty" name along with a suffix based on the gensym's "unique" name, separated by a dot ("."). If the gensym's unique name is generated automatically during the current session, the suffix is that portion of the unique name that is not common to all gensyms created during the current session. Otherwise, the suffix is the entire unique name. When set to `#f`, the pretty name only is shown, with no prefix.

Since objects printed through the `~s` option in the format control strings of `format`, `printf`, `errorf`, etc., are printed as with `write`, the printing of such objects is also affected by `print-gensym`.

When printing an object that may contain more than one occurrence of a gensym and `print-graph` is set to `pretty` or `#f`, it is useful to set `print-graph` to `#t` so that multiple occurrences of the same gensym are marked as identical in the output.

```
(let ([g (gensym)])
  (format "~s" g))  ⇒  "#{g0 bdids2xl6v49vgwe-a}"

(let ([g (gensym)])
  (parameterize ([print-gensym 'pretty])
    (format "~s" g)))  ⇒  "#:g1

(let ([g (gensym)])
  (parameterize ([print-gensym #f])
    (format "~s" g)))  ⇒  "g2"

(let ([g (gensym)])
  (parameterize ([print-graph #t] [print-gensym 'pretty])
    (format "~s" (list g g))))  ⇒  "(#0=#:g3 #0#)"

(let ([g1 (gensym "x")]
      [g2 (gensym "x")]
      [g3 (gensym "y")])
  (parameterize ([print-gensym 'pretty/suffix])
    (format "~s ~s ~s" g1 g2 g3)))  ⇒  "x.1 x.2 y.3"
```

`print-brackets` **thread parameter**

libraries: `(chezscheme)`

When `print-brackets` is set to a true value, the pretty printer (see `pretty-print`) uses square brackets rather than parentheses around certain subexpressions of common control structures, e.g., around `let` bindings and `cond` clauses. `print-brackets` is set to `#t` by default.

```
(let ([p (open-output-string)])
  (pretty-print '(let ([x 3]) x) p)  ⇒  "(let ([x 3]) x)
  (get-output-string p))                "

(parameterize ([print-brackets #f])
  (let ([p (open-output-string)])
    (pretty-print '(let ([x 3]) x) p)  ⇒  "(let ((x 3)) x)
    (get-output-string p)))               "
```

`print-extended-identifiers` **thread parameter**

libraries: `(chezscheme)`

Chez Scheme extends the syntax of identifiers as described in Section 1.1, except within a

set of forms prefixed by `#!r6rs` (which is implied by in a library or top-level program).

When this parameter is set to false (the default), identifiers in the extended set are printed with hex scalar value escapes as necessary to conform to the R6RS syntax for identifiers. When this parameter is set to a true value, identifiers in the extended set are printed without the escapes. Identifiers whose names fall outside of both syntaxes are printed with the escapes regardless of the setting of this parameter.

For example:

```
(parameterize ([print-extended-identifiers #f])
  (printf "~s\n~s\n"
    '(1+ --- { } .xyz)
    (string->symbol "123")))
```

prints

```
(\x31;+ \x2D;-- \x7B; \x7D; \x2E;xyz)
\x31;23
```

while

```
(parameterize ([print-extended-identifiers #t])
  (printf "~s\n~s\n"
    '(1+ --- { } .xyz)
    (string->symbol "123")))
```

prints

```
(1+ --- { } .xyz)
\x31;23
```

print-vector-length **thread parameter**
libraries: `(chezscheme)`

When `print-vector-length` is set to a true value, `write`, `put-datum`, and `pretty-print` includes the length for all vectors between the "`#`" and open parenthesis, all bytevectors between the "`#vu8`" and open parenthesis, and all fxvectors between the "`#vfx`" and open parenthesis. This parameter is set to `#f` by default.

When `print-vector-length` is set to a true value, `write`, `put-datum`, and `pretty-print` also suppress duplicated trailing elements in the vector to reduce the amount of output. This form is also recognized by the reader.

Since objects printed through the `~s` option in the format control strings of `format`, `printf`, and `fprintf` are printed as with `write`, the printing of such objects is also affected by the setting of `print-vector-length`.

```
(format "~s" (vector 'a 'b 'c 'c 'c))  ⇒  "#(a b c c c)"

(parameterize ([print-vector-length #t])
  (format "~s" (vector 'a 'b 'c 'c 'c)))  ⇒  "#5(a b c)"
```

```
(parameterize ([print-vector-length #t])
  (format "~s" (bytevector 1 2 3 4 4 4)))  ⇒  "#6vu8(1 2 3 4)"

(parameterize ([print-vector-length #t])
  (format "~s" (fxvector 1 2 3 4 4 4)))  ⇒  "#6vfx(1 2 3 4)"
```

`print-precision` **thread parameter**
libraries: `(chezscheme)`

When `print-precision` is set to `#f` (the default), `write`, `put-datum`, `pretty-print`, and the `format` "`~s`" directive do not include the vertical-bar "mantissa-width" syntax after each floating-point number. When set to a nonnegative exact integer, the mantissa width is included, as per the precision argument to `number->string`.

`print-unicode` **thread parameter**
libraries: `(chezscheme)`

When `print-unicode` is set to `#f`, `write`, `put-datum`, `pretty-print`, and the `format` "`~s`" directive display Unicode characters with encodings 80_{16} (128) and above that appear within character objects, symbols, and strings using hexadecimal character escapes. When set to a true value (the default), they are displayed like other printing characters, as if by `put-char`.

```
(format "~s" #\x3bb)  ⇒  "#\\λ"
(parameterize ([print-unicode #f])
  (format "~s" #\x3bb))  ⇒  "#\\x3BB"
```

9.15. Fasl Output

The procedures `write` and `pretty-print` print objects in a human readable format. For objects with external datum representations, the output produced by `write` and `pretty-print` is also machine-readable with `read`. Objects with external datum representations include pairs, symbols, vectors, strings, numbers, characters, booleans, and records but not procedures and ports.

An alternative *fast loading*, or *fasl*, format may be used for objects with external datum representations. The fasl format is not human readable, but it is machine readable and both more compact and more quickly processed by `read`. This format is always used for compiled code generated by `compile-file`, but it may also be used for data that needs to be written and read quickly, such as small databases encoded with Scheme data structures.

Objects are printed in fasl format with `fasl-write`. Because the fasl format is a binary format, fasl output must be written to a binary port. For this reason, it is not possible to include data written in fasl format with textual data in the same file, as was the case in earlier versions of *Chez Scheme*. Similarly, the (textual) reader does not handle objects written in fasl format; the procedure `fasl-read`, which requires a binary input port, must be used instead.

`(fasl-write` *obj binary-output-port*`)` **procedure**
`(fasl-read` *binary-input-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`fasl-write` writes the fasl representation of *obj* to *binary-output-port*. An exception is raised with condition-type `&assertion` if *obj* or any portion of *obj* has no external fasl representation, e.g., if *obj* is or contains a procedure.

`fasl-read` reads one object from *binary-input-port*, which must be positioned at the front of an object written in fasl format. `fasl-read` returns the eof object if the file is positioned at the end of file.

```
(define bop (open-file-output-port "tmp.fsl"))
(fasl-write '(a b c) bop)
(close-port bop)

(define bip (open-file-input-port "tmp.fsl"))
(fasl-read bip)  ⇒  (a b c)
(fasl-read bip)  ⇒  #!eof
(close-port bip)
```

`(fasl-file` *ifn ofn*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

ifn and *ofn* must be strings. `fasl-file` may be used to convert a file in human-readable format into an equivalent file written in fasl format. `fasl-file` reads each object in turn from the file named by *ifn* and writes the fasl format for the object onto the file named by *ofn*. If the file named by *ofn* already exists, it is replaced.

9.16. File System Interface

This section describes operations on files, directories, and pathnames.

`current-directory` **global parameter**
`cd` **global parameter**
libraries: `(chezscheme)`

When invoked without arguments, `current-directory` returns a string representing the current working directory. Otherwise, the current working directory is changed to the directory specified by the argument, which must be a string representing a valid directory pathname.

`cd` is bound to the same parameter.

`(directory-list` *path*`)` **procedure**
returns: a list of file names
libraries: `(chezscheme)`

path must be a string. The return value is a list of strings representing the names of files found in the directory named by *path*. `directory-list` raises an exception with condition type `&i/o-filename` if *path* does not name a directory or if the process cannot list the directory.

`(file-exists?` *path*`)` **procedure**
`(file-exists?` *path follow?*`)` **procedure**
returns: `#t` if the file named by *path* exists, `#f` otherwise
libraries: `(chezscheme)`

path must be a string. If the optional *follow?* argument is true (the default), `file-exists?` follows symbolic links; otherwise it does not. Thus, `file-exists?` will return `#f` when handed the pathname of a broken symbolic link unless *follow?* is provided and is `#f`.

The Revised[6] Report `file-exists?` does not accept the optional *follow?* argument. Whether it follows symbolic links is unspecified.

`(file-regular?` *path*`)` **procedure**
`(file-regular?` *path follow?*`)` **procedure**
returns: `#t` if the file named by *path* is a regular file, `#f` otherwise
libraries: `(chezscheme)`

path must be a string. If the optional *follow?* argument is true (the default), `file-regular?` follows symbolic links; otherwise it does not.

`(file-directory?` *path*`)` **procedure**
`(file-directory?` *path follow?*`)` **procedure**
returns: `#t` if the file named by *path* is a directory, `#f` otherwise
libraries: `(chezscheme)`

path must be a string. If the optional *follow?* argument is true (the default), this procedure follows symbolic links; otherwise it does not.

`(file-symbolic-link?` *path*`)` **procedure**
returns: `#t` if the file named by *path* is a symbolic link, `#f` otherwise
libraries: `(chezscheme)`

path must be a string. `file-symbolic-link?` never follows symbolic links in making its determination.

`(file-access-time` *path/port*`)` **procedure**
`(file-access-time` *path/port follow?*`)` **procedure**
returns: the access time of the specified file
`(file-change-time` *path/port*`)` **procedure**
`(file-change-time` *path/port follow?*`)` **procedure**
returns: the change time of the specified file
`(file-modification-time` *path/port*`)` **procedure**
`(file-modification-time` *path/port follow?*`)` **procedure**
returns: the modification time of the specified file
libraries: `(chezscheme)`

path/port must be a string or port. If *path/port* is a string, the time returned is for the file named by the string, and the optional *follow?* argument determines whether symbolic links are followed. If *follow?* is true (the default), this procedure follows symbolic links; otherwise it does not. If *path/port* is a port, it must be a file port, and the time returned is for the associated file. In this case, *follow?* is ignored.

The returned times are represented as time objects (Section 12.10).

`(mkdir` *path*`)` **procedure**
`(mkdir` *path mode*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. *mode* must be a fixnum.

`mkdir` creates a directory with the name given by *path*. All *path* path components leading up to the last must already exist. If the optional *mode* argument is present, it overrides the default permissions for the new directory. Under Windows, the *mode* argument is ignored.

`mkdir` raises an exception with condition type `&i/o-filename` if the directory cannot be created.

`(delete-file` *path*`)` **procedure**
`(delete-file` *path error?*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

path must be a string. `delete-file` removes the file named by *path*. If the optional *error?* argument is `#f` (the default), `delete-file` returns a boolean value: `#t` if the operation is successful and `#f` if it is not. Otherwise, `delete-file` returns an unspecified value if the operation is successful and raises an exception with condition type `&i/o-filename` if it is not.

The Revised[6] Report `delete-file` does not accept the optional *error?* argument but behaves as if *error?* is true.

`(delete-directory` *path*`)` **procedure**
`(delete-directory` *path error?*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

path must be a string. `delete-directory` removes the directory named by *path*. If the optional *error?* argument is `#f` (the default), `delete-directory` returns a boolean value: `#t` if the operation is successful and `#f` if it is not. Otherwise, `delete-directory` returns an unspecified value if the operation is successful and raises an exception with condition type `&i/o-filename` if it is not. The behavior is unspecified if the directory is not empty, but on most systems the operations will not succeed.

`(rename-file` *old-pathname new-pathname*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

old-pathname and *new-pathname* must be strings. `rename-file` changes the name of the file named by *old-pathname* to *new-pathname*. If the file does not exist or cannot be renamed, an exception is raised with condition type `&i/o-filename`.

`(chmod` *path mode*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. *mode* must be a fixnum.

`chmod` sets the permissions on the file named by *path* to *mode*. Bits 0, 1, and 2 of *mode* are the execute, write, and read permission bits for users other than the file's owner who are not in the file's group. Bits 3-5 are the execute, write, and read permission bits for users other than the file's owner but in the file's group. Bits 6-8 are the execute, write, and read permission bits for the file's owner. Bits 7-9 are the Unix sticky, set-group-id, and set-user-id bits. Under Windows, all but the user "write" bit are ignored. If the file does not exist or the permissions cannot be changed, an exception is raised with condition type `&i/o-filename`.

`(get-mode` *path*`)` **procedure**
`(get-mode` *path follow?*`)` **procedure**
returns: the current permissions mode for *path*
libraries: `(chezscheme)`

path must be a string. `get-mode` retrieves the permissions on the file named by *path* and returns them as a fixnum in the same form as the *mode* argument to `chmod`. If the optional *follow?* argument is true (the default), this procedure follows symbolic links; otherwise it does not.

`(directory-separator?` *char*`)` **procedure**
returns: `#t` if *char* is a directory separator, `#f` otherwise
libraries: `(chezscheme)`

The character `#\/` is a directory separator on all current machine types, and `#\\` is a directory separator under Windows.

`(directory-separator)` **procedure**
returns: the preferred directory separator
libraries: `(chezscheme)`

The preferred directory separator is `#\\` for Windows and `#\/` for other systems.

`(path-first` *path*`)` **procedure**
`(path-rest` *path*`)` **procedure**
`(path-last` *path*`)` **procedure**
`(path-parent` *path*`)` **procedure**
`(path-extension` *path*`)` **procedure**
`(path-root` *path*`)` **procedure**
returns: the specified component of *path*
`(path-absolute?` *path*`)` **procedure**
returns: `#t` if *path* is absolute, otherwise `#f`
libraries: `(chezscheme)`

path must be a string. The return value is also a (possibly empty) string.

The path first component is the first directory in the path, or the empty string if the path consists only of a single filename. The path rest component is the portion of the path that does not include the path first component or the directory separator (if any) that separates it from the rest of the path. The path last component is the last (filename) portion of *path*. The path parent component is the portion of *path* that does not include the path last component, if any, or the directory separator that separates it from the rest of the path.

If the first component of the path names a root directory (including drives and shares under Windows), home directory (e.g., `~/abc` or `~user/abc`), the current directory (`.`), or the parent directory (`..`), `path-first` returns that component. For paths that consist only of such a directory, both `path-first` and `path-parent` act as identity procedures, while `path-rest` and `path-last` return the empty string.

The path extension component is the portion of *path* that follows the last dot (period) in the last component of a path name. The path root component is the portion of *path* that does not include the extension, if any, or the dot that precedes it.

If the first component names a root directory (including drives and shares under Windows) or home directory, `path-absolute?` returns `#t`. Otherwise, `path-absolute?` returns `#f`.

The tables below identify the components for several example paths, with underscores representing empty strings.

path	abs	first	rest	parent	last	root	ext
a	#f	_	a	_	a	a	_
a/	#f	a	_	a	_	a/	_
a/b	#f	a	b	a	b	a/b	_
a/b.c	#f	a	b.c	a	b.c	a/b	c
/	#t	/	_	/	_	/	_
/a/b.c	#t	/	a/b.c	/a	b.c	/a/b	c
~/a/b.c	#t	~	a/b.c	~/a	b.c	~/a/b	c
~u/a/b.c	#t	~u	a/b.c	~u/a	b.c	~u/a/b	c
../..	#f	..	..	..	..	../..	_

The second table shows the components when Windows drives and shares are involved.

path	abs	first	rest	parent	last	root	ext
c:	#t	c:	_	c:	_	c:	_
c:/	#t	c:/	_	c:/	_	c:/	_
c:a/b	#t	c:	a/b	c:a	b	c:a/b	_
//s/a/b.c	#t	//s	a/b.c	//s/a	b.c	//s/a/b	c
//s.com	#t	//s.com	_	//s.com	_	//s.com	_

The following procedure can be used to reproduce the tables above.

```
(define print-table
  (lambda path*
    (define print-row
      (lambda (abs? path first rest parent last root extension)
        (printf "~a~11t~a~17t~a~28t~a~39t~a~50t~a~61t~a~73t~a\n"
          abs? path first rest parent last root extension)))
    (print-row "path" "abs" "first" "rest" "parent" "last" "root" "ext")
    (for-each
      (lambda (path)
        (define uscore (lambda (s) (if (eqv? s "") "_" s)))
        (apply print-row path
          (map (lambda (s) (if (eqv? s "") "_" s))
               (list (path-absolute? path) (path-first path)
                 (path-rest path) (path-parent path) (path-last path)
                 (path-root path) (path-extension path)))))
      path*)))
```

For example, the first table can be produced with:

```
(print-table "a" "a/" "a/b" "a/b.c" "/" "/a/b.c" "~/a/b.c"
  "~u/a/b.c" "../..")
```

while the second can be produced (under Windows) with:

```
(print-table "c:" "c:/" "c:a/b" "//s/a/b.c" "//s.com")
```

9.17. Generic Port Examples

This section presents the definitions for three types of generic ports: two-way ports, transcript ports, and process ports.

Two-way ports. The first example defines `make-two-way-port`, which constructs a textual input/output port from a given pair of textual input and output ports. For example:

```
(define ip (open-input-string "this is the input"))
(define op (open-output-string))
(define p (make-two-way-port ip op))
```

The port returned by `make-two-way-port` is both an input and an output port, and it is also a textual port:

```
(port? p)  ⇒  #t
(input-port? p)  ⇒  #t
(output-port? p)  ⇒  #t
(textual-port? p)  ⇒  #t
```

Items read from a two-way port come from the constituent input port, and items written to a two-way port go to the constituent output port:

```
(read p)  ⇒  this
(write 'hello p)
(get-output-string op)  ⇒  hello
```

The definition of `make-two-way-port` is straightforward. To keep the example simple, no local buffering is performed, although it would be more efficient to do so.

```
(define make-two-way-port
  (lambda (ip op)
    (define handler
      (lambda (msg . args)
        (record-case (cons msg args)
          [block-read (p s n) (block-read ip s n)]
          [block-write (p s n) (block-write op s n)]
          [char-ready? (p) (char-ready? ip)]
          [clear-input-port (p) (clear-input-port ip)]
          [clear-output-port (p) (clear-output-port op)]
          [close-port (p) (mark-port-closed! p)]
          [flush-output-port (p) (flush-output-port op)]
          [file-position (p . pos) (apply file-position ip pos)]
          [file-length (p) (file-length ip)]
          [peek-char (p) (peek-char ip)]
          [port-name (p) "two-way"]
          [read-char (p) (read-char ip)]
          [unread-char (c p) (unread-char c ip)]
          [write-char (c p) (write-char c op)]
          [else (assertion-violationf 'two-way-port
```

```
                "operation ~s not handled"
                msg)])))
  (make-input/output-port handler "" "")))
```

Most of the messages are passed directly to one of the constituent ports. Exceptions are `close-port`, which is handled directly by marking the port closed, `port-name`, which is also handled directly. `file-position` and `file-length` are rather arbitrarily passed off to the input port.

Transcript ports. The next example defines `make-transcript-port`, which constructs a textual input/output port from three ports: a textual input port *ip* and two textual output ports, *op* and *tp*. Input read from a transcript port comes from *ip*, and output written to a transcript port goes to *op*. In this manner, transcript ports are similar to two-way ports. Unlike two-way ports, input from *ip* and output to *op* is also written to *tp*, so that *tp* reflects both input from *ip* and output to *op*.

Transcript ports may be used to define the Scheme procedures `transcript-on` and `transcript-off`, or the *Chez Scheme* procedure `transcript-cafe`. For example, here is a definition of `transcript-cafe`:

```
(define transcript-cafe
  (lambda (pathname)
    (let ([tp (open-output-file pathname 'replace)])
      (let ([p (make-transcript-port
                 (console-input-port)
                 (console-output-port)
                 tp)])
       ; set both console and current ports so that
       ; the waiter and read/write will be in sync
        (parameterize ([console-input-port p]
                       [console-output-port p]
                       [current-input-port p]
                       [current-output-port p])
          (let-values ([vals (new-cafe)])
            (close-port p)
            (close-port tp)
            (apply values vals)))))))
```

The implementation of transcript ports is significantly more complex than the implementation of two-way ports defined above, primarily because it buffers input and output locally. Local buffering is needed to allow the transcript file to reflect accurately the actual input and output performed in the presence of `unread-char`, `clear-output-port`, and `clear-input-port`. Here is the code:

```
(define make-transcript-port
  (lambda (ip op tp)
    (define (handler msg . args)
      (record-case (cons msg args)
        [block-read (p str cnt)
         (with-interrupts-disabled
```

```
   (let ([b (port-input-buffer p)]
         [i (port-input-index p)]
         [s (port-input-size p)])
     (if (< i s)
         (let ([cnt (fxmin cnt (fx- s i))])
           (do ([i i (fx+ i 1)]
                [j 0 (fx+ j 1)])
               ((fx= j cnt)
                (set-port-input-index! p i)
                cnt)
               (string-set! str j (string-ref b i))))
         (let ([cnt (block-read ip str cnt)])
           (unless (eof-object? cnt)
             (block-write tp str cnt))
           cnt))))]
[char-ready? (p)
 (or (< (port-input-index p) (port-input-size p))
     (char-ready? ip))]
[clear-input-port (p)
 ; set size to zero rather than index to size
 ; in order to invalidate unread-char
 (set-port-input-size! p 0)]
[clear-output-port (p)
 (set-port-output-index! p 0)]
[close-port (p)
 (with-interrupts-disabled
   (flush-output-port p)
   (set-port-output-size! p 0)
   (set-port-input-size! p 0)
   (mark-port-closed! p))]
[file-position (p . pos)
 (if (null? pos)
     (most-negative-fixnum)
     (assertion-violationf 'transcript-port "cannot reposition"))]
[flush-output-port (p)
 (with-interrupts-disabled
   (let ([b (port-output-buffer p)]
         [i (port-output-index p)])
     (unless (fx= i 0)
       (block-write op b i)
       (block-write tp b i)
       (set-port-output-index! p 0)
       (set-port-bol! p
         (char=? (string-ref b (fx- i 1)) #\newline))))
   (flush-output-port op)
   (flush-output-port tp))]
[peek-char (p)
 (with-interrupts-disabled
   (let ([b (port-input-buffer p)]
         [i (port-input-index p)]
```

```
          [s (port-input-size p)])
      (if (fx< i s)
          (string-ref b i)
          (begin
            (flush-output-port p)
            (let ([s (block-read ip b)])
              (if (eof-object? s)
                  s
                  (begin
                    (block-write tp b s)
                    (set-port-input-size! p s)
                    (string-ref b 0))))))))]
[port-name (p) "transcript"]
[constituent-ports (p) (values ip op tp)]
[read-char (p)
 (with-interrupts-disabled
   (let ([c (peek-char p)])
     (unless (eof-object? c)
       (set-port-input-index! p
         (fx+ (port-input-index p) 1)))
     c))]
[unread-char (c p)
 (with-interrupts-disabled
   (let ([b (port-input-buffer p)]
         [i (port-input-index p)]
         [s (port-input-size p)])
     (when (fx= i 0)
       (assertion-violationf 'unread-char
         "tried to unread too far on ~s"
         p))
     (set-port-input-index! p (fx- i 1))
     ; following could be skipped; it's supposed
     ; to be the same character anyway
     (string-set! b (fx- i 1) c)))]
[write-char (c p)
 (with-interrupts-disabled
   (let ([b (port-output-buffer p)]
         [i (port-output-index p)]
         [s (port-output-size p)])
     (string-set! b i c)
    ; could check here to be sure that we really
    ; need to flush; we may end up here even if
    ; the buffer isn't full
     (block-write op b (fx+ i 1))
     (block-write tp b (fx+ i 1))
     (set-port-output-index! p 0)
     (set-port-bol! p (char=? c #\newline))))]
[block-write (p str cnt)
 (with-interrupts-disabled
  ; flush buffered data
```

```
          (let ([b (port-output-buffer p)]
                [i (port-output-index p)])
            (unless (fx= i 0)
              (block-write op b i)
              (block-write tp b i)
              (set-port-output-index! p 0)
              (set-port-bol! p (char=? (string-ref b (fx- i 1)) #\newline))))
          ; write new data
          (unless (fx= cnt 0)
            (block-write op str cnt)
            (block-write tp str cnt)
            (set-port-bol! p
              (char=? (string-ref str (fx- cnt 1)) #\newline))))]
      [else (assertion-violationf 'transcript-port
              "operation ~s not handled"
              msg)]))
  (let ([ib (make-string 1024)] [ob (make-string 1024)])
    (let ([p (make-input/output-port handler ib ob)])
      (set-port-input-size! p 0)
      (set-port-output-size! p (fx- (string-length ob) 1))
      p))))
```

The chosen length of both the input and output ports is the same; this is not necessary. They could have different lengths, or one could be buffered locally and the other not buffered locally. Local buffering could be disabled effectively by providing zero-length buffers.

After we create the port, the input size is set to zero since there is not yet any data to be read. The port output size is set to one less than the length of the buffer. This is done so that `write-char` always has one character position left over into which to write its character argument. Although this is not necessary, it does simplify the code somewhat while allowing the buffer to be flushed as soon as the last character is available.

Block reads and writes are performed on the constituent ports for efficiency and (in the case of writes) to ensure that the operations are performed immediately.

The call to `flush-output-port` in the handling of `read-char` insures that all output written to `op` appears before input is read from `ip`. Since `block-read` is typically used to support higher-level operations that are performing their own buffering, or for direct input and output in support of I/O-intensive applications, the flush call has been omitted from that part of the handler.

Critical sections are used whenever the handler manipulates one of the buffers, to protect against untimely interrupts that could lead to reentry into the handler. The critical sections are unnecessary if no such reentry is possible, i.e., if only one "thread" of the computation can have access to the port.

Process ports. The final example demonstrates how to incorporate the socket interface defined in Section 4.9 into a generic port that allows transparent communication with subprocesses via normal Scheme input/output operations.

A process port is created with `open-process`, which accepts a shell command as a string.

`open-process` sets up a socket, forks a child process, sets up two-way communication via the socket, and invokes the command in a subprocess.

The sample session below demonstrates the use of `open-process`, running and communicating with another Scheme process started with the "`-q`" switch to suppress the greeting and prompts.

```
> (define p (open-process "exec scheme -q"))
> (define s (make-string 1000 #\nul))
> (pretty-print '(+ 3 4) p)
> (read p)
7
> (pretty-print '(define (f x) (if (= x 0) 1 (* x (f (- x 1))))) p)
> (pretty-print '(f 10) p)
> (read p)
3628800
> (pretty-print '(exit) p)
> (read p)
#!eof
> (close-port p)
```

Since process ports, like transcript ports, are two-way, the implementation is somewhat similar. The main difference is that a transcript port reads from and writes to its subordinate ports, whereas a process port reads from and writes to a socket. When a process port is opened, the socket is created and subprocess invoked, and when the port is closed, the socket is closed and the subprocess is terminated.

```
(define open-process
  (lambda (command)
    (define handler
      (lambda (pid socket)
        (define (flush-output who p)
          (let ([i (port-output-index p)])
            (when (fx> i 0)
              (check who (c-write socket (port-output-buffer p) i))
              (set-port-output-index! p 0))))
        (lambda (msg . args)
          (record-case (cons msg args)
            [block-read (p str cnt)
             (with-interrupts-disabled
               (let ([b (port-input-buffer p)]
                     [i (port-input-index p)]
                     [s (port-input-size p)])
                 (if (< i s)
                     (let ([cnt (fxmin cnt (fx- s i))])
                       (do ([i i (fx+ i 1)]
                            [j 0 (fx+ j 1)])
                           ((fx= j cnt)
                            (set-port-input-index! p i)
                            cnt)
```

```
              (string-set! str j (string-ref b i))))
          (begin
            (flush-output 'block-read p)
            (let ([n (check 'block-read
                       (c-read socket str cnt))])
              (if (fx= n 0)
                  #!eof
                  n))))))]
[char-ready? (p)
 (or (< (port-input-index p) (port-input-size p))
     (bytes-ready? socket))]
[clear-input-port (p)
 ; set size to zero rather than index to size
 ; in order to invalidate unread-char
 (set-port-input-size! p 0)]
[clear-output-port (p) (set-port-output-index! p 0)]
[close-port (p)
 (with-interrupts-disabled
   (flush-output 'close-port p)
   (set-port-output-size! p 0)
   (set-port-input-size! p 0)
   (mark-port-closed! p)
   (terminate-process pid))]
[file-length (p) 0]
[file-position (p . pos)
 (if (null? pos)
     (most-negative-fixnum)
     (assertion-violationf 'process-port "cannot reposition"))]
[flush-output-port (p)
 (with-interrupts-disabled
   (flush-output 'flush-output-port p))]
[peek-char (p)
 (with-interrupts-disabled
   (let ([b (port-input-buffer p)]
         [i (port-input-index p)]
         [s (port-input-size p)])
     (if (fx< i s)
         (string-ref b i)
         (begin
           (flush-output 'peek-char p)
           (let ([s (check 'peek-char
                      (c-read socket b (string-length b)))])
             (if (fx= s 0)
                 #!eof
                 (begin (set-port-input-size! p s)
                        (string-ref b 0)))))))))]
[port-name (p) "process"]
[read-char (p)
 (with-interrupts-disabled
   (let ([b (port-input-buffer p)]
```

```
                    [i (port-input-index p)]
                    [s (port-input-size p)])
               (if (fx< i s)
                   (begin
                     (set-port-input-index! p (fx+ i 1))
                     (string-ref b i))
                   (begin
                     (flush-output 'peek-char p)
                     (let ([s (check 'read-char
                                (c-read socket b (string-length b)))])
                       (if (fx= s 0)
                           #!eof
                           (begin (set-port-input-size! p s)
                                  (set-port-input-index! p 1)
                                  (string-ref b 0)))))))]
          [unread-char (c p)
           (with-interrupts-disabled
             (let ([b (port-input-buffer p)]
                   [i (port-input-index p)]
                   [s (port-input-size p)])
               (when (fx= i 0)
                 (assertion-violationf 'unread-char
                   "tried to unread too far on ~s"
                   p))
               (set-port-input-index! p (fx- i 1))
              ; following could be skipped; supposed to be
              ; same character
               (string-set! b (fx- i 1) c)))]
          [write-char (c p)
           (with-interrupts-disabled
             (let ([b (port-output-buffer p)]
                   [i (port-output-index p)]
                   [s (port-output-size p)])
               (string-set! b i c)
               (check 'write-char (c-write socket b (fx+ i 1)))
               (set-port-output-index! p 0)))]
          [block-write (p str cnt)
           (with-interrupts-disabled
            ; flush buffered data
             (flush-output 'block-write p)
            ; write new data
             (check 'block-write (c-write socket str cnt)))]
          [else
           (assertion-violationf 'process-port
             "operation ~s not handled"
             msg)])))
  (let* ([server-socket-name (tmpnam 0)]
         [server-socket (setup-server-socket server-socket-name)])
    (dofork
      (lambda () ; child
```

```
  (check 'close (close server-socket))
  (let ([sock (setup-client-socket server-socket-name)])
    (dodup 0 sock)
    (dodup 1 sock))
  (check 'execl (execl4 "/bin/sh" "/bin/sh" "-c" command))
  (assertion-violationf 'open-process "subprocess exec failed"))
(lambda (pid) ; parent
  (let ([sock (accept-socket server-socket)])
    (check 'close (close server-socket))
    (let ([ib (make-string 1024)] [ob (make-string 1024)])
      (let ([p (make-input/output-port
                 (handler pid sock)
                 ib ob)])
        (set-port-input-size! p 0)
        (set-port-output-size! p (fx- (string-length ob) 1))
        p)))))))
```

10. Libraries and Top-level Programs

The Revised[6] Report describes two units of portable code: libraries and top-level programs. A library is a named collection of bindings with a declared set of explicitly exported bindings, a declared set of imported libraries, and a body that initializes its bindings. A top-level program is a stand-alone program with a declared set of imported libraries and a body that is run when the top-level program is run. The bindings in a library are created and its initialization code run only if the library is used, directly or indirectly, by a top-level program.

The `import` declarations appearing within libraries and top-level programs serve two purposes: first, they cause the imported libraries to be loaded, and second, they cause the bindings of the imported libraries to become visible in the importing library or top-level program. Libraries are typically stored in the file system, with one library per file, and the library name typically identifies the file-system path to the library, possibly relative to a default or programmer-specified set of library locations. The exact mechanism by which top-level programs are run and libraries are loaded is implementation-dependent.

This chapter describes the mechanisms by which libraries and programs are loaded in *Chez Scheme* along with various features for controlling and tracking this process. It also describes the set of built-in libraries and syntactic forms for defining new libraries and top-level programs outside of a library or top-level program file.

10.1. Built-in Libraries

In addition to the RNRS libraries mandated by the Revised[6] Report:

```
(rnrs base (6))
(rnrs arithmetic bitwise (6))
(rnrs arithmetic fixnums (6))
(rnrs arithmetic flonums (6))
(rnrs bytevectors (6))
(rnrs conditions (6))
(rnrs control (6))
(rnrs enums (6))
(rnrs eval (6))
(rnrs exceptions (6))
(rnrs files (6))
(rnrs hashtables (6))
```

```
(rnrs io ports (6))
(rnrs io simple (6))
(rnrs lists (6))
(rnrs mutable-pairs (6))
(rnrs mutable-strings (6))
(rnrs programs (6))
(rnrs r5rs (6))
(rnrs records procedural (6))
(rnrs records syntactic (6))
(rnrs records inspection (6))
(rnrs sorting (6))
(rnrs syntax-case (6))
(rnrs unicode (6))
```

Chez Scheme also provides two additional libraries: `(chezscheme)` and `(chezscheme csv7)`. The former can also be referenced as `(scheme)` and the latter can also be referenced as `(scheme csv7)`.

The `(chezscheme)` library exports bindings for every identifier whose binding is described in this document, including those for keywords like `lambda`, auxiliary keywords like `else`, module names like `scheme`, and procedure names like `cons`. In most cases where an identifier exported from the `(chezscheme)` library corresponds to an identifier exported from one of the RNRS libraries, the bindings are identical. In some cases, however, the `(chezscheme)` bindings extend the `rnrs` bindings in some way. For example, the `(chezscheme)` `syntax-rules` form allows its clauses to have fenders (Section 11.2), while the `(rnrs)` `syntax-rules` form does not. Similarly, the `(chezscheme)` `current-input-port` procedure accepts an optional *port* argument that, when specified, sets the current input port to *port* (Section 9.8), while the `(rnrs)` `current-input-port` procedure does not. When the `(chezscheme)` library extends an RNRS binding in some way, the `(chezscheme)` library also exports the RNRS version, with the name prefixed by `r6rs:`, e.g., `r6rs:syntax-rules` or `r6rs:current-input-port`.

The `(chezscheme csv7)` Version 7 backward compatibility library contains bindings for a set of syntactic forms and procedures whose syntax or semantics directly conflicts with the RNRS bindings for the same identifiers. The following identifiers are exported from `(chezscheme csv7)`.

```
record-field-accessible?
record-field-accessor
record-field-mutable?
record-field-mutator
record-type-descriptor
record-type-field-decls
record-type-field-names
record-type-name
record-type-symbol
```

The bindings of this library should be used only for old code; new code should use the RNRS variants. Each of these is also available in the `(chezscheme)` library with the prefix `csv7:`, e.g., `csv7:record-type-name`.

The interaction environment in which code outside of a library or RNRS top-level program is scoped contains all of the bindings of the `(chezscheme)` library, as described in Section 2.3.

10.2. Running Top-level Programs

A top-level program must reside in its own file, which may have any name and may reside anywhere in the file system. A top-level program residing in a file is run by one of three mechanisms: the `scheme-script` command, the `--program` command-line argument, or the `load-program` procedure.

The `scheme-script` command is used as follows:

`scheme-script` *program-filename arg* `...`

It may also be run implicitly on Unix-based systems by placing the line

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

at the front of the file containing the top-level program, making the top-level program file executable, and executing the file. This line may be replaced with

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

with `/usr/bin` replaced by the absolute path to the directory containing `scheme-script` if it is not in `/usr/bin`. The first form is recommended in the nonnormative appendices to the Revised[6] Report [29], and works wherever `scheme-script` appears in the path.

The `--program` command is used similarly with the `scheme` or `petite` executables, either by running:

`scheme --program` *program-filename arg* `...`
`petite --program` *program-filename arg* `...`

or by including

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

or

```
#! /usr/bin/petite --script
```

at the front of the top-level program file, making the file executable, and executing the file. Again, `/usr/bin` should be replaced with the absolute path to the actual directory in which `scheme` and/or `petite` resides, if not `/usr/bin`.

The `load-program` procedure, described in Section 12.4, is used like `load`:

`(load-program` *string*`)`

where *string* names the file in which the top-level program resides.

Regardless of the mechanism used, if the opening line is in one of the forms described above, or more generally, consists of `#!` followed by a space or a forward slash, the opening line is not considered part of the program and is ignored once the Scheme system starts up and begins to run the program. Thus, the line may be present even in a file loaded by `load-program`. In fact, `load-program` is ultimately used by the other two mechanisms described above, via the value of the `scheme-program` parameter described in Section 12.8, and it is `load-program` that scans past the `#!` line, if present, before evaluating the program.

A top-level program may be compiled with the `compile-program` procedure described in Section 12.4. `compile-program` copies the `#!` line from the source file to the object file, followed by a compiled version of the source code. Any libraries upon which the top-level program depends, other than built-in libraries, must be compiled first via `compile-file` or `compile-library`. This can be done manually or by setting the parameter `compile-imported-libraries` to `#t` before compiling the program. The program must be recompiled if any of the libraries upon which it depends are recompiled. A compiled top-level program can be run just like a source top-level program via each of the mechanisms described above.

In *Chez Scheme*, a library may also be defined in the REPL or placed in a file to be loaded via `load` or `load-library`. The syntax for a library is the same whether the library is placed in its own file and implicitly loaded via `import`, entered into the REPL, or placed in a file along with other top-level expressions to be evaluated by `load`. A top-level program may also be defined in the REPL or placed in a file to be loaded via `load`, but in this case, the syntax is slightly different. In the language of the Revised[6] Report, a top-level program is merely an unwrapped sequence of subforms consisting of an `import` form and a body, delimited only by the boundaries of the file in which it resides. In order for a top-level program to be entered in the REPL or placed in a file to be evaluated by `load`, *Chez Scheme* allows top-level programs to be enclosed in a `top-level-program` form.

10.3. Library and Top-level Program Forms

`(library` *name exports imports library-body*`)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

The `library` form defines a new library with the specified name, exports, imports, and body. Details on the syntax and semantics of the library form are given in Section 10.3 of *The Scheme Programming Language, 4th Edition* and in the Revised[6] Report.

Only one version of a library can be loaded at any given time, and an exception is raised if a library is implicitly loaded via `import` when another version of the library has already been loaded. *Chez Scheme* permits a different version of the library, or a new instance of the same version, to be entered explicitly into the REPL or loaded explicitly from a file, to facilitate interactive testing and debugging. The programmer should take care to make sure that any code that uses the library is also reentered or reloaded, to make sure that code accesses the bindings of the new instance of the library.

```
(library (test (1)) (export x) (import (rnrs)) (define x 3))
(import (test))
(define f (lambda () x))
(f)  ⇒  3

(library (test (1)) (export x) (import (rnrs)) (define x 4))
(import (test))
(f)  ⇒  3    ; oops---forgot to redefine f
(define f (lambda () x))
(f)  ⇒  4

(library (test (2)) (export x) (import (rnrs)) (define x 5))
(import (test))
(define f (lambda () x))
(f)  ⇒  5
```

As with module imports (Section 11.5), a library `import` may appear anywhere a definition may appear, including at top level in the REPL, in a file to be loaded by `load`, or within a `lambda`, `let`, `letrec`, `letrec*`, etc., body. The same `import` form may be used to import from both libraries and modules.

```
(library (foo) (export a) (import (rnrs)) (define a 'a-from-foo))
(module bar (b) (define b 'b-from-bar))
(let () (import (foo) bar) (list a b))  ⇒  (a-from-foo b-from-bar)
```

The `import` keyword is not visible within a library body unless the library imports it from the `(chezscheme)` library.

`(top-level-program` *imports body*`)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

A `top-level-program` form may be entered into the REPL or placed in a file to be loaded via `load`, where it behaves as if its subforms were placed in a file and loaded via `load-program`. Details on the syntax and semantics of a top-level program are given in Section 10.3 of *The Scheme Programming Language, 4th Edition* and in the Revised[6] Report.

The following transcript illustrates a `top-level-program` being tested in the REPL.

```
> (top-level-program (import (rnrs))
    (display "hello!\n"))
hello!
```

10.4. Standalone import and export forms

Although not required by the Revised[6] Report, *Chez Scheme* supports the use of standalone import and export forms. The import forms can appear anywhere other definitions can appear, including within a `library` body, `module` (Section 11.5) body, `lambda` or other local

body, and at top level. The export forms can appear within the definitions of a `library` or `module` body to specify additional exports for the library or module.

Within a library or top-level program, the keywords for these forms must be imported from the `(chezscheme)` library to be available for use, since they are not defined in any of the Revised[6] Report libraries.

`(import` *import-spec* `...)` **syntax**
`(import-only` *import-spec* `...)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

An `import` or `import-only` form is a definition and can appear anywhere other definitions can appear, including at the top level of a program, nested within the bodies of `lambda` expressions, and nested within modules and libraries.

Each *import-spec* must take one of the following forms.

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

The `for` wrapper and *import-level* are described in Chapter 10 of *The Scheme Programming Language, 4th Edition*. They are ignored by *Chez Scheme*, which determines automatically the levels at which identifiers must be imported, as permitted by the Revised[6] Report. This frees the programmer from the obligation to do so and results in more generality as well as more precision in the set of libraries actually imported at compile and run time [21, 19].

An *import-set* must take one of the following forms:

library-spec
module-name
`(only` *import-set identifier* `...)`
`(except` *import-set identifier* `...)`
`(prefix` *import-set prefix*`)`
`(add-prefix` *import-set prefix*`)`
`(drop-prefix` *import-set prefix*`)`
`(rename` *import-set* `(`*import-name internal-name*`) ...)`
`(alias` *import-set* `(`*import-name internal-name*`) ...)`

Several of these are specified by the Revised[6] Report; the remainder are *Chez Scheme* extensions, including *module-name* and the `add-prefix`, `drop-prefix`, and `alias` forms.

An `import` or `import-only` form makes the specified bindings visible in the scope in which they appear. Except at top level, they differ in that `import` leaves all bindings except for those shadowed by the imported names visible, whereas `import-only` hides all existing bindings, i.e., makes only the imported names visible. At top level, `import-only` behaves like `import`.

Each *import-set* identifies a set of names to make visible as follows.

library-spec: all exports of the library identified by the Revised[6] Report *library-spec* (Chapter 10.

module-name: all exports of module named by the identifier *module-name*

`(only` *import-set identifier* `...)`: of those specified by *import-set*, just *identifier* `...`

`(except` *import-set identifier* `...)`: all specified by *import-set* except *identifier* `...`

`(prefix` *import-set prefix*`)`: all specified by *import-set*, each prefixed by *prefix*

`(add-prefix` *import-set prefix*`)`: all specified by *import-set*, each prefixed by *prefix* (just like `prefix`)

`(drop-prefix` *import-set prefix*`)`: all specified by *import-set*, with prefix *prefix* removed

`(rename` *import-set* `(`*import-name internal-name*`) ...)`: all specified by *import-set*, with each identifier *import-name* renamed to the corresponding identifier *internal-name*

`(alias` *import-set* `(`*import-name internal-name*`) ...)`: all specified by *import-set*, with each *internal-name* as an alias for *import-name*

The `alias` form differs from the `rename` form in that both *import-name* and *internal-name* are in the resulting set, rather than just *internal-name*.

It is a syntax violation if the given selection or transformation cannot be made because of a missing export or prefix.

An identifier made visible via an import of a module or library is scoped as if its definition appears where the import occurs. The following example illustrates these scoping rules, using a local module `m`.

```
(library (A) (export x) (import (rnrs)) (define x 0))
(let ([x 1])
  (module m (x setter)
    (define-syntax x (identifier-syntax z))
    (define setter (lambda (x) (set! z x)))
    (define z 2))
  (let ([y x] [z 3])
    (import m (prefix (A) a:))
    (setter 4)
    (list x a:x y z)))  ⇒  (4 0 1 3)
```

The inner `let` expression binds `y` to the value of the `x` bound by the outer `let`. The import of `m` makes the definitions of `x` and `setter` visible within the inner `let`. The import of `(A)` makes the variable `x` exported from `(A)` visible as `a:x` within the body of the inner `let`. Thus, in the expression `(list x a:x y z)`, `x` refers to the identifier macro exported from `m` while `a:x` refers to the variable `x` exported from `(A)` and `y` and `z` refer to the bindings established by the inner `let`. The identifier macro `x` expands into a reference to the variable `z` defined within the module.

With local import forms, it is rarely necessary to use the extended import specifiers. For example, an abstraction that encapsulates the import and reference can easily be defined and used as follows.

```
(define-syntax from
  (syntax-rules ()
    [(_ m id) (let () (import-only m) id)]))
```

```
(library (A) (export x) (import (rnrs)) (define x 1))
(let ([x 10])
  (module M (x) (define x 2))
  (cons (from (A) x) (from M x)))  ⇒  (1 . 2)
```

The definition of `from` could use `import` rather than `import-only`, but by using `import-only` we get feedback if an attempt is made to import an identifier from a library or module that does not export the identifier. With `import` instead of `import-only`, the current binding, if any, would be visible if the library or module does not export the specified name.

```
(define-syntax lax-from
  (syntax-rules ()
    [(_ m id) (let () (import m) id)]))

(library (A) (export x) (import (rnrs)) (define x 1))

(let ([x 10])
  (module M (x) (define x 2))
  (+ (from (A) x) (from M y)))  ⇒  exception: unbound identifier y

(let ([x 10] [y 20])
  (module M (x) (define x 2))
  (+ (lax-from (A) x) (lax-from M y)))  ⇒  21
```

Import visibility interacts with hygienic macro expansion in such a way that, as one might expect, an identifier x imported from a module M is treated in the importing context as if the corresponding export identifier had been present in the import form along with M.

The `from` abstraction above works because both M and id appear in the input to the abstraction, so the imported id captures the reference to id.

The following variant of *from* also works, because both names are introduced into the output by the transformer.

```
(module M (x) (define x 'x-of-M))
(define-syntax x-from-M
  (syntax-rules ()
    [(_) (let () (import M) x)]))

(let ([x 'local-x]) (x-from-M))  ⇒  x-of-M
```

On the other hand, imports of introduced module names do not capture free references.

```
(let ([x 'local-x])
  (define-syntax alpha
    (syntax-rules ()
      [(_ var) (let () (import M) (list x var))]))

  (alpha x))  ⇒  (x-of-M local-x)
```

Similarly, imports from free module names do not capture references to introduced variables.

```
(let ([x 'local-x])
  (define-syntax beta
    (syntax-rules ()
      [(_ m var) (let () (import m) (list x var))])))

  (beta M x))  ⇒  (local-x x-of-M)
```

This semantics extends to prefixed, renamed, and aliased bindings created by the extended `import` specifiers `prefix`, `rename`, and `alias`.

The `from` abstraction works for variables but not for exported keywords, record names, or module names, since the output is an expression and may thus appear only where expressions may appear. A generalization of this technique is used in the following definition of `import*`, which supports renaming of imported bindings and selective import of specific bindings—without the use of the built-in `import` subforms for selecting and renaming identifiers

```
(define-syntax import*
  (syntax-rules ()
    [(_ m) (begin)]
    [(_ m (new old))
     (module (new)
       (module (tmp)
         (import m)
         (alias tmp old))
       (alias new tmp))]
    [(_ m id) (module (id) (import m))]
    [(_ m spec0 spec1 ...)
     (begin (import* m spec0) (import* m spec1 ...))]))
```

To selectively import an identifier from module or library `m`, the `import*` form expands into an anonymous module that first imports all exports of `m` then re-exports only the selected identifier. To rename on import the macro expands into an anonymous module that instead exports an alias (Section 11.10) bound to the new name.

If the output placed the definition of `new` in the same scope as the import of `m`, a naming conflict would arise whenever `new` is also present in the interface of `m`. To prevent this, the output instead places the import within a nested anonymous module and links `old` and `new` by means of an alias for the introduced identifier `tmp`.

The macro expands recursively to handle multiple import specifications. Each of the following examples imports `cons` as `+` and `+` as `cons`, which is probably not a very good idea.

```
(let ()
  (import* scheme (+ cons) (cons +))
  (+ (cons 1 2) (cons 3 4)))  ⇒  (3 . 7)

(let ()
  (import* (rnrs) (+ cons) (cons +))
  (+ (cons 1 2) (cons 3 4)))  ⇒  (3 . 7)
```

`(export` *export-spec* `...)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

An `export` form is a definition and can appear with other definitions at the front of a `library` or `module`. It is a syntax error for an `export` form to appear in other contexts, including at top level or among the definitions of a top-level program or `lambda` body.

Each *export-spec* must take one of the following forms.

identifier
`(rename (`*internal-name export-name*`) ...)`
`(import` *import-spec* `...)`

where each *internal-name* and *export-name* is an identifier. The first two are syntactically identical to `library` *export-spec*s, while the third is syntactically identical to a *Chez Scheme* `import` form, which is an extension of the R6RS library `import` subform. The first form names a single export, *identifier*, whose export name is the same as its internal name. The second names a set of exports, each of whose export name is given explicitly and may differ from its internal name.

For the third, the identifiers identified by the `import` form become exports, with aliasing, renaming, prefixing, etc., as specified by the *import-spec*s. The module or library whose bindings are exported by an `import` form appearing within an `export` form can be defined within or outside the exporting module or library and need not be imported elsewhere within the exporting module or library.

The following library exports a two-armed-only variant of `if` along with all remaining bindings of the `(rnrs)` library.

```
(library (rnrs-no-one-armed-if) (export) (import (except (chezscheme) if))
  (export if (import (except (rnrs) if)))
  (define-syntax if
    (let ()
      (import (only (rnrs) if))
      (syntax-rules ()
        [(_ tst thn els) (if tst thn els)]))))

(import (rnrs-no-one-armed-if))
(if #t 3 4)  ⇒  3
(if #t 3)  ⇒  exception: invalid syntax
```

Another way to define the same library would be to define the two-armed-only `if` with a different internal name and use `rename` to export it under the name `if`:

```
(library (rnrs-no-one-armed-if) (export) (import (chezscheme))
  (export (rename (two-armed-if if)) (import (except (rnrs) if)))
  (define-syntax two-armed-if
    (syntax-rules ()
      [(_ tst thn els) (if tst thn els)])))
```

```
(import (rnrs-no-one-armed-if))
(if #t 3 4)  ⇒  3
(if #t 3)  ⇒  exception: invalid syntax
```

The placement of the `export` form in the library body is irrelevant, e.g., the `export` form can appear after the definition in the examples above.

`(indirect-export` *id indirect-id* `...)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

This form is a definition and can appear wherever any other definition can appear.

An `indirect-export` form declares that the named *indirect-id*s are indirectly exported to top level if *id* is exported to top level.

In general, if an identifier is not directly exported by a library or module, it can be referenced outside of the library or module only in the expansion of a macro defined within and exported from the library or module. Even this cannot occur for libraries or modules defined at top level (or nested within other libraries or modules), unless either (1) the library or module has been set up to implicitly export all identifiers as indirect exports, or (2) each indirectly exported identifier is explicitly declared as an indirect export of some other identifier that is exported, either directly or indirectly, from the library or module, via an `indirect-export` or the built-in indirect export feature of a `module` export subform. By default, (1) is true for a library and false for a module, but the default can be overridden via the `implicit-exports` form, which is described below.

This form is meaningful only within a top-level library, top-level module, or module enclosed within a library or top-level module, although it has no effect if the library or module already implicitly exports all bindings. It is allowed anywhere else definitions can appear, however, so macros that expand into indirect export forms can be used in any definition context.

Indirect exports are listed so the compiler can determine the exact set of bindings (direct and indirect) that must be inserted into the top-level environment, and conversely, the set of bindings that may be treated more efficiently as local bindings (and perhaps discarded, if they are not used).

In the example below, `indirect-export` is used to indirectly export `count` to top level when `current-count` is exported to top level.

```
(module M (bump-count current-count)
  (define-syntax current-count (identifier-syntax count))
  (indirect-export current-count count)
  (define count 0)
  (define bump-count
    (lambda ()
      (set! count (+ count 1)))))
```

```
(import M)
(bump-count)
current-count ⇒ 1
count ⇒ exception: unbound identifier count
```

An `indirect-export` form is not required to make `count` visible for `bump-count`, since it is a procedure whose code is contained within the module rather than a macro that might expand into a reference to `count` somewhere outside the module.

It is often useful to use `indirect-export` in the output of a macro that expands into another macro named *a* if *a* expands into references to identifiers that might not be directly exported, as illustrated by the alternative definition of module `M` above.

```
(define-syntax define-counter
  (syntax-rules ()
    [(_ getter bumper init incr)
     (begin
       (define count init)
       (define-syntax getter (identifier-syntax count))
       (indirect-export getter count)
       (define bumper
         (lambda ()
           (set! count (incr count)))))]))

(module M (bump-count current-count)
  (define-counter current-count bump-count 0 add1))
```

`(implicit-exports #t)` **syntax**
`(implicit-exports #f)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

An `implicit-exports` form is a definition and can appear with other definitions at the front of a `library` or `module`. It is a syntax error for an `implicit-exports` form to appear in other contexts, including at top level or among the definitions of a top-level program or `lambda` body.

The `implicit-exports` form determines whether identifiers not directly exported from a module or library are automatically indirectly exported to the top level if any meta-binding (keyword, meta definition, or property definition) is directly exported to top level from the library or module. The default for libraries is `#t`, to match the behavior required by the Revised[6] Report, while the default for modules is `#f`. The `implicit-exports` form is meaningful only within a library, top-level module, or module enclosed within a library or top-level module. It is allowed in a module enclosed within a `lambda`, `let`, or similar body, but ignored there because none of that module's bindings can be exported to top level.

The advantage of `(implicit-exports #t)` is that indirect exports need not be listed explicitly, which is convenient. A disadvantage is that it often results in more bindings than necessary being elevated to top level where they cannot be discarded as useless by the optimizer. For modules, another disadvantage is such bindings cannot be proven immutable,

which inhibits important optimizations such as procedure inlining. This can result in significantly lower run-time performance.

10.5. Library Parameters

The parameters described below control where `import` looks when attempting to load a library, whether it compiles the libraries it loads, and whether it displays tracking messages as it performs its search.

`library-directories` **thread parameter**
`library-extensions` **thread parameter**
libraries: `(chezscheme)`

The parameter `library-directories` determines where the files containing library source and object code are located in the file system, and the parameter `library-extensions` determines the filename extensions for the files holding the code, as described in section 2.5. The values of both parameters are lists of pairs of strings. The first string in each `library-directories` pair identifies a source-file root directory, and the second identifies the corresponding object-file root directory. Similarly, the first string in each `library-extensions` pair identifies a source-file extension, and the second identifies the corresponding object-file extension. The full path of a library source or object file consists of the source or object root followed by the components of the library name prefixed by slashes, with the library extension added on the end. For example, for root `/usr/lib/scheme`, library name `(app lib1)`, and extension `.sls`, the full path is `/usr/lib/scheme/app/lib1.sls`. If the library name portion forms an absolute pathname, e.g., `~/.myappinit`, the `library-directories` parameter is ignored and no prefix is added.

The initial values of these parameters are shown below.

```
(library-directories)  ⇒  (("." . "."))

(library-extensions)  ⇒  ((".chezscheme.sls" . ".chezscheme.so")
                          (".ss" . ".so")
                          (".sls" . ".so")
                          (".scm" . ".so")
                          (".sch" . ".so"))
```

As a convenience, when either of these parameters is set, any element of the list can be specified as a single source string, in which case the object string is determined automatically. For `library-directories`, the object string is the same as the source string, effectively naming the same directory as a source- and object-code root. For `library-extensions`, the object string is the result of removing the last (or only) extension from the string and appending `".so"`. The `library-directories` and `library-extensions` parameters also accept as input strings in the format described in Section 2.5 for the `--libdirs` and `--libexts` command-line options.

`compile-imported-libraries` **thread parameter**
libraries: `(chezscheme)`

When the value of this parameter is `#t`, `import` automatically calls the value of the `compile-library-handler` parameter (which defaults to a procedure that simply calls `compile-library`) on any imported library if the object file is missing, older than the corresponding source file, older than any source files included (via `include`) when the object file was created, or itself requires a library that has or must be recompiled, as described in Section 2.5. The default initial value of this parameter is `#f`. It can be set to `#t` via the command-line option `--compile-imported-libraries`.

When `import` compiles a library via this mechanism, it does not also load the compiled library, because this would cause portions of library to be reevaluated. Because of this, run-time expressions in the file outside of a `library` form will not be evaluated. If such expressions are present and should be evaluated, the library should be loaded explicitly.

`import-notify` **thread parameter**
libraries: `(chezscheme)`

When the new parameter `import-notify` is set to a true value, `import` displays messages to the console-output port as it searches for the file containing each library it needs to load. The default value of this parameter is `#f`.

10.6. Library Inspection

`(library-list)` **procedure**
returns: a list of the libraries currently defined
libraries: `(chezscheme)`

The set of libraries initially defined includes those listed in Section 10.1 above.

`(library-version` *libref*`)` **procedure**
returns: the version of the specified library

`(library-exports` *libref*`)` **procedure**
returns: a list of the exports of the specified library

`(library-requirements` *libref*`)` **procedure**
returns: a list of libraries required by the specified library

`(library-requirements` *libref options*`)` **procedure**
returns: a list of libraries required by the specified library, filtered by *options*

`(library-object-filename` *libref*`)` **procedure**
returns: the name of the object file holding the specified library, if any
libraries: `(chezscheme)`

Information can be obtained only for built-in libraries or libraries previously loaded into the system. *libref* must be an s-expression in the form of a library reference. The syntax for library references is given in Chapter 10 of *The Scheme Programming Language, 4th Edition* and in the Revised[6] Report.

The `library-version` return value is a list of numbers (possibly empty) representing the library's version.

The list of exports returned by `library-exports` is a list of symbols, each identifying one of the library's exports. The order in which the elements appear is unspecified.

When the optional *options* argument is supplied, it must be an enumeration set over the symbols constituting valid library-requirements options, as described in the `library-requirements-options` entry below. It defaults to a set containing all of the options. Each element of the list of libraries returned by `library-requirements` is an s-expression form of a library reference. The library reference includes the actual version of the library that is present in the system (if nonempty), even if a version was not specified when it was imported. The order in which the libraries appear in the list returned by `library-requirements` is unspecified.

`library-object-filename` returns a string naming the object file if the specified library was loaded from or compiled to an object file. Otherwise, it returns `#f`.

```
(with-output-to-file "A.ss"
  (lambda ()
    (pretty-print
      '(library (A (1 2)) (export x z)
         (import (rnrs))
         (define x 'ex)
         (define y 23)
         (define-syntax z
           (syntax-rules ()
             [(_ e) (+ y e)])))))
  'replace)
(with-output-to-file "B.ss"
  (lambda ()
    (pretty-print
      '(library (B) (export x w)
         (import (rnrs) (A))
         (define w (cons (z 12) x)))))
  'replace)
(compile-imported-libraries #t)
(import (B))
(library-exports '(A))  ⇒  (x z) ; or (z x)
(library-exports '(A (1 2)))  ⇒  (x z) ; or (z x)
(library-exports '(B))  ⇒  (x w) ; or (w x)
(library-version '(A))  ⇒  (1 2)
(library-version '(B))  ⇒  ()
(library-requirements '(A))  ⇒  ((rnrs (6)))
(library-requirements '(B))  ⇒  ((rnrs (6)) (A (1 2)))
(library-object-filename '(A))  ⇒  "A.so"
(library-object-filename '(B))  ⇒  "B.so"
```

`(library-requirements-options` *symbol* `...)` **syntax**
returns: a library-requirements-options enumeration set
libraries: `(chezscheme)`

Library-requirements-options enumeration sets are passed to `library-requirements` to determine the library requirements to be listed. The available options are described below.

`import`: Include the libraries that must be imported when the specified library is imported.

`visit@visit`: Includes the libraries that must be visited when the specified library is visited.

`invoke@visit`: Include the libraries that must be invoked when the specified library is visited.

`invoke`: Includes the libraries that must be invoked when the specified library is invoked.

11. Syntactic Extension and Modules

This chapter describes the *Chez Scheme* extensions to the syntax-case syntactic abstraction mechanism now standardized in the Revised[6] Report. These extensions include the module system (Section 11.5), meta definitions (Section 11.8), conditional expansion (Section 11.9) `syntax-rules` fenders, `fluid-let-syntax`, and `include`.

11.1. Fluid Keyword Bindings

Keyword bindings established via the Revised[6] Report `define-syntax`, `let-syntax`, or `letrec-syntax` forms may be rebound temporarily with `fluid-let-syntax`.

`(fluid-let-syntax ((`*keyword expr*`) ...)` *form*$_1$ *form*$_2$ `...)` **syntax**
returns: see explanation
libraries: `(chezscheme)`

Each *expr* must evaluate to a transformer. `fluid-let-syntax` is similar to the standard `let-syntax`, except that instead of introducing new bindings for the keywords *keyword* ..., `fluid-let-syntax` temporarily alters the existing bindings for the keywords during the expansion of its body. That is, during the expansion of *form*$_1$ *form*$_2$ ..., the visible lexical (or top-level) binding for each `keyword` is temporarily replaced by a new association between the keyword and the corresponding transformer. This affects any references to the keyword that resolve to the same lexical (or top-level) binding whether the references occur in the text of the body or are introduced during its expansion. In contrast, `let-syntax` captures only those references that occur within the text of its body.

The following example shows how `fluid-let-syntax` differs from `let-syntax`.

```
(let ([f (lambda (x) (+ x 1))])
  (let-syntax ([g (syntax-rules ()
                    [(_ x) (f x)])])
    (let-syntax ([f (syntax-rules ()
                      [(_ x) x])])
      (g 1))))  ⇒  2
```

```
(let ([f (lambda (x) (+ x 1))])
  (let-syntax ([g (syntax-rules ()
                    [(_ x) (f x)])])
    (fluid-let-syntax ([f (syntax-rules ()
                            [(_ x) x])])
      (g 1))))  ⇒  1
```

The two expressions are identical except that the inner `let-syntax` form in the first expression is a `fluid-let-syntax` form in the second. In the first expression, the `f` occurring in the expansion of `(g 1)` refers to the `let`-bound variable `f`, whereas in the second it refers to the keyword `f` by virtue of the fluid syntax binding for `f`.

The following code employs `fluid-let-syntax` in the definition of a `define-integrable` form that is similar to `define` for procedure definitions except that it causes the code for the procedure to be *integrated*, or inserted, wherever a direct call to the procedure is found. No semantic difference is visible between procedures defined with `define-integrable` and those defined with `define`, except that a top-level `define-integrable` form must appear before the first reference to the defined identifier. Lexical scoping is preserved, the actual parameters in an integrated call are evaluated once and at the proper time, integrable procedures may be used as first-class values, and recursive procedures do not cause indefinite recursive expansion.

```
(define-syntax define-integrable
  (syntax-rules (lambda)
    [(_ name (lambda formals form1 form2 ...))
     (begin
       (define xname
         (fluid-let-syntax ([name (identifier-syntax xname)])
           (lambda formals form1 form2 ...)))
       (define-syntax name
         (lambda (x)
           (syntax-case x ()
             [_ (identifier? x) #'xname]
             [(_ arg (... ...))
              #'((fluid-let-syntax ([name (identifier-syntax xname)])
                   (lambda formals form1 form2 ...))
                 arg
                 (... ...))]))))]))
```

A `define-integrable` has the following form.

`(define-integrable` *name lambda-expression*`)`

A `define-integrable` form expands into a pair of definitions: a syntax definition of *name* and a variable definition of `xname`. The transformer for *name* converts apparent calls to *name* into direct calls to *lambda-expression*. Since the resulting forms are merely direct `lambda` applications (the equivalent of `let` expressions), the actual parameters are evaluated exactly once and before evaluation of the procedure's body, as required. All other references to *name* are replaced with references to `xname`. The definition of `xname` binds it to the value of *lambda-expression*. This allows the procedure to be used as a first-class value. Because

`xname` is introduced by the transformer, the binding for `xname` is not visible anywhere except where references to it are introduced by the transformer for *name*.

Within *lambda-expression*, wherever it appears, *name* is rebound to a transformer that expands all references into references to `xname`. The use of `fluid-let-syntax` for this purpose prevents indefinite expansion from indirect recursion among integrable procedures. This allows the procedure to be recursive without causing indefinite expansion. Nothing special is done by `define-integrable` to maintain lexical scoping, since lexical scoping is maintained automatically by the expander.

Chez Scheme integrate locally defined procedures automatically when it is appropriate to do so. It cannot integrate procedures defined at top-level, however, since code that assigns top-level variables can be introduced into the system (via `eval` or `load`) at any time. `define-integrable` can be used to force the integration of procedures bound at top-level, even if the integration of locally bound procedures is left to the compiler. It can also be used to force the integration of large procedures that the compiler would not normally integrate. (The `expand/optimize` procedure is useful for determining when integration does or does not take place.)

11.2. Syntax-Rules Transformers

Chez Scheme extends `syntax-rules` to permit clause to include fenders just like those allowed within `syntax-case` clauses.

`(syntax-rules (`*literal* `...)` *clause* `...)` **syntax**
returns: a transformer
libraries: `(chezscheme)`

Each *literal* must be an identifier other than an underscore (`_`) or ellipsis (`...`). Each clause must take the form below.

(*pattern template*)
(*pattern fender template*)

The first form is the only form supported by the Revised[6] Report.

11.3. Syntax-Case Transformers

Chez Scheme provides several procedures and syntactic forms that may be used to simplify the coding of certain syntactic abstractions.

`(syntax->list` *syntax-object*`)` **procedure**
returns: a list of syntax objects
libraries: `(chezscheme)`

This procedure takes a syntax object representing a list-structured form and returns a list

of syntax objects, each representing the corresponding subform of the input form.

`syntax->list` may be defined as follows.

```
(define syntax->list
  (lambda (ls)
    (syntax-case ls ()
      [() '()]
      [(x . r) (cons #'x (syntax->list #'r))])))
```

```
#'(a b c)  ⇒  #<syntax (a b c)>
(syntax->list #'(a b c))  ⇒  (#<syntax a> #<syntax b> #<syntax c>)
```

`syntax->list` is not required for list structures constructed from individual pattern variable values or sequences of pattern-variable values, since such structures are already lists. For example:

```
(list? (with-syntax ([x #'a] [y #'b] [z #'c]) #'(x y z)))  ⇒  #t
(list? (with-syntax ([(x ...) #'(a b c)]) #'(x ...)))  ⇒  #t
```

(syntax->vector *syntax-object*) **procedure**

returns: a list of syntax objects
libraries: `(chezscheme)`

This procedure takes a syntax object representing a vector-structured form and returns a list of syntax objects, each representing the corresponding subform of the input form.

`syntax->vector` may be defined as follows.

```
(define syntax->vector
  (lambda (v)
    (syntax-case v ()
      [#(x ...) (apply vector (syntax->list #'(x ...)))])))
```

```
#'#(a b c)  ⇒  #<syntax #(a b c)>
(syntax->vector #'#(a b c))  ⇒  #(#<syntax a> #<syntax b> #<syntax c>)
```

`syntax->vector` is not required for vector structures constructed from individual pattern variable values or sequences of pattern-variable values, since such structures are already vectors. For example:

```
(vector? (with-syntax ([x #'a] [y #'b] [z #'c]) #'#(x y z)))  ⇒  #t
(vector? (with-syntax ([(x ...) #'(a b c)]) #'#(x ...)))  ⇒  #t
```

(syntax-object->datum *obj*) **procedure**

returns: *obj* stripped of syntactic information
libraries: `(chezscheme)`

`syntax-object->datum` is identical to the Revised[6] Report `syntax->datum`.

`(datum` *template*`)` **syntax**
returns: see below
libraries: `(chezscheme)`

`(datum` *template*`)` is a convenient shorthand syntax for

```
(syntax->datum (syntax template))
```

datum may be defined simply as follows.

```
(define-syntax datum
  (syntax-rules ()
    [(_ t) (syntax->datum #'t)]))

(with-syntax ((a #'(a b c))) (datum a))  ⇒  (a b c)
```

`(datum->syntax-object` *template-identifier obj*`)` **procedure**
returns: a syntax object
libraries: `(chezscheme)`

`datum->syntax-object` is identical to the Revised[6] Report `datum->syntax`.

`(with-implicit (`id_0 id_1 `...)` $body_1$ $body_2$ `...)` **syntax**
returns: see below
libraries: `(chezscheme)`

This form abstracts over the common usage of `datum->syntax` for creating implicit identifiers (see above). The form

```
(with-implicit (id0 id1 ...)
  body1 body2 ...)
```

is equivalent to

```
(with-syntax ([id1 (datum->syntax #'id0 'id1)] ...)
  body1 body2 ...)
```

`with-implicit` can be defined simply as follows.

```
(define-syntax with-implicit
  (syntax-rules ()
    [(_ (tid id ...) b1 b2 ...)
     (with-syntax ([id (datum->syntax #'tid 'id)] ...)
       b1 b2 ...)]))
```

We can use `with-implicit` to simplify the (correct version of) `loop` above.

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      [(k e ...)
       (with-implicit (k break)
         #'(call-with-current-continuation
             (lambda (break)
               (let f () e ... (f)))))])))
```

(include *path*) **syntax**

returns: unspecified
libraries: `(chezscheme)`

path must be a string. `include` expands into a `begin` expression containing the forms found in the file named by *path*. For example, if the file `f-def.ss` contains `(define f (lambda () x))`, the expression

```
(let ([x "okay"])
  (include "f-def.ss")
  (f))
```

evaluates to `"okay"`. An include form is treated as a definition if it appears within a sequence of definitions and the forms on the file named by *path* are all definitions, as in the above example. If the file contains expressions instead, the `include` form is treated as an expression.

`include` may be defined portably as follows, although *Chez Scheme* uses an implementation-dependent definition that allows it to capture and maintain source information for included code.

```
(define-syntax include
  (lambda (x)
    (define read-file
      (lambda (fn k)
        (let ([p (open-input-file fn)])
          (let f ([x (read p)])
            (if (eof-object? x)
                (begin (close-input-port p) '())
                (cons (datum->syntax k x)
                      (f (read p))))))))
    (syntax-case x ()
      [(k filename)
       (let ([fn (datum filename)])
         (with-syntax ([(exp ...) (read-file fn #'k)])
           #'(begin exp ...)))])))
```

The definition of `include` uses `datum->syntax` to convert the objects read from the file into syntax objects in the proper lexical context, so that identifier references and definitions within those expressions are scoped where the `include` form appears.

In *Chez Scheme*'s implementation of `include`, the parameter `source-directories` (Section 12.5) determines the set of directories searched for source files not identified by absolute path names.

`(syntax-error` *obj string* `...)` **procedure**
returns: does not return
libraries: `(chezscheme)`

Syntax errors may be reported with `syntax-error`, which produces a message by concatenating *string* `...` and a printed representation of *obj*. If no string arguments are provided, the string `"invalid syntax"` is used instead. When *obj* is a syntax object, the syntax-object wrapper is stripped (as with `syntax->datum`) before the printed representation is created. If source file information is present in the syntax-object wrapper, `syntax-error` incorporates this information into the error message.

`syntax-case` and `syntax-rules` call `syntax-error` automatically if the input fails to match one of the clauses.

We can use `syntax-error` to precisely report the cause of the errors detected in the following definition of (unnamed) `let`.

```
(define-syntax let
  (lambda (x)
    (define check-ids!
      (lambda (ls)
        (unless (null? ls)
          (unless (identifier? (car ls))
            (syntax-error (car ls) "let cannot bind non-identifier"))
          (check-ids! (cdr ls)))))
    (define check-unique!
      (lambda (ls)
        (unless (null? ls)
          (let ([x (car ls)])
            (when (let mem? ([ls (cdr ls)])
                    (and (not (null? ls))
                         (or (bound-identifier=? x (car ls))
                             (mem? (cdr ls)))))
              (syntax-error x "let cannot bind two occurrences of")))
          (check-unique! (cdr ls)))))
    (syntax-case x ()
      [(_ ((i e) ...) b1 b2 ...)
       (begin
         (check-ids! #'(i ...))
         (check-unique! #'(i ...))
         #'((lambda (i ...) b1 b2 ...) e ...))])))
```

With this change, the expression

```
(let ([a 3] [a 4]) (+ a a))
```

produces the error message "let cannot bind two occurrences of `a`."

`(literal-identifier=?` *identifier*$_1$ *identifier*$_2$`)` **procedure**
returns: see below
libraries: `(chezscheme)`

This procedure is identical to the Revised^6 Report `free-identifier=?`, and is provided for backward compatibility only.

11.4. Compile-time Values and Properties

When defining sets of dependent macros, it is often convenient to attach information to identifiers in the same *compile time environment* that the expander uses to record information about variables, keywords, module names, etc. For example, a record-type definition macro, like `define-record-type`, might need to attach information to the record-type name in the compile-time environment for use in handling child record-type definitions.

Chez Scheme provides two mechanisms for attaching information to identifiers in the compile-time environment: compile-time values and compile-time properties. A compile-time value is a kind of transformer that can be associated with an identifier via `define-syntax`, `let-syntax`, `letrec-syntax`, and `fluid-let-syntax`. When an identifier is associated with a compile-time value, it cannot also have any other meaning, and an attempt to reference it as an ordinary identifier results in a syntax error. A compile-time property, on the other hand, is maintained alongside an existing binding, providing additional information about the binding. Properties are ignored when ordinary references to an identifier occur.

The mechanisms used by a macro to obtain compile-time values and properties are similar. In both cases, the macro's transformer returns a procedure *p* rather than a syntax object. The expander invokes *p* with one argument, an environment-lookup procedure *lookup*, which *p* can then use to obtain compile-time values and properties for one or more identifiers before it constructs the macro's final output. *lookup* accepts one or two identifier arguments. With one argument, *id*, *lookup* returns the compile-time value of *id*, or `#f` if *id* has no compile-time value. With two arguments, *id* and *key*, *lookup* returns the value of *id*'s *key* property, or `#f` if *id* has no *key* property.

`(make-compile-time-value` *obj*`)` **procedure**
returns: a compile-time value
libraries: `(chezscheme)`

A compile time value is a kind of transformer with which a keyword may be associated by any of the keyword binding constructs, e.g., `define-syntax` or `let-syntax`. The transformer encapsulates the supplied *obj*. The encapsulated object may be retrieved as described above.

The following example illustrates how this feature might be used to define a simple syntactic record-definition mechanism where the record type descriptor is generated at expansion time.

```
(define-syntax drt
  (lambda (x)
    (define construct-name
      (lambda (template-identifier . args)
        (datum->syntax template-identifier
          (string->symbol
            (apply string-append
              (map (lambda (x)
                     (if (string? x)
                         x
                         (symbol->string (syntax->datum x))))
                   args))))))
    (define do-drt
      (lambda (rname fname* prtd)
        (with-syntax ([rname rname]
                      [rtd (make-record-type-descriptor
                             (syntax->datum rname) prtd #f #f #f
                             (list->vector
                               (map (lambda (fname)
                                      `(immutable ,(syntax->datum fname)))
                                    fname*)))]
                      [make-rname (construct-name rname "make-" rname)]
                      [rname? (construct-name rname rname "?")]
                      [(rname-fname ...)
                       (map (lambda (fname)
                              (construct-name fname rname "-" fname))
                            fname*)]
                      [(i ...) (enumerate fname*)])
          #'(begin
              (define-syntax rname (make-compile-time-value 'rtd))
              (define rcd (make-record-constructor-descriptor 'rtd #f #f))
              (define make-rname (record-constructor rcd))
              (define rname? (record-predicate 'rtd))
              (define rname-fname (record-accessor 'rtd i))
              ...))))
    (syntax-case x (parent)
      [(_ rname (fname ...))
       (for-all identifier? #'(rname fname ...))
       (do-drt #'rname #'(fname ...) #f)]
      [(_ rname pname (fname ...))
       (for-all identifier? #'(rname pname fname ...))
       (lambda (lookup)
         (let ([prtd (lookup #'pname)])
           (unless (record-type-descriptor? prtd)
             (syntax-error #'pname "unrecognized parent record type"))
           (do-drt #'rname #'(fname ...) prtd)))])))
```

```
(drt prec (x y))
(drt crec prec (z))
(define r (make-crec 1 2 3))
(prec? r)  ⇒  #t
(prec-x r)  ⇒  1
(crec-z r)  ⇒  3
prec  ⇒  exception: invalid syntax prec
```

`(define-property` *id key expr*`)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

A `define-property` form attaches a property to an existing identifier binding without disturbing the existing meaning of the identifier in the scope of that binding. It is typically used by one macro to record information about a binding for use by another macro. Both *id* and *key* must be identifiers. The expression *expr* is evaluated when the `define-property` form is expanded, and a new property associating *key* with the value of *expr* is attached to the existing binding of *id*, which must have a visible local or top-level binding.

`define-property` is a definition and can appear anywhere other definitions can appear. The scope of a property introduced by `define-property` is the entire body in which the `define-property` form appears or global if it appears at top level, except where it is replaced by a property for the same *id* and *key* or where the binding to which it is attached is shadowed. Any number of properties can be attached to the same binding with different keys. Attaching a new property with the same name as an property already attached to a binding shadows the existing property with the new property.

The following example defines a macro, `get-info`, that retrieves the `info` property of a binding, defines the variable `x`, attaches an `info` property to the binding of `x`, retrieves the property via `get-info`, references `x` to show that its normal binding is still intact, and uses `get-info` again within the scope of a different binding of `x` to show that the properties are shadowed as well as the outer binding of `x`.

```
(define info)
(define-syntax get-info
  (lambda (x)
    (lambda (lookup)
      (syntax-case x ()
        [(_ q)
         (let ([info-value (lookup #'q #'info)])
           #`'#,(datum->syntax #'* info-value))])))))
(define x "x-value")
(define-property x info "x-info")
(get-info x)  ⇒  "x-info"
x  ⇒  "x-value"
(let ([x "inner-x-value"]) (get-info x))  ⇒  #f
```

For debugging, it is often useful to have a form that retrieves an arbitrary property, given an identifier and a key. The `get-property` macro below does just that.

```
(define-syntax get-property
  (lambda (x)
    (lambda (r)
      (syntax-case x ()
        [(_ id key)
         #‘’#,(datum->syntax #’* (r #’id #’key))]))))
(get-property x info)  ⇒  "x-info"
```

The bindings for both identifiers must be visible where `get-property` is used.

The version of `drt` defined below is like the one defined using `make-compile-time-value` above, except that it defines the record name as a macro that raises an exception with a more descriptive message, while attaching the record type descriptor to the binding as a separate property. The variable `drt-key` defined along with `drt` is used only as the key for the property that `drt` attaches to a record name. Both `drt-key` and `drt` are defined within a module that exports only the latter, ensuring that the properties used by `drt` cannot be accessed or forged.

```
(library (drt) (export drt) (import (chezscheme))
  (define drt-key)
  (define-syntax drt
    (lambda (x)
      (define construct-name
        (lambda (template-identifier . args)
          (datum->syntax template-identifier
            (string->symbol
              (apply string-append
                (map (lambda (x)
                       (if (string? x)
                           x
                           (symbol->string (syntax->datum x))))
                     args))))))
      (define do-drt
        (lambda (rname fname* prtd)
          (with-syntax ([rname rname]
                        [rtd (make-record-type-descriptor
                               (syntax->datum rname) prtd #f #f #f
                               (list->vector
                                 (map (lambda (fname)
                                        ‘(immutable ,(syntax->datum fname)))
                                      fname*)))]
                        [make-rname (construct-name rname "make-" rname)]
                        [rname? (construct-name rname rname "?")]
                        [(rname-fname ...)
                         (map (lambda (fname)
                                (construct-name fname rname "-" fname))
                              fname*)]
                        [(i ...) (enumerate fname*)])
            #’(begin
                (define-syntax rname
```

```
                (lambda (x)
                  (syntax-error x "invalid use of record name")))
              (define rcd (make-record-constructor-descriptor 'rtd #f #f))
              (define-property rname drt-key 'rtd)
              (define make-rname (record-constructor rcd))
              (define rname? (record-predicate 'rtd))
              (define rname-fname (record-accessor 'rtd i))
              ...))))
      (syntax-case x (parent)
        [(_ rname (fname ...))
         (for-all identifier? #'(rname fname ...))
         (do-drt #'rname #'(fname ...) #f)]
        [(_ rname pname (fname ...))
         (for-all identifier? #'(rname pname fname ...))
         (lambda (lookup)
           (let ([prtd (lookup #'pname #'drt-key)])
             (unless prtd
               (syntax-error #'pname "unrecognized parent record type"))
             (do-drt #'rname #'(fname ...) prtd)))])))

(import (drt))
(drt prec (x y))
(drt crec prec (z))
(define r (make-crec 1 2 3))
(prec? r) ⇒ #t
(prec-x r) ⇒ 1
(crec-z r) ⇒ 3
prec ⇒ exception: invalid use of record name prec
```

11.5. Modules

Modules are used to help organize programs into separate parts that interact cleanly via declared interfaces. Although modular programming is typically used to facilitate the development of large programs possibly written by many individuals, it may also be used in *Chez Scheme* at a "micro-modular" level, since *Chez Scheme* module and import forms are definitions and may appear anywhere any other kind of definition may appear, including within a `lambda` body or other local scope.

Modules control visibility of bindings and can be viewed as extending lexical scoping to allow more precise control over where bindings are or are not visible. Modules export identifier bindings, i.e., variable bindings, keyword bindings, or module name bindings. Modules may be *named* or *anonymous*. Bindings exported from a named module may be made visible via an import form wherever the module's name is visible. Bindings exported from an anonymous module are implicitly imported where the module form appears. Anonymous modules are useful for hiding some of a set of bindings while allowing the remaining bindings in the set to be visible.

Some of the text and examples given in this section are adapted from the paper "Extending

the scope of syntactic abstraction" [32], which describes modules and their implementation in more detail.

(module *name interface defn ... init ...***)** **syntax**
(module *interface defn ... init ...***)** **syntax**
returns: unspecified
libraries: `(chezscheme)`

name is an identifier, *defn* ... are definitions, and *init* ... are expressions. *interface* is a list of exports (*export* ...), where each *export* is either an identifier *identifier* or of the form (*identifier export* ...).

The first syntax for `module` establishes a named scope that encapsulates a set of identifier bindings. The exported bindings may be made visible via `import` or `import-only` (Section 10.4) anywhere the module name is visible. The second syntax for `module` introduces an anonymous module whose bindings are implicitly imported (as if by `import` of a hidden module name) where the module form appears.

A module consists of a (possibly empty) set of definitions and a (possibly empty) sequence of initialization expressions. The identifiers defined within a module are visible within the body of the module and, if exported, within the scope of an import for the module. Each identifier listed in a module's interface must be defined within or imported into that module. A `module` form is a definition and can appear anywhere other definitions can appear, including at the top level of a program, nested within the bodies of `lambda` expressions, nested within `library` and top-level program forms, and nested within other modules. Also, because module names are scoped like other identifiers, modules and libraries may export module names as well as variables and keywords.

When an interface contains an export of the form (*identifier export* ...), only *identifier* is visible in the importing context. The identifiers within *export* ... are *indirect imports*, as if declared via an `indirect-export` form (Section 10.4).

Module names occupy the same namespace as other identifiers and follow the same scoping rules. Unless exported, identifiers defined within a module are visible only within that module.

Expressions within a module can reference identifiers bound outside of the module.

```
(let ([x 3])
  (module m (plusx)
    (define plusx (lambda (y) (+ x y))))
  (import m)
  (let ([x 4])
    (plusx 5)))  ⇒  8
```

Similarly, `import` does not prevent access to identifiers that are visible where the import form appears, except for those variables shadowed by the imported identifiers.

```
(module m (y) (define y 'm-y))
(let ([x 'local-x] [y 'local-y])
  (import m)
  (list x y))  ⇒  (local-x m-y)
```

On the other hand, use of `import-only` within a module establishes an isolated scope in which the only visible identifiers are those exported by the imported module.

```
(module m (y) (define y 'm-y))
(let ([x 'local-x] [y 'local-y])
  (import-only m)
  x)  ⇒  Error: x is not visible
```

This is sometimes desirable for static verification that no identifiers are used except those explicitly imported into a module or local scope.

Unless a module imported via `import-only` exports `import` or `import-only` and the name of at least one module, subsequent imports within the scope of the `import-only` form are not possible. To create an isolated scope containing the exports of more than one module without making `import` or `import-only` visible, all of the modules to be imported must be listed in the same `import-only` form.

Another solution is to create a single module that contains the exports of each of the other modules.

```
(module m2 (y) (define y 'y))
(module m1 (x) (define x 'x))
(module mega-module (cons x y)
  (import m1)
  (import m2)
  (import scheme))
(let ([y 3])
  (import-only mega-module)
  (cons x y))  ⇒  (x . y)
```

Before it is compiled, a source program is translated into a core language program containing no syntactic abstractions, syntactic definitions, library definitions, module definitions, or import forms. Translation is performed by a *syntax expander* that processes the forms in the source program via recursive descent.

A `define-syntax` form associates a keyword with a transformer in a translation-time environment. When the expander encounters a keyword, it invokes the associated transformer and reprocesses the resulting form. A `module` form associates a module name with an interface. When the expander encounters an `import` form, it extracts the corresponding module interface from the translation-time environment and makes the exported bindings visible in the scope where the `import` form appears.

Internal definitions and definitions within a `module` body are processed from left to right so that a module's definition and import may appear within the same sequence of definitions. Expressions appearing within a body and the right-hand sides of variable definitions, how-

ever, are translated only after the entire set of definitions has been processed, allowing full mutual recursion among variable and syntactic definitions.

Module and import forms affect only the visibility of identifiers in the source program, not their meanings. In particular, variables are bound to locations whether defined within or outside of a module, and `import` does not introduce new locations. Local variables are renamed as necessary to preserve the scoping relationships established by both modules and syntactic abstractions. Thus, the expression:

```
(let ([x 1])
  (module m (x setter)
    (define-syntax x (identifier-syntax z))
    (define setter (lambda (x) (set! z x)))
    (define z 5))
  (let ([y x] [z 0])
    (import m)
    (setter 3)
    (+ x y z)))  ⇒  4
```

is equivalent to the following program in which identifiers have been consistently renamed as indicated by subscripts.

```
(let ([x0 1])
  (define-syntax x1 (identifier-syntax z1))
  (define setter1 (lambda (x2) (set! z1 x2)))
  (define z1 5)
  (let ([y3 x0] [z3 0])
    (setter1 3)
    (+ x1 y3 z3)))
```

Definitions within a top-level `begin`, `lambda`, top-level program, `library`, or `module` body are processed from left to right by the expander at expand time, and the variable definitions are evaluated from left-to-right at run time. Initialization expressions appearing within a `module` body are evaluated in sequence after the evaluation of the variable definitions.

Mutually recursive modules can be defined in several ways. In the following program, `a` and `b` are mutually recursive modules exported by an anonymous module whose local scope is used to statically link the two. For example, the free variable `y` within module `a` refers to the binding for `y`, provided by importing `b`, in the enclosing module.

```
(module (a b)
  (module a (x) (define x (lambda () y)))
  (module b (y) (define y (lambda () x)))
  (import a)
  (import b))
```

The following syntactic abstraction generalizes this pattern to permit the definition of multiple mutually recursive modules.

```
(define-syntax rec-modules
  (syntax-rules (module)
```

```
    [(_ (module m (id ...) form ...) ...)
     (module (m ...)
       (module m (id ...) form ...) ...
       (import m) ...)]))
```

Because a module can re-export imported bindings, it is quite easy to provide multiple views on a single module, as `s` and `t` provide for `r` below, or to combine several modules into a compound, as `r` does.

```
(module p (x y)
  (define x 1) (define y 2))
(module q (y z)
  (define y 3) (define z 4))
(module r (a b c d)
  (import* p (a x) (b y))
  (import* q (c y) (d z)))
(module s (a c) (import r))
(module t (b d) (import r))
```

To allow interfaces to be separated from implementations, the following syntactic abstractions support the definition and use of named interfaces.

```
(define-syntax define-interface
  (syntax-rules ()
    [(_ name (export ...))
     (define-syntax name
       (lambda (x)
         (syntax-case x ()
           [(_ n defs)
            (with-implicit (n export ...)
              #'(module n (export ...) .
                  defs))]))))]))

(define-syntax define-module
  (syntax-rules ()
    [(_ name interface defn ...)
     (interface name (defn ...))]))
```

`define-interface` creates an interface macro that, given a module name and a list of definitions, expands into a module definition with a concrete interface.

`with-implicit` is used to ensure that the introduced `export` identifiers are visible in the same scope as the name of the module in the `define-module` form.

`define-interface` and `define-module` can be used as follows.

```
(define-interface simple (a b))
(define-module m simple
  (define-syntax a (identifier-syntax 1))
  (define b (lambda () c))
  (define c 2))
(let () (import m) (+ a (b)))  ⇒  3
```

The abstract module facility defined below allows a module interface to be satisfied incrementally when module forms are evaluated. This permits flexibility in the separation between the interface and implementation, supports separate compilation of mutually recursive modules, and permits redefinition of module implementations.

```
(define-syntax abstract-module
  (syntax-rules ()
    [(_ name (ex ...) (kwd ...) defn ...)
     (module name (ex ... kwd ...)
       (declare ex) ...
       defn ...)]))

(define-syntax implement
  (syntax-rules ()
    [(_ name form ...)
     (module () (import name) form ...)]))
```

Within an `abstract-module` form, each of the exports in the list *ex* `...` must be variables. The values of these variables are supplied by one or more separate `implement` forms. Since keyword bindings must be present at compile time, they cannot be satisfied incrementally and are instead listed as separate exports and defined within the abstract module.

Within an `implement` form, the sequence of forms *form* `...` is a sequence of zero or more definitions followed by a sequence of zero or more expressions. Since the module used in the expansion of `implement` does not export anything, the definitions are all local to the `implement` form. The expressions may be arbitrary expressions, but should include one `satisfy` form for each variable whose definition is supplied by the `implement` form. A `satisfy` form has the syntax

`(satisfy` *variable expr*`)`

`declare` and `satisfy` may simply be the equivalents of `define` and `set!`.

```
(define-syntax declare (identifier-syntax define))
(define-syntax satisfy (identifier-syntax set!))
```

Alternatively, `declare` can initialize the declared variable to the value of a flag known only to `declare` and `satisfy`, and `satisfy` can verify that this flag is still present to insure that only one attempt to satisfy the value of a given identifier is made.

```
(module ((declare cookie) (satisfy cookie))
  (define cookie "chocolate chip")
  (define-syntax declare
    (syntax-rules () [(_ var) (define var cookie)]))
  (define-syntax satisfy
    (syntax-rules ()
      [(_ var exp)
       (if (eq? var cookie)
           (set! var exp)
           (assertion-violationf 'satisfy
             "value of variable ~s has already been satisfied"
```

```
            'var))])))
```

Using `abstract-module` and `implement`, we can define mutually recursive and separately compilable modules as follows.

```
(abstract-module e (even?) (pred)
  (define-syntax pred
    (syntax-rules () [(_ exp) (- exp 1)])))

(abstract-module o (odd?) ())

(implement e
  (import o)
  (satisfy even?
    (lambda (x)
      (or (zero? x) (odd? (pred x))))))

(implement o
  (import e)
  (satisfy odd?
    (lambda (x) (not (even? x)))))

(let () (import-only e) (even? 38))  ⇒  #t
```

`only`	**syntax**
`except`	**syntax**
`add-prefix`	**syntax**
`drop-prefix`	**syntax**
`rename`	**syntax**
`alias`	**syntax**

libraries: `(chezscheme)`

These identifiers are auxiliary keywords for `import` and `import-only`. It is a syntax violation to reference these identifiers except in contexts where they are recognized as auxiliary keywords.

11.6. Standalone import and export forms

The local import and export forms described in Section 10.4 can be used equally well for and within modules.

11.7. Built-in Modules

Five modules are built-in to *Chez Scheme*: `scheme`, `r5rs`, `r5rs-syntax`, `ieee`, and `$system`. Each module is immutable, i.e., the exported bindings cannot be altered.

`scheme` **module**
libraries: `(chezscheme)`

`scheme` contains all user-visible top-level bindings (variables, keywords, and module names) built into *Chez Scheme*.

`r5rs` **module**
libraries: `(chezscheme)`

`r5rs` contains all top-level bindings (variables and keywords) defined in the Revised[5] Report on Scheme. The bindings exported from `r5rs` are precisely those that are available within an expression evaluated via `eval` with the environment specifier returned by `scheme-report-environment`.

`r5rs-syntax` **module**
libraries: `(chezscheme)`

`r5rs-syntax` contains all top-level keyword bindings defined in the Revised[5] Report on Scheme. The bindings exported from `r5rs-syntax` are precisely those that are available within an expression evaluated via `eval` with the environment specifier returned by `null-environment`.

`ieee` **module**
libraries: `(chezscheme)`

`ieee` contains all top-level bindings (variables and keywords) defined in the ANSI/IEEE standard for Scheme. The bindings exported from `ieee` are precisely those that are available within an expression evaluated via `eval` with the environment specifier returned by `ieee-environment`.

`$system` **module**
libraries: `(chezscheme)`

`$system` contains all user-visible top-level bindings built into *Chez Scheme* along with various undocumented system bindings.

11.8. Meta Definitions

`(meta .` *definition*`)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

The `meta` keyword is actually a prefix that can be placed in front of any definition keyword, e.g.,

```
(meta define x 3)
```

It tells the expander that any variable definition resulting from the definition is to be an expand-time definition available only to the right-hand sides of other meta definitions and, most importantly, transformer expressions. It is used to define expand-time helpers and other information for use by one or more `syntax-case` transformers.

```
(module M (helper1 a b)
  (meta define helper1
    (lambda (---)
      ---))
  (meta define helper2
    (lambda (---)
      --- (helper2 ---) ---))
  (define-syntax a
    (lambda (x)
      --- (helper1 ---) ---))
  (define-syntax b
    (lambda (x)
      --- (helper1 ---) ---
      --- (helper2 ---) ---)))
```

The right-hand-side expressions of a syntax definition or meta definition can refer only to identifiers whose values are already available in the compile-time environment. Because of the left-to-right expansion order for `library`, `module`, `lambda`, and similar bodies, this implies a semantics similar to `let*` for a sequence of meta definitions, in which each right-hand side can refer only to the variables defined earlier in the sequence. An exception is that the right-hand side of a meta definition can refer to its own name as long as the reference is not evaluated until after the value of the expression has been computed. This permits meta definitions to be self-recursive but not mutually recursive. The right-hand side of a meta definition can, however, build syntax objects containing occurrences of any identifiers defined in the body in which the meta definition appears.

Meta definitions propagate through macro expansion, so one can write, for example:

```
(module (a)
  (meta define-record foo (x))
  (define-syntax a
    (let ([q (make-foo #''q)])
      (lambda (x) (foo-x q)))))
a ⇒ q
```

where define-record is a macro that expands into a set of defines.

It is also sometimes convenient to write

```
(meta begin defn ...)
```

or

```
(meta module {exports} defn ...)
```

or

`(meta include "`*path*`")`

to create groups of meta bindings.

11.9. Conditional expansion

Expansion-time decisions can be made via `meta-cond`, which is similar to `cond` but evaluates the test expressions at expansion time and can be used in contexts where definitions are expected as well as in expression contexts.

`(meta-cond` *clause*$_1$ *clause*$_2$ `...)` **syntax**
returns: see below
libraries: `(chezscheme)`

Each *clause* but the last must take the form:

`(`*test* *expr*$_1$ *expr*$_2$ `...)`

The last may take the same form or be an `else` clause of the form:

`(`*else* *expr*$_1$ *expr*$_2$ `...)`

During expansion, the *test* expressions are evaluated in order until one evaluates to a true value or until all of the tests have been evaluated. If a *test* evaluates to a true value, the `meta-cond` form expands to a `begin` form containing the corresponding expressions *expr*$_1$ *expr*$_2$ .... If no *test* evaluates to a true value and an `else` clause is present, the `meta-cond` form expands to a `begin` form containing the expressions *expr*$_1$ *expr*$_2$ ... from the `else` clause. Otherwise the `meta-cond` expression expands into a call to the `void` procedure.

`meta-cond` might be defined as follows.

```
(define-syntax meta-cond
  (syntax-rules ()
    [(_ [a0 a1 a2 ...] [b0 b1 b2 ...] ...)
     (let-syntax ([expr (cond
                          [a0 (identifier-syntax (begin a1 a2 ...))]
                          [b0 (identifier-syntax (begin b1 b2 ...))]
                          ...)])
       expr)]))
```

`meta-cond` is used to choose, at expansion time, from among a set of possible forms. For example, one might have safe (error-checking) and unsafe (non-error-checking) versions of a procedure and decide which to call based on the compile-time optimization level, as shown below.

```
(meta-cond
  [(= (optimize-level) 3) (unsafe-frob x)]
  [else (safe-frob x)])
```

11.10. Aliases

`(alias` id_1 id_2`)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

`alias` is a definition and can appear anywhere other definitions can appear. It is used to transfer the binding from one identifier to another.

```
(let ([x 3]) (alias y x) (set! y 4) (list x y))  ⇒  (4 4)
```

```
(module lisp (if)
  (module (scheme:if)
    (import scheme)
    (alias scheme:if if))
  (define-syntax if
    (syntax-rules ()
      [(_ e_1 e_2 e_3)
       (scheme:if (not (memq e_1 '(#f ()))) e_2 e_3)])))
(define (length ls)
  (import lisp)
  (if ls (+ (length (cdr ls)) 1) 0))
(length '(a b c))  ⇒  3
```

Because of left-to-right expansion order, aliases should appear after the definition of the right-hand-side identifier, e.g.:

```
(let ()
  (import-only (chezscheme))
  (define y 3)
  (alias x y)
  x)  ⇒  3
```

rather than:

```
(let ()
  (import-only (chezscheme))
  (alias x y)
  (define y 3)
  x)  ⇒  exception: unbound identifier
```

11.11. Annotations

When source code is read from a file by `load`, `compile-file`, or variants of these, such as `load-library`, the reader attaches *annotations* to each object read from the file. These annotations identify the file and the position of the object within the file. Annotations are tracked through the compilation process and associated with compiled code at run time. The expander and compiler use the annotations to produce syntax errors and compiler warnings that identify the location of the offending form, and the inspector uses them to identify the locations of calls and procedure definitions. The compiler and run time also use annotations to associate source positions with profile counts.

While these annotations are usually maintained "behind the scenes," the programmer can manipulate them directly via a set of routines for creating and accessing annotations.

Annotations are values of a type distinct from other types and have four components: an expression, possibly with annotated subexpressions, a *source object*, a stripped version of the expression, and usage options. Annotations can be created via `make-annotation`, which has three required arguments corresponding to the first three components and an optional fourth argument corresponding to the fourth component. The second argument must be a source object, and the third argument should be a stripped version of the first argument, i.e., equivalent to the first argument with each annotation replaced by its expression component. An annotation is essentially equivalent to its stripped component as a representation of source code, with the source information attached and available to the expander or evaluator. The optional fourth argument, if present, must be an enumeration set over the symbols `debug` and `profile` and defaults to an enumeration set containing both `debug` and `profile`.

Annotations marked `debug` are used for compile-time error reporting and run-time error reporting and inspection; annotations marked `profile` are used for profiling. Annotations created by the Scheme reader are always marked both `debug` and `profile`, but other readers and parsers might choose to mark some annotations only `debug` or only `profile`. In particular, it might be useful to annotate multiple expressions in the output of a parser with the same source object for debugging purposes and mark only one of them `profile` to avoid duplicate counts. It might also be useful to mark no expressions `profile` and instead introduce explicit `profile` forms (Section 12.7) to identify the set of source locations to be profiled.

Source objects are also values of a type distinct from other types and also have three or five components: a *source-file descriptor* (sfd), a beginning file position (bfp), an ending file position (efp), an optional beginning line, and an optional beginning column. The sfd identifies the file from which an expression is read and the bfp and efp identify the range of character positions occupied by the object in the file, with the bfp being inclusive and the efp being exclusive. The line and column are either both numbers or both not present. A source object can be created via `make-source-object`, which takes either three or five arguments corresponding to these components. The first argument must be a source-file descriptor, the second and third must be nonnegative exact integers, the second must not be greater than the third, and the fourth and fifth (if provided) must be positive exact integers.

Source-file descriptors are also values of a type distinct from all other types and have two components: the file's path, represented by a string, and a checksum, represented by a number. The path might or might not be an absolute path depending on how the file's path was specified when the source-file descriptor was created. The checksum is computed based on the file's length and contents when the file is created and checked by tools that look for the source file to make sure that the proper file has been found and has not been modified. Source-file descriptors can be created with `make-source-file-descriptor`, which accepts two arguments: a string naming the path and a binary input port, along with an optional third boolean argument, *reset?*, which defaults to false. `make-source-file-descriptor` computes a checksum based on the contents of the port, starting at its current position. It resets the port, using `set-port-position!`, after computing the checksum if *reset?* is true; otherwise, it leaves the port at end-of-file.

The procedures that create, check for, and access annotations, source objects, and source-file descriptors are summarized below and described in more detail later in this section.

`(make-annotation` *obj source-object obj*`)` → *annotation*
`(annotation?` *obj*`)` → *boolean*
`(annotation-expression` *annotation*`)` → *obj*
`(annotation-source` *annotation*`)` → *source-object*
`(annotation-stripped` *annotation*`)` → *obj*

`(make-source-object` *sfd uint uint*`)` → *source-object*
`(make-source-object` *sfd uint uint uint uint*`)` → *source-object*
`(source-object?` *obj*`)` → *boolean*
`(source-object-sfd` *source-object*`)` → *sfd*
`(source-object-bfp` *source-object*`)` → *uint*
`(source-object-efp` *source-object*`)` → *uint*
`(source-object-line` *source-object*`)` → *uint* `or #f`
`(source-object-column` *source-object*`)` → *uint* `or #f`

`(make-source-file-descriptor` *string binary-input-port*`)` → *sfd*
`(make-source-file-descriptor` *string binary-input-port reset?*`)` → *sfd*
`(source-file-descriptor?` *obj*`)` → *boolean*
`(source-file-descriptor-checksum` *sfd*`)` → *obj*
`(source-file-descriptor-path` *sfd*`)` → *obj*

A program might open a source file with `open-file-input-port`, create an sfd using `make-source-file-descriptor`, create a textual port from the binary port using transcoded-port, and create source objects and annotations for each of the objects it reads from the file. If a custom reader is not required, the Scheme reader can be used to read annotations via the `get-datum/annotations` procedure:

`(get-datum/annotations` *textual-input-port sfd uint*`)` → *obj*`,` *uint*

`get-datum/annotations` is like `get-datum` but instead of returning a plain datum, it returns an annotation encapsulating a datum (possibly with nested annotations), a source object, and the plain (stripped) datum. It also returns a second value, the position of the first character beyond the object in the file. Character positions are accepted and returned by `get-datum/annotations` so that the textual port need not support `port-position` and

need not report positions in characters if it does support `port-position`. (Positions are usually reported in bytes.) The bfp and efp positions recorded in the annotations returned by `get-datum/annotations` are correct only if the positions supplied to it are correct.

Once read, an annotation can be passed to the expander, interpreter, or compiler. The procedures `eval`, `expand`, `interpret`, and `compile` all accept annotated or unannotated input.

Two additional procedures complete the set of annotation-related primitives:

`(open-source-file` *sfd*`)` → `#f or` *port*
`(syntax->annotation` *obj*`)` → `#f or` *annotation*

`open-source-file` attempts to locate and open the source file identified by *sfd*. It returns a textual input port, positioned at the beginning of the file, if successful, and `#f` otherwise.

`syntax->annotation` accepts a syntax object. If the syntax object's expression is annotated, it returns the annotation; otherwise, it returns `#f`. It can be used by a macro to extract source information, when available, from an input form.

The procedure `datum->syntax` accepts either an annotated or unannotated input datum.

`(make-annotation` *obj source-object stripped-obj*`)` **procedure**
`(make-annotation` *obj source-object stripped-obj options*`)` **procedure**
returns: an annotation
libraries: `(chezscheme)`

The annotation is formed with *obj* as its expression component, *source-object* as its source-object component, and *stripped-obj* as its stripped component. *obj* should represent an expression, possibly with embedded annotations. *stripped-obj* should be a stripped version of *obj*, i.e., equivalent to *obj* with each annotation replaced by its expression component. *options*, if present must be an enumeration set over the symbols `debug` and `profile`, and defaults to an enumeration set containing both `debug` and `profile`. Annotations marked `debug` are used for compile-time error reporting and run-time error reporting and inspection; annotations marked `profile` are used for profiling.

`(annotation?` *obj*`)` **procedure**
returns: `#t` if *obj* is an annotation, otherwise `#f`
libraries: `(chezscheme)`

`(annotation-expression` *annotation*`)` **procedure**
returns: the expression component of *annotation*
libraries: `(chezscheme)`

`(annotation-source` *annotation*`)` **procedure**
returns: the source-object component of *annotation*
libraries: `(chezscheme)`

(annotation-stripped *annotation*) **procedure**
returns: the stripped component of *annotation*
libraries: (chezscheme)

(annotation-options *annotation*) **procedure**
returns: the options enumeration set of *annotation*
libraries: (chezscheme)

(make-source-object *sfd bfp efp*) **procedure**
(make-source-object *sfd bfp efp line column*) **procedure**
returns: a source-object
libraries: (chezscheme)

sfd must be a source-file descriptor. *bfp* and *efp* must be exact nonnegative integers, and *bfp* should not be greater than *efp*. *line* and *column* must be exact positive integers.

(source-object? *obj*) **procedure**
returns: `#t` if *obj* is a source object, otherwise `#f`
libraries: (chezscheme)

(source-object-sfd *source-object*) **procedure**
returns: the sfd component of *source-object*
libraries: (chezscheme)

(source-object-bfp *source-object*) **procedure**
returns: the bfp component of *source-object*
libraries: (chezscheme)

(source-object-efp *source-object*) **procedure**
returns: the efp component of *source-object*
libraries: (chezscheme)

(source-object-line *source-object*) **procedure**
returns: the line component of *source-object* if present, otherwise `#f`
libraries: (chezscheme)

(source-object-column *source-object*) **procedure**
returns: the column component of *source-object* if present, otherwise `#f`
libraries: (chezscheme)

`current-make-source-object` **thread parameter**
libraries: `(chezscheme)`

`current-make-source-object` is used by the reader to construct a source object for an annotation. `current-make-source-object` is initially bound to `make-source-object`, and the reader always calls the function bound to the paramater with three arguments.

Adjust this parameter to, for example, eagerly convert a position integer to a file-position object, instead of delaying the conversion to `locate-source`.

`(make-source-file-descriptor` *string binary-input-port*`)` **procedure**
`(make-source-file-descriptor` *string binary-input-port reset?*`)` **procedure**
returns: a source-file descriptor
libraries: `(chezscheme)`

To compute the checksum encapsulated in the source-file descriptor, this procedure must read all of the data from *binary-input-port*. If *reset?* is present and `#t`, the port is reset to its original position, as if via `port-position`. Otherwise, it is left pointing at end-of-file.

`(source-file-descriptor?` *obj*`)` **procedure**
returns: `#t` if *obj* is a source-file descriptor, otherwise `#f`
libraries: `(chezscheme)`

`(source-file-descriptor-checksum` *sfd*`)` **procedure**
returns: the checksum component of *sfd*
libraries: `(chezscheme)`

`(source-file-descriptor-path` *sfd*`)` **procedure**
returns: the path component of *sfd*
libraries: `(chezscheme)`

sfd must be a source-file descriptor.

`(source-file-descriptor` *path checksum*`)` **procedure**
returns: a new source-file-descriptor
libraries: `(chezscheme)`

path must be a string, and *checksum* must be an exact nonnegative integer. This procedure can be used to construct custom source-file descriptors or to reconstitute source-file descriptors from the *path* and *checksum* components.

`(annotation-option-set` *symbol ...*`)` **syntax**
returns: an annotation-options enumeration set
libraries: `(chezscheme)`

Annotation-options enumeration sets may be passed to `make-annotation` to control whether

the annotation is used for debugging, profiling, both, or neither. Accordingly, each *symbol* must be either *debug* or `profile`.

`(syntax->annotation` *obj*`)` **procedure**
returns: an annotation or `#f`
libraries: `(chezscheme)`

If *obj* is an annotation or syntax-object encapsulating an annotation, the annotation is returned.

`(get-datum/annotations` *textual-input-port sfd bfp*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

sfd must be a source-file descriptor. *bfd* must be an exact nonnegative integer and should be the character position of the next character to be read from *textual-input-port*.

This procedure returns two values: an annotated object and an ending file position. In most cases, *bfp* should be 0 for the first call to `get-datum/annotation` at the start of a file, and it should be the second return value of the preceding call to `get-datum/annotation` for each subsequent call. This protocol is necessary to handle files containing multiple-byte characters, since file positions do not necessarily correspond to character positions.

`(open-source-file` *sfd*`)` **procedure**
returns: a port or `#f`
libraries: `(chezscheme)`

sfd must be a source-file descriptor. This procedure attempts to locate and open the source file identified by *sfd*. It returns a textual input port, positioned at the beginning of the file, if successful, and `#f` otherwise. It can fail even if a file with the correct name exists in one of the source directories when the file's checksum does not match the checksum recorded in *sfd*.

`(locate-source` *sfd pos*`)` **procedure**
`(locate-source` *sfd pos use-cache?*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

sfd must be a source-file descriptor, and *pos* must be an exact nonnegative integer.

This procedure either uses cached information from a previous request for *sfd* (only when *use-cache?* is provided as true) or attempts to locate and open the source file identified by *sfd*. If successful, it returns three values: a string *path*, an exact nonnegative integer *line*, and an exact nonnegative integer *char* representing the absolute pathname, line, and character position within the line represented by the specified source-file descriptor and file position. If unsuccessful, it returns zero values. It can fail even if a file with the correct

name exists in one of the source directories when the file's checksum does not match the checksum recorded in *sfd*.

`(locate-source-object-source` *source-object get-start? use-cache?*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

This procedure is similar to `locate-source`, but instead of taking an sfd and a position, it takes a source object plus a request for either the start or end location.

If *get-start?* is true and *source-object* has a line and column, this procedure returns the path in *source-objects*'s sfd, *source-object*'s line, and *source-objects*'s column.

If *source-object* has no line and column, then this procedure calls `locate-source` on *source-object*'s sfd, either *source-object*'s bfp or efp depending on *get-start?*, and *use-cache?*.

`current-locate-source-object-source` **thread parameter**
libraries: `(chezscheme)`

`current-locate-source-object-source` determines the source-location lookup function that is used by the system to report errors based on source objects. This parameter is initially bound to `locate-source-object-object`.

Adjust this parameter to control the way that source locations are extracted from source objects, possibly using recorded information, caches, and the filesystem in a way different from `locate-source-object-object`.

12. System Operations

This chapter describes operations for handling exceptions, interrupts, environments, compilation and evaluation, profiling, controlling the operation of the system, timing and statistics, defining and setting parameters, and querying the operating system environment.

12.1. Exceptions

Chez Scheme provides some extensions to the Revised[6] Report exception-handling mechanism, including mechanisms for producing formatted error messages, displaying conditions, and redefining the base exception handler. These extensions are described in this section.

`(warning` *who msg irritant* `...)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`warning` raises a continuable exception with condition type `&warning` and should be used to describe situations for which the `&warning` condition type is appropriate, typically a situation that should not prevent the program from continuing but might result in a more serious problem at some later point.

The continuation object with which the exception is raised also includes a `&who` condition whose who field is *who* if *who* is not `#f`, a `&message` condition whose message field is *msg*, and an `&irritants` condition whose irritants field is `(`*irritant* `...)`.

who must be a string, a symbol, or `#f` identifying the procedure or syntactic form reporting the warning upon whose behalf the warning is being reported. It is usually best to identify a procedure the programmer has called rather than some other procedure the programmer may not be aware is involved in carrying out the operation. *msg* must be a string and should describe the exceptional situation. The irritants may be any Scheme objects and should include values that may have caused or been materially involved in the exceptional situation.

`(assertion-violationf` *who msg irritant ...*`)` **procedure**
returns: does not return
`(errorf` *who msg irritant ...*`)` **procedure**
returns: does not return
`(warningf` *who msg irritant ...*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

These procedures are like `assertion-violation`, `error`, and `warning` except that *msg* is assumed to be a format string, as if in a call to `format` (Section 9.13), with *irritant ...* treated as the additional arguments to `format`. This allows programs to control the appearance of the error message, at least when the default exception handler is in place.

For each of these procedures, the continuation object with which the exception is raised includes a `&format` condition to signify that the string contained in the condition object's `&message` condition is a `format` string and the objects contained in the condition object's `&irritants` condition should be treated as the additional `format` arguments.

`&format` **syntax**
`(make-format-condition)` **procedure**
returns: a condition of type `&format`
`(format-condition?` *obj*`)` **procedure**
returns: `#t` if *obj* is a condition of type `&format`, `#f` otherwise
libraries: `(chezscheme)`

Presence of this condition type within a compound condition indicates that the string provided by the `&message` condition, if present, is a `format` string and the list of objects provided by the `&irritants` condition, if present, should be treated as additional `format` arguments. This condition type might be defined as follows.

```
(define-condition-type &format &condition
  make-format-condition format-condition?)
```

`&source` **syntax**
`(make-source-condition` *form*`)` **procedure**
returns: a condition of type `&source`
`(source-condition?` *obj*`)` **procedure**
returns: `#t` if *obj* is a condition of type `&source`, `#f` otherwise
`(source-condition-form` *condition*`)` **procedure**
returns: the contents of *condition*'s `form` field
libraries: `(chezscheme)`

This condition type can be included within a compound condition when a source expression can be identified in situations in which a `&syntax` condition would be inappropriate, such as when a run-time assertion violation is detected. The `form` argument should be an s-expression or syntax object representing the source expression. This condition type might be defined as follows.

```
(define-condition-type &source &condition
  make-source-condition source-condition?
  (form source-condition-form))
```

`&continuation` **syntax**

(`make-continuation-condition` *continuation*) **procedure**

returns: a condition of type `&continuation`

(`continuation-condition?` *obj*) **procedure**

returns: `#t` if *obj* is a condition of type `&continuation`, `#f` otherwise

(`condition-continuation` *condition*) **procedure**

returns: the contents of *condition*'s `continuation` field

libraries: `(chezscheme)`

This condition type can be included within a compound condition to indicate the current continuation at the point where the exception described by the condition occurred. The continuation of a failed `assert` or a call to `assertion-violation`, `assertion-violationf`, `error`, `errorf`, or `syntax-error` is now included via this condition type in the conditions passed to `raise`. The `continuation` argument should be a continuation. This condition type might be defined as follows.

```
(define-condition-type &continuation &condition
  make-continuation-condition continuation-condition?
  (continuation condition-continuation))
```

(`display-condition` *obj*) **procedure**

(`display-condition` *obj textual-output-port*) **procedure**

returns: unspecified

libraries: `(chezscheme)`

If *textual-output-port* is not supplied, it defaults to the current output port. This procedure displays a message to the effect that an exception has occurred with value *obj*. If *obj* is a condition (Chapter 11 of *The Scheme Programming Language, 4th Edition*), it displays information encapsulated within the condition, handling messages, *who* conditions, irritants, source information, etc., as appropriate.

(`default-exception-handler` *obj*) **procedure**

returns: unspecified

libraries: `(chezscheme)`

This procedure is the default value of the `base-exception-handler` parameter called on a condition when no other exception handler has been defined or when all dynamically established exception handlers have chosen not to handle the condition. It first displays *obj*, as if with `display-condition`, to the console error port. For non-serious warning conditions, it returns immediately after displaying the condition.

For serious or other non-warning conditions, it saves the condition in the parameter `debug-condition`, where `debug` (Section 3.2) can retrieve it and allow it to be inspected. If the `debug-on-exception` parameter is set to `#f` (the default unless the `--debug-on-exception` command-line option is provided), the handler prints a message instructing the user to type `(debug)` to enter the debugger, then resets to the current café. Otherwise, the handler invokes `debug` directly and resets if `debug` returns.

If an I/O exception occurs while attempting to display the condition, the default exception handler resets (as if by calling `reset`). The intent is to avoid an infinite regression (ultimately ending in exhaustion of memory) in which the process repeatedly recurs back to the default exception handler trying to write to a console-error port (typically stderr) that is no longer writable, e.g., due to the other end of a pipe or socket having been closed.

`debug-on-exception` **global parameter**
libraries: `(chezscheme)`

The value of this parameter determines whether the default exception handler immediately enters the debugger immediately when it receives a serious or non-warning condition. If the `--debug-on-exception` command-line option (Section 2.1) has been provided, the initial value of this parameter is `#t`. Otherwise, the initial value is `#f`.

`base-exception-handler` **thread parameter**
libraries: `(chezscheme)`

The value of this parameter must be a procedure, and the procedure should accept one argument. The default value of `base-exception-handler` is the procedure `default-exception-handler`.

The value of this parameter is invoked whenever no exception handler established by a program has chosen to handle an exception.

`debug-condition` **thread parameter**
libraries: `(chezscheme)`

This parameter is used by the default exception handler to hold the last serious or non-warning condition received by the handler, where it can be inspected via the `debug` procedure (Section 3.2). It can also be invoked by user code to store or retrieve a condition.

`current-exception-state` **thread parameter**
libraries: `(chezscheme)`

`current-exception-state` may be used to get or set the current exception state. When called without arguments, `current-exception-state` returns an *exception state* comprising the current stack of handlers established by `with-exception-handler` and `guard`. When called with a single argument, which must be an exception state, `current-exception-state` sets the exception state.

(create-exception-state) **procedure**
(create-exception-state *procedure*) **procedure**
libraries: (chezscheme)

create-exception-state creates an exception state whose stack of exception handlers is empty except for, in effect, an infinite number of occurrences of *handler* at its base. *handler* must be a procedure, and should accept one argument. If not provided, *handler* defaults to a procedure equivalent to the value of the following expression.

```
(lambda (x) ((base-exception-handler) x))
```

12.2. Interrupts

Chez Scheme allows programs to control the action of the Scheme system when various events occur, including an interrupt from the keyboard, the expiration of an internal timer set by set-timer, a breakpoint caused by a call to break, or a request from the storage manager to initiate a garbage collection. These mechanisms are described in this section, except for the collect request mechanism, which is described in Section 13.1.

Timer, keyboard, and collect-request interrupts are supported via a counter that is decremented approximately once for each call to a nonleaf procedure. (A leaf procedure is one that does not itself make any calls.) When no timer is running, this counter is set to a default value (1000 in Version 9) when a program starts or after an interrupt occurs. If a timer is set (via set-timer), the counter is set to the minimum of the default value and the number of ticks to which the timer is set. When the counter reaches zero, the system looks to see if the timer is set and has expired or if a keyboard or collect request interrupt has occurred. If so, the current procedure call is pended ("put on hold") while the appropriate interrupt handler is invoked to handle the interrupt. When (if) the interrupt handler returns, the pended call takes place. Thus, timer, keyboard, and collect-request interrupts effectively occur synchronously with respect to the procedure call mechanism, and keyboard and collect request interrupts may be delayed by a number of calls equal to the default timer value.

Calls to the break handler occur immediately whenever break is called.

(break *who msg irritant ...*) **procedure**
(break *who*) **procedure**
(break) **procedure**
returns: unspecified
libraries: (chezscheme)

The arguments to break follow the protocol described above for errorf. The default break handler (see break-handler) displays a message and invokes the debugger. The format string and objects may be omitted, in which case the message issued by the default break handler identifies the break using the *who* argument but provides no more information about the break. If the *who* argument is omitted as well, no message is generated. The default break handler returns normally if the debugger exits normally.

break-handler **thread parameter**
libraries: (chezscheme)

The value of this parameter must be a procedure. The current break handler is called by `break`, which passes along its arguments. See `break` for a description of the default break handler. The example below shows how to disable breaks.

```
(break-handler (lambda args (void)))
```

keyboard-interrupt-handler **thread parameter**
libraries: (chezscheme)

The value of this parameter must be a procedure. The keyboard-interrupt handler is called (with no arguments) when a keyboard interrupt occurs. The default keyboard-interrupt handler invokes the interactive debugger. If the debugger exits normally the interrupted computation is resumed. The example below shows how to install a keyboard-interrupt handler that resets without invoking the debugger.

```
(keyboard-interrupt-handler
  (lambda ()
    (newline (console-output-port))
    (reset)))
```

(set-timer *n*) **procedure**
returns: previous current timer value
libraries: (chezscheme)

n must be a nonnegative integer. When n is nonzero, `set-timer` starts an internal timer with an initial value of n. When n ticks elapse, a timer interrupt occurs, resulting in invocation of the timer interrupt handler. Each tick corresponds roughly to one nonleaf procedure call (see the introduction to this section); thus, ticks are not uniform time units but instead depend heavily on how much work is done by each procedure call.

When n is zero, `set-timer` turns the timer off.

The value returned in either case is the value of the timer before the call to `set-timer`. A return value of 0 should not be taken to imply that the timer was not on; the return value may also be 0 if the timer was just about to fire when the call to `set-timer` occurred.

The engine mechanism (Section 6.4) is built on top of the timer interrupt so timer interrupts should not be used with engines.

timer-interrupt-handler **thread parameter**
libraries: (chezscheme)

The value of this parameter must be a procedure. The timer interrupt handler is called by the system when the internal timer (set by `set-timer`) expires. The default handler raises

an exception with condition type `&assertion` to say that the handler has not been defined; any program that uses the timer should redefine the handler before setting the timer.

`(disable-interrupts)` **procedure**
`(enable-interrupts)` **procedure**
returns: disable count
libraries: `(chezscheme)`

`disable-interrupts` disables the handling of interrupts, including timer, keyboard, and collect request interrupts. `enable-interrupts` re-enables these interrupts. The system maintains a disable count that starts at zero; when zero, interrupts are enabled. Each call to `disable-interrupts` increments the count, effectively disabling interrupts. Each call to `enable-interrupts` decrements the count, if not already zero, effectively enabling interrupts. For example, two calls to `disable-interrupts` followed by one call to `enable-interrupts` leaves interrupts disabled. Calls to `enable-interrupts` when the count is already zero (and interrupts are enabled) have no effect. The value returned by either procedure is the number of calls to `enable-interrupts` required to enable interrupts.

Great care should be exercised when using these procedures, since disabling interrupts inhibits the normal processing of keyboard interrupts, timer interrupts, and, perhaps most importantly, collect request interrupts. Since garbage collection does not happen automatically when interrupts are disabled, it is possible for the storage allocator to run out of space unnecessarily should interrupts be disabled for a long period of time.

The `with-interrupts-disabled` syntactic form should be used instead of these more primitive procedures whenever possible, since `with-interrupts-disabled` ensures that interrupts are re-enabled whenever a nonlocal exit occurs, such as when an exception is handled by the default exception handler.

`(with-interrupts-disabled` *body*$_1$ *body*$_2$ `...)` **syntax**
`(critical-section` *body*$_1$ *body*$_2$ `...)` **syntax**
returns: the values of the body *body*$_1$ *body*$_2$ `...`
libraries: `(chezscheme)`

`with-interrupts-disabled` evaluates the body *body*$_1$ *body*$_2$ ... with interrupts disabled. That is, upon entry, interrupts are disabled, and upon exit, interrupts are re-enabled. Thus, `with-interrupts-disabled` allows the implementation of indivisible operations in nonthreaded versions of *Chez Scheme* or within a single thread in threaded versions of *Chez Scheme*. `critical-section` is the same as `with-interrupts-disabled` and is provided for backward compatibility.

`with-interrupts-disabled` can be defined as follows.

```
(define-syntax with-interrupts-disabled
  (syntax-rules ()
    [(_ b1 b2 ...)
     (dynamic-wind
       disable-interrupts
```

```
(lambda () b1 b2 ...)
enable-interrupts)]))
```

The use of `dynamic-wind` ensures that interrupts are disabled whenever the body of the `with-interrupts-disabled` expression is active and re-enabled whenever it is not. Since calls to `disable-interrupts` are counted (see the discussion under `disable-interrupts` and `enable-interrupts` above), `with-interrupts-disabled` expressions may be nested with the desired effect.

`(register-signal-handler` *sig procedure*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`register-signal-handler` is used to establish a signal handler for a given low-level signal. *sig* must be an exact integer identifying a valid signal, and *procedure* should accept one argument. See your host system's `<signal.h>` or documentation for a list of valid signals and their numbers. After a signal handler for a given signal has been registered, receipt of the specified signal results in a call to the handler. The handler is passed the signal number, allowing the same handler to be used for different signals while differentiating among them.

Signals handled in this fashion are treated like keyboard interrupts in that the handler is not called immediately when the signal is delivered to the process, but rather at some procedure call boundary after the signal is delivered. It is generally not a good idea, therefore, to establish handlers for memory faults, illegal instructions, and the like, since the code that causes the fault or illegal instruction will continue to execute (presumably erroneously) for some time before the handler is invoked.

`register-signal-handler` is supported only on Unix-based systems.

12.3. Environments

Environments are first-class objects containing identifier bindings. They are similar to modules but, unlike modules, may be manipulated at run time. Environments may be provided as optional arguments to `eval`, `expand`, and the procedures that define, assign, or reference top-level values.

There are several built-in environments, and new environments can be created by copying existing environments or selected bindings from existing environments.

Environments can be mutable or immutable. A mutable environment can be extended with new bindings, its existing bindings can be modified, and its variables can be assigned. An immutable environment cannot be modified in any of these ways.

`(environment?` *obj*`)` **procedure**
returns: `#t` if *obj* is an environment, otherwise `#f`
libraries: `(chezscheme)`

```
(environment? (interaction-environment))  ⇒  #t
(environment? 'interaction-environment)  ⇒  #f
(environment? (copy-environment (scheme-environment)))  ⇒  #t
(environment? (environment '(prefix (rnrs) $rnrs-)))  ⇒  #t
```

`(environment-mutable?` *env*`)` **procedure**
returns: `#t` if *env* is mutable, otherwise `#f`
libraries: `(chezscheme)`

```
(environment-mutable? (interaction-environment))  ⇒  #t
(environment-mutable? (scheme-environment))  ⇒  #f
(environment-mutable? (copy-environment (scheme-environment)))  ⇒  #t
(environment-mutable? (environment '(prefix (rnrs) $rnrs-)))  ⇒  #f
```

`(scheme-environment)` **procedure**
returns: an environment
libraries: `(chezscheme)`

`scheme-environment` returns an environment containing the initial top-level bindings. This environment corresponds to the `scheme` module.

The environment returned by this procedure is immutable.

```
(define cons 3)
(top-level-value 'cons (scheme-environment))  ⇒  #<procedure cons>
(set-top-level-value! 'cons 3 (scheme-environment))  ⇒  exception
```

`(ieee-environment)` **procedure**
returns: an IEEE/ANSI standard compatibility environment
libraries: `(chezscheme)`

`ieee-environment` returns an environment containing bindings for the keywords and variables whose meanings are defined by the IEEE/ANSI Standard for Scheme [26].

The bindings for each of the identifiers in the IEEE environment are those of the corresponding Revised[6] Report library, so this does not provide full backward compatibility.

The environment returned by this procedure is immutable.

```
(define cons 3)
(top-level-value 'cons (ieee-environment))  ⇒  #<procedure cons>
(set-top-level-value! 'cons 3 (ieee-environment))  ⇒  exception
```

`interaction-environment` **thread parameter**
libraries: `(chezscheme)`

The original value of `interaction-environment` is the default top-level environment. It is

initially set to a mutable copy of `(scheme-environment)` and which may be extended or otherwise altered by top-level definitions and assignments. It may be set to any environment, mutable or not, to change the default top-level evaluation environment.

An expression's top-level bindings resolve to the environment that is in effect when the expression is expanded, and changing the value of this parameter has no effect on running code. Changes affect only code that is subsequently expanded, e.g., as the result of a call to `eval`, `load`, or `compile-file`.

```
(define cons 3)
cons  ⇒  3
(top-level-value 'cons (interaction-environment))  ⇒  3

(interaction-environment (scheme-environment))
cons  ⇒  #<procedure cons>
(set! cons 3)  ⇒  exception: attempt to assign immutable variable
(define cons 3)  ⇒  exception: invalid definition in immutable environment
```

`(copy-environment` *env*`)` **procedure**
`(copy-environment` *env mutable?*`)` **procedure**
`(copy-environment` *env mutable? syms*`)` **procedure**
returns: a new environment
libraries: `(chezscheme)`

`copy-environment` returns a copy of *env*, i.e., a new environment that contains the same bindings as *env*.

The environment is mutable if *mutable?* is omitted or true; if *mutable?* is false, the environment is immutable.

The set of bindings copied from *env* to the new environment is determined by *syms*, which defaults to the value of `(environment-symbols` *env*`)`. The binding, if any, for each element of *syms* is copied to the new environment, and no other bindings are present in the new environment.

In the current implementation, the storage space used by an environment is never collected, so repeated use of `copy-environment` will eventually cause the system to run out of memory.

```
(define e (copy-environment (scheme-environment)))
(eval '(define cons +) e)
(eval '(cons 3 4) e)                      ⇒  7
(eval '(cons 3 4) (scheme-environment))   ⇒  (3 . 4)
```

`(environment-symbols` *env*`)` **procedure**
returns: a list of symbols
libraries: `(chezscheme)`

This procedure returns a list of symbols representing the identifiers bound in environment *env*. It is primarily useful in building the list of symbols to be copied from one environment to another.

```
(define listless-environment
  (copy-environment
    (scheme-environment)
    #t
    (remq 'list (environment-symbols (scheme-environment)))))
(eval '(let ([x (cons 3 4)]) x) listless-environment)  ⇒  (3 . 4)
(eval '(list 3 4) listless-environment)  ⇒  exception
```

(apropos-list *s*) **procedure**
(apropos-list *s env*) **procedure**
returns: see below
libraries: (chezscheme)

This procedure returns a selected list of symbols and pairs. Each symbol in the list represents an identifier bound in *env*. Each pair represents a set of identifiers exported by a predefined library or a library previously defined or loaded into the system. The car of the pair is the library name, and the cdr is a list of symbols. If *s* is a string, only entries whose names have *s* as a substring are included, and if *s* is a symbol, only those whose names have the name of *s* as a substring are selected. If no environment is provided, it defaults to the value of `interaction-environment`.

```
(library (a) (export a-vector-sortof) (import (rnrs))
  (define a-vector-sortof '(vector 1 2 3)))
(apropos-list 'vector-sort)  ⇒
  (vector-sort vector-sort!
   ((a) a-vector-sortof)
   ((chezscheme) vector-sort vector-sort!)
   ((rnrs) vector-sort vector-sort!)
   ((rnrs sorting) vector-sort vector-sort!)
   ((scheme) vector-sort vector-sort!))
```

(apropos *s*) **procedure**
(apropos *s env*) **procedure**
returns: unspecified
libraries: (chezscheme)

`apropos` is like `apropos-list` except the information is displayed to the current output port, as shown in the following transcript.

```
> (library (a) (export a-vector-sortof) (import (rnrs))
    (define a-vector-sortof '(vector 1 2 3)))
> (apropos 'vector-sort)
interaction environment:
  vector-sort, vector-sort!
(a):
  a-vector-sortof
(chezscheme):
  vector-sort, vector-sort!
```

```
(rnrs):
  vector-sort, vector-sort!
(rnrs sorting):
  vector-sort, vector-sort!
(scheme):
  vector-sort, vector-sort!
```

12.4. Compilation, Evaluation, and Loading

`(eval` *obj*`)` **procedure**
`(eval` *obj env*`)` **procedure**
returns: value of the Scheme form represented by *obj*
libraries: `(chezscheme)`

`eval` treats *obj* as the representation of an expression. It evaluates the expression in environment *env* and returns its value. If no environment is provided, it defaults to the environment returned by `interaction-environment`.

Single-argument `eval` is a *Chez Scheme* extension. *Chez Scheme* also permits *obj* to be the representation of a nonexpression form, i.e., a definition, whenever the environment is mutable. *Chez Scheme* further allows *obj* to be an annotation (Section 11.11), and the default evaluators make use of annotations to incorporate source-file information in error messages and associate source-file information with compiled code.

In *Chez Scheme*, `eval` is actually a wrapper that simply passes its arguments to the current evaluator. (See `current-eval`.) The default evaluator is `compile`, which expands the expression via the current expander (see `current-expand`), compiles it, executes the resulting code, and returns its value. If the environment argument, *env*, is present, `compile` passes it along to the current expander, which is `sc-expand` by default.

`current-eval` **thread parameter**
libraries: `(chezscheme)`

`current-eval` determines the evaluation procedure used by the procedures `eval`, `load`, and `new-cafe`. `current-eval` is initially bound to the value of `compile`. (In *Petite Chez Scheme*, it is initially bound to the value of `interpret`.) The evaluation procedureshould expect one or two arguments: an object to evaluate and an optional environment. The second argument might be an annotation (Section 11.11).

```
(current-eval interpret)
(+ 1 1)  ⇒  2

(current-eval (lambda (x . ignore) x))
(+ 1 1)  ⇒  (+ 1 1)
```

`(compile` *obj*`)` **procedure**
`(compile` *obj env*`)` **procedure**
returns: value of the Scheme form represented by *obj*
libraries: `(chezscheme)`

obj, which can be an annotation (Section 11.11) or unannotated value, is treated as a Scheme expression, expanded with the current expander (the value of `current-expand`) in the specified environment (or the interaction environment, if no environment is provided), compiled to machine code, and executed. `compile` is the default value of the `current-eval` parameter.

`(interpret` *obj*`)` **procedure**
`(interpret` *obj env*`)` **procedure**
returns: value of the Scheme form represented by *obj*
libraries: `(chezscheme)`

`interpret` is like `compile`, except that the expression is interpreted rather than compiled. `interpret` may be used as a replacement for `compile`, with the following caveats:

- Interpreted code runs significantly slower.
- Inspector information is not generated for interpreted code, so the inspector is not as useful for interpreted code as it is for compiled code.
- Foreign procedure expressions cannot be interpreted, so the interpreter invokes the compiler for all foreign procedure expressions (this is done transparently).

`interpret` is sometimes faster than `compile` when the form to be evaluated is short running, since it avoids some of the work done by `compile` prior to evaluation.

`(load` *path*`)` **procedure**
`(load` *path eval-proc*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. `load` reads and evaluates the contents of the file specified by *path*. The file may contain source or object code. By default, `load` employs `eval` to evaluate each source expression found in a source file. If *eval-proc* is specified, `load` uses this procedure instead. *eval-proc* must accept one argument, the expression to evaluate. The expression passed to *eval-proc* might be an annotation (Section 11.11) or an unannotated value.

The *eval-proc* argument facilitates the implementation of embedded Scheme-like languages and the use of alternate evaluation mechanisms to be used for Scheme programs. *eval-proc* can be put to other uses as well. For example,

```
(load "myfile.ss"
  (lambda (x)
    (pretty-print
      (if (annotation? x)
          (annotation-stripped x)
          x))
    (newline)
    (eval x)))
```

pretty-prints each expression before evaluating it.

The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files not identified by absolute path names.

`(load-library` *path*`)` **procedure**
`(load-library` *path eval-proc*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`load-library` is identical to `load` except that it treats the input file as if it were prefixed by an implicit `#!r6rs`. This effectively disables any non-R6RS lexical syntax except where subsequently overridden by `#!chezscheme`.

`(load-program` *path*`)` **procedure**
`(load-program` *path eval-proc*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. `load-program` reads and evaluates the contents of the file specified by *path*. The file may contain source or object code. If it contains source code, `load-program` wraps the code in a `top-level-program` form so that the file's content is treated as an RNRS top-level program (Section 10.3 of *The Scheme Programming Language, 4th Edition*). By default, `load-program` employs `eval` to evaluate each source expression found in the file. If *eval-proc* is specified, `load-program` uses this procedure instead. *eval-proc* must accept one argument, the expression to evaluate. The expression passed to *eval-proc* might be an annotation (Section 11.11) or an unannotated value.

The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files not identified by absolute path names.

`(visit` *path*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. `visit` reads the named file, which must contain compiled object code compatible with the current machine type and version, and it runs those portions of the compiled object code that establish compile-time information or correspond to expressions identified as "visit" time by `eval-when` forms contained in the original source file.

For example, assume the file `t1.ss` contains the following forms:

```
(define-syntax a (identifier-syntax 3))
(module m (x) (define x 4))
(define y 5)
```

If `t1.ss` is compiled to `t1.so`, applying `load` to `t1.so` has the effect of defining all three identifiers. Applying `visit` to `t1.so`, however, has the effect of installing the transformer for `a`, installing the interface for `m` (for use by `import`), and recording `y` as a variable. `visit` is useful when separately compiling one file that depends on bindings defined in another without actually loading and evaluating the code in the supporting file.

The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files not identified by absolute path names.

`(revisit` *path*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string. `revisit` reads the named file, which must contain compiled object code compatible with the current machine type and version, and it runs those portions of the compiled object code that compute run-time values or correspond to expressions identified as "revisit" time by `eval-when` forms contained in the original source file.

Continuing the example given for `visit` above, applying `revisit` to the object file, `t1.so`, has the effect of establishing the values of the variable `x` exported from `m` and the top-level variable `y`, without installing either the interface for `m` or the transformer for `a`.

`revisit` is useful for loading compiled application code without loading unnecessary compile-time information. Care must be taken when using this feature if the application calls `eval` or uses `top-level-value`, `set-top-level-value!`, or `top-level-syntax` to access top-level bindings at run-time, since these procedures use compile-time information to resolve top-level bindings.

The parameter `source-directories` (Section 12.5) determines the set of directories searched for source files not identified by absolute path names.

`(compile-file` *input-filename*`)` **procedure**
`(compile-file` *input-filename output-filename*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

input-filename and *output-filename* must be strings. *input-filename* must name an existing, readable file. It must contain a sequence of zero or more source expressions; if this is not the case, `compile-file` raises an exception with condition type `&syntax`.

The normal evaluation process proceeds in two steps: compilation and execution. `compile-file` performs the compilation process for an entire source file, producing an object file. When the object file is subsequently loaded (see `load`), the compilation process is not necessary, and the file typically loads several times faster.

If the optional *output-filename* argument is omitted, the actual input and output filenames are determined as follows. If *input-filename* has no extension, the input filename is *input-filename* followed by `.ss` and the output filename is *input-filename* followed by `.so`. If *input-filename* has the extension `.so`, the input filename is *input-filename* and the output filename is *input-filename* followed by `.so`. Otherwise, the input filename is *input-filename* and the output filename is *input-filename* without its extension, followed by `.so`. For example, `(compile-file "myfile")` produces an object file with the name `"myfile.so"` from the source file named `"myfile.ss"`, `(compile-file "myfile.sls")` produces an object file with the name `"myfile.so"` from the source file named `"myfile.sls"`, and `(compile-file "myfile1" "myfile2")` produces an object file with the name `"myfile2"` from the source file name `"myfile1"`.

Before compiling a file, `compile-file` saves the values of the following parameters:

```
optimize-level
debug-level
run-cp0
cp0-effort-limit
cp0-score-limit
cp0-outer-unroll-limit
generate-inspector-information
compile-profile
generate-interrupt-trap
enable-cross-library-optimization
```

It restores the values after the file has been compiled. This allows the programmer to control the values of these parameters on a per-file basis, e.g., via an `eval-when` with situation `compile` embedded in the source file. For example, if

```
(eval-when (compile) (optimize-level 3))
```

appears at the top of a source file, the optimization level is set to 3 just while the remainder of file is compiled.

`(compile-script` *input-filename*`)` **procedure**
`(compile-script` *input-filename output-filename*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

input-filename and *output-filename* must be strings.

`compile-script` is like `compile-file` but differs in that it copies the leading `#!` line from the source-file script into the object file. When the `#!` line is present it is uncompressed in the output file even when the parameter `compile-compressed` is set to `#t`, causing the remainder of the file to be compressed. This allows it to be interpreted properly by the operating system.

`compile-script` permits compiled script files to be created from source script to reduce script load time. As with source-code scripts, compiled scripts may be run with the `--script` command-line option.

`(compile-library` *input-filename*`)` **procedure**
`(compile-library` *input-filename output-filename*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

input-filename and *output-filename* must be strings.

`compile-library` is identical to `compile-file` except that it treats the input file as if it were prefixed by an implicit `#!r6rs`. This effectively disables any non-R6RS lexical syntax except where subsequently overridden by `#!chezscheme`.

`(compile-program` *input-filename*`)` **procedure**
`(compile-program` *input-filename output-filename*`)` **procedure**
returns: a list of libraries invoked by the program
libraries: `(chezscheme)`

input-filename and *output-filename* must be strings.

`compile-program` is like `compile-script` but differs in that it implements the semantics of RNRS top-level programs, while `compile-script` implements the semantics of the interactive top-level. The resulting compiled program will also run faster than if compiled via `compile-file` or `compile-script`.

`compile-program` returns a list of libraries directly invoked by the compiled top-level program, excluding built-in libraries like `(rnrs)` and `(chezscheme)`. The procedure `library-requirements` may be used to determine the indirect requirements, i.e., additional libraries required by the directly invoked libraries. When combined with `library-object-filename`, this information can be used to determine the set of files that must be distributed with the compiled program file.

A program invokes a library only if it references one or more variables exported from the library. The set of libraries invoked by a top-level program, and hence loaded when the program is loaded, might be smaller than the set imported by the program, and it might be larger than the set directly imported by the program.

As with source-code top-level programs, compiled top-level programs may be run with the `--program` command-line option.

`(maybe-compile-file` *input-filename*`)` **procedure**
`(maybe-compile-file` *input-filename output-filename*`)` **procedure**
`(maybe-compile-library` *input-filename*`)` **procedure**
`(maybe-compile-library` *input-filename output-filename*`)` **procedure**
`(maybe-compile-program` *input-filename*`)` **procedure**
`(maybe-compile-program` *input-filename output-filename*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

These procedures are like their non-`maybe` counterparts but compile the source file only if the object file is out-of-date. An object file X is considered out-of-date if it does not exist or if it is older than the source file or any files included (via `include`) when X was

created. When the value of the parameter `compile-imported-libraries` is `#t`, X is also considered out-of-date if the object file for any library imported when X was compiled is out-of-date. If `maybe-compile-file` determines that compilation is necessary, it compiles the source file by passing `compile-file` the input and output filenames. `compile-library` does so by similarly invoking the value of the `compile-library-handler` parameter, and `compile-program` does so by similarly invoking the value of the `compile-program-handler` parameter.

When *output-filename* is not specified, the input and output filenames are determined in the same manner as for `compile-file`.

`compile-library-handler` **thread parameter**
libraries: `(chezscheme)`

This parameter must be set to a procedure, and the procedure should accept two string arguments naming a source file and an object file. The procedure should typically invoke `compile-library` and pass it the two arguments, but it can also use one of the other file or port compilation procedures. For example, it might read the source file using its own parser and use `compile-to-file` to finish the compilation process. The procedure can perform other actions as well, such as parameterizing compilation parameters, establishing guards, or gathering statistics. The default value of this parameter simply invokes `compile-library` on the two string arguments without taking any other action.

The value of this parameter is called by `maybe-compile-library` when the object file is out-of-date. It is also called by the expander to compile an imported library when `compile-imported-libraries` is `#t` and the expander determines the object file is out-of-date.

`compile-program-handler` **thread parameter**
libraries: `(chezscheme)`

This parameter must be set to a procedure, and the procedure should accept two string arguments naming a source file and an object file. The procedure should typically invoke `compile-program` and pass it the two arguments, but it can also use one of the other file or port compilation procedures. For example, it might read the source file using its own parser and use `compile-to-file` to finish the compilation process. The procedure can perform other actions as well, such as parameterizing compilation parameters, establishing guards, or gathering statistics. The default value of this parameter simply invokes `compile-program` on the two string arguments without taking any other action and returns the list of libraries returned by `compile-program`.

The value of this parameter is called by `maybe-compile-program` when the object file is out-of-date.

`(compile-whole-program` *input-filename output-filename*`)` **procedure**
`(compile-whole-program` *input-filename output-filename libs-visible?*`)` **procedure**
returns: a list of libraries left to be loaded at run time
libraries: `(chezscheme)`

`compile-whole-program` accepts as input a filename naming a "whole program optimization" (wpo) file for a top-level program and produces an object file incorporating the program and each library upon which it depends, provided that a wpo file for the library can be found.

If a wpo file for a required library cannot be found, but an object file for the library can, the library is not incorporated in the resulting object file. Such libraries are left to be loaded at run time. `compile-whole-program` returns a list of such libraries. If there are no such libraries, the resulting object file is self-contained and `compile-whole-program` returns the empty list.

The libraries incorporated into the resulting object file are visible (for use by `environment` and `eval`) if the *libs-visible?* argument is supplied and non-false. Any library incorporated into the resulting object file and required by an object file left to be loaded at run time is also visible.

input-filename and *output-filename* must be strings. *input-filename* must identify a wpo file, and a wpo or object file must also be present for each required library somewhere in the directories specified by the `library-directories` parameter.

To the extent possible given the specified set of visible libraries and requirements of libraries to be loaded at run time, `compile-whole-program` discards unused code and optimizes across program and library boundaries, potentially reducing program load time, run time, and memory requirements. Some optimization also occurs even across the boundaries of libraries that are not incorporated into the output, though this optimization is limited in nature.

The procedures `compile-file`, `compile-program`, `compile-library`, `compile-script`, and `compile-whole-library` produce wpo files as well as ordinary object files when the `generate-wpo-files` parameter is set to `#t` (the default is `#f`). `compile-port` and `compile-to-port` do so when passed an optional wpo port.

`(compile-whole-library` *input-filename output-filename*`)` **procedure**
returns: a list of libraries left to be loaded at run time
libraries: `(chezscheme)`

`compile-whole-library` is like `compile-whole-program`, except *input-filename* must specify a wpo file for a library, all libraries are automatically made visible, and a new wpo file is produced (when `generate-wpo-files` is `#t`) as well as an object file for the resulting combination of libraries.

(**compile-port** *input-port output-port*) **procedure**
(**compile-port** *input-port output-port sfd*) **procedure**
(**compile-port** *input-port output-port sfd wpo-port*) **procedure**
returns: unspecified
libraries: (**chezscheme**)

input-port must be a textual input port. *output-port* and, if present, *wpo-port* must be binary output ports. If present, *sfd* must be a source-file descriptor.

compile-port is like **compile-file** except that it takes input from an arbitrary textual input port and sends output to an arbitrary binary output port. If *sfd* is present, it is passed to the reader so that source information can be associated with the expressions read from *input-port*. It is also used to associate block-profiling information with the input file name encapsulated within *sfd*. If *wpo-port* is present, it sends whole-program optimization information to *wpo-port* for use by **compile-whole-program**.

None of the ports is closed automatically after compilation; it is assumed that the program that opens the ports and invokes **compile-port** will take care of closing the ports. The output will be compressed only if *binary-output-port* is set up to do compression, e.g., if it was opened with the **compressed** file option.

(**compile-to-port** *obj-list output-port*) **procedure**
(**compile-to-port** *obj-list output-port sfd*) **procedure**
(**compile-to-port** *obj-list output-port sfd wpo-port*) **procedure**
returns: see below
libraries: (**chezscheme**)

obj-list must be a list containing a sequence of objects that represent syntactically valid expressions, each possibly annotated (Section 11.11). If any of the objects does not represent a syntactically valid expression, **compile-to-port** raises an exception with condition type **&syntax**. *output-port* and, if present, *wpo-port* must be binary output ports. If present, *sfd* must be a source-file descriptor.

compile-to-port is like **compile-file** except that it takes input from a list of objects and sends output to an arbitrary binary output port. *sfd* is used to associate block-profiling information with the input file name encapsulated within *sfd*. If *wpo-port* is present, it sends whole-program optimization information to *wpo-port* for use by **compile-whole-program**.

The output port is not closed automatically after compilation; it is assumed that the program that opens the port and invokes **compile-to-port** will take care of closing the port.

The output will be compressed only if *binary-output-port* is set up to do compression, e.g., if it was opened with the **compressed** file option.

When *obj-list* contains a single list-structured element whose first-element is the symbol **top-level-program**, **compile-to-port** returns a list of the libraries the top-level program requires at run time, as with **compile-program**. Otherwise, the return value is unspecified.

`(compile-to-file` *obj-list output-file*`)` **procedure**
`(compile-to-file` *obj-list output-file sfd*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

obj-list must be a list containing a sequence of objects that represent syntactically valid expressions, each possibly annotated (Section 11.11). If any of the objects does not represent a syntactically valid expression, `compile-to-file` raises an exception with condition type `&syntax`. *output-file* must be a string. If present, *sfd* must be a source-file descriptor.

`compile-to-file` is like `compile-file` except that it takes input from a list of objects. *sfd* is used to associate block-profiling information with the input file name encapsulated within *sfd*.

When *obj-list* contains a single list-structured element whose first-element is the symbol `top-level-program`, `compile-to-file` returns a list of the libraries the top-level program requires at run time, as with `compile-program`. Otherwise, the return value is unspecified.

`(make-boot-file` *output-filename base-boot-list input-filename* `...)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

output-filename, *input-filename*, and the elements of *base-boot-list* must be strings.

`make-boot-file` writes a boot header to the file named by *output-filename*, followed by the object code for each *input-filename* in turn. If an input file is not already compiled, `make-boot-file` compiles the file as it proceeds.

The boot header identifies the elements of *base-boot-list* as alternative boot files upon which the new boot file depends. If the list of strings naming base boot files is empty, the first named input file should be a base boot file, i.e., petite.boot or some boot file derived from petite.boot.

Boot files are loaded explicitly via the `--boot` or `-b` command-line options or implicitly based on the name of the executable (Section 2.9).

See Section 2.8 for more information on boot files and the use of `make-boot-file`.

`(make-boot-header` *output-filename* $base\text{-}boot_1$ $base\text{-}boot_2$ `...)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedure has been subsumed by `make-boot-file` and is provided for backward compatibility. The call

`(make-boot-header` *output-filename* $base\text{-}boot_1$ $base\text{-}boot_2$ `...)`

is equivalent to

`(make-boot-file` *output-filename* `'(`$base\text{-}boot_1$ $base\text{-}boot_2$ `...))`

`(strip-fasl-file` *input-path output-path options*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

input-path and *output-path* must be strings. *input-path* must name an existing, readable file containing object code produced by `compile-file`, one of the other file-compiling procedures, or an earlier run of `strip-fasl-file`. *options* must be an enumeration set over the symbols constituting valid strip options, as described in the `fasl-strip-options` entry below.

The new procedure `strip-fasl-file` allows the removal of source information of various sorts from a compiled object (fasl) file produced by `compile-file` or one of the other file compiling procedures. It also allows removal of library visit code from object files containing compiled libraries. Visit code is the code for macro transformers and meta definitions required to compile (but not run) dependent libraries.

On most platforms, the input and output paths can be the same, in which case the input file is replaced with a new file containing the stripped object code. Using the same path will likely fail on Windows file systems, which do not generally permit an open file to be removed.

If *options* is empty, the output file is effectively equivalent to the input file, though it will not necessarily be identical.

`(fasl-strip-options` *symbol* `...)` **syntax**
returns: a fasl-strip-options enumeration set
libraries: `(chezscheme)`

Fasl-strip-options enumeration sets are passed to `strip-fasl-file` to determine what is stripped. The available options are described below.

`inspector-source`: Strip inspector source information. This includes source expressions that might otherwise be available for procedures and continuations with the "code" and "call" commands and messages in the interactive and object inspectors. It also includes filename and position information that might otherwise be available for the same via the "file" command and "source" messages.

`source-annotations`: Strip source annotations, which typically appear only on syntax objects, e.g., identifiers, in the templates of macro transformers.

`profile-source`: Strip source file and character position information from profiled code objects. This does not remove the profile counters or eliminate the overhead for incrementing them at run time.

`compile-time-information`: This strips compile-time information from compiled libraries, potentially reducing the size of the resulting file but making it impossible to use the file to compile dependent code. This option is useful for creating smaller object files to ship as part of a binary-only package.

`(machine-type)` **procedure**
returns: the current machine type
libraries: `(chezscheme)`

Consult the release notes for the current version of *Chez Scheme* for a list of supported machine types.

`(expand` *obj*`)` **procedure**
`(expand` *obj env*`)` **procedure**
returns: expansion of the Scheme form represented by *obj*
libraries: `(chezscheme)`

`expand` treats *obj* as the representation of an expression. It expands the expression in environment *env* and returns an object representing the expanded form. If no environment is provided, it defaults to the environment returned by `interaction-environment`.

obj can be an annotation (Section 11.11), and the default expander makes use of annotations to incorporate source-file information in error messages.

`expand` actually passes its arguments to the current expander (see `current-expand`), initially `sc-expand`.

See also `expand-output` (page 353) which can be used to request that the compiler or interpreter show expander output.

`current-expand` **thread parameter**
libraries: `(chezscheme)`

`current-expand` determines the expansion procedure used by the compiler, interpreter, and direct calls to `expand` to expand syntactic extensions. `current-expand` is initially bound to the value of `sc-expand`.

It may be set another procedure, but since the format of expanded code expected by the compiler and interpreter is not publicly documented, only `sc-expand` produces correct output, so the other procedure must ultimately be defined in terms of `sc-expand`.

The first argument to the expansion procedure represents the input expression. It can be an annotation (Section 11.11) or an unannotated value. the second argument is an environment. Additional arguments might be passed to the expansion procedure by the compiler, interpreter, and `expand`; their number and roles are unspecified.

`(sc-expand` *obj*`)` **procedure**
`(sc-expand` *obj env*`)` **procedure**
returns: the expanded form of *obj*
libraries: `(chezscheme)`

The procedure `sc-expand` is used to expand programs written using `syntax-case` macros. `sc-expand` is the default expander, i.e., the initial value of `current-expand`. *obj* represents the program to be expanded, and *env* must be an environment. *obj* can be an annotation

(Section 11.11) or unannotated value. If not provided, *env* defaults to the environment returned by `interaction-environment`.

`(expand/optimize` *obj*`)` **procedure**
`(expand/optimize` *obj env*`)` **procedure**
returns: result of expanding and optimizing form represented by *obj*
libraries: `(chezscheme)`

`expand/optimize` treats *obj* as the representation of an expression. *obj* can be an annotation (Section 11.11) or unannotated value. `expand/optimize` expands the expression in environment *env* and passes the expression through the source optimizer `cp0` (unless `cp0` is disabled via `run-cp0`). It also simplifies `letrec` and `letrec*` expressions within the expression and makes their undefined checks explicit. It returns an object representing the expanded, simplified, and optimized form. If no environment is provided, it defaults to the environment returned by `interaction-environment`.

`expand/optimize` is primarily useful for understanding what `cp0` does and does not optimize. Many optimizations are performed later in the compiler, so `expand/optimize` does not give a complete picture of optimizations performed.

```
(expand/optimize
  '(let ([y '(3 . 4)])
     (+ (car y) (cdr y))))  ⇒  7

(print-gensym #f)
(expand/optimize
  '(let ([y '(3 . 4)])
     (lambda (x)
       (* (+ (car y) (cdr y)) x))))  ⇒  (lambda (x) (#2%* 7 x))

(expand/optimize
  '(let ([n (expt 2 10)])
     (define even?
       (lambda (x) (or (zero? x) (not (odd? x)))))
     (define odd?
       (lambda (x) (not (even? (- x 1)))))
     (define f
       (lambda (x)
         (lambda (y)
           (lambda (z)
             (if (= z 0) (omega) (+ x y z))))))
     (define omega
       (lambda ()
         ((lambda (x) (x x)) (lambda (x) (x x)))))
     (let ([g (f 1)] [m (f n)])
       (let ([h (if (> ((g 2) 3) 5)
                    (lambda (x) (+ x 1))
                    odd?)])
         (h n)))))  ⇒  1025
```

See also `expand/optimize-output` (page 353) which can be used to request that the compiler or interpreter show source-optimizer output.

`(eval-when` *situations* *form*$_1$ *form*$_2$ `...)` **syntax**
returns: see below
libraries: `(chezscheme)`

situations must be a list containing some combination of the symbols `eval`, `compile`, `load`, `visit`, and `revisit`.

When source files are loaded (see `load`), the forms in the file are read, compiled, and executed sequentially, so that each form in the file is fully evaluated before the next one is read. When a source file is compiled (see `compile-file`), however, the forms are read and compiled, *but not executed*, in sequence. This distinction matters only when the execution of one form in the file affects the compilation of later forms, e.g., when the form results in the definition of a module or syntactic form or sets a compilation parameter such as `optimize-level` or `case-sensitive`.

For example, assume that a file contains the following two forms:

```
(define-syntax reverse-define
  (syntax-rules ()
    [(_ e x) (define x e)]))

(reverse-define 3 three)
```

Loading this from source has the effect of defining `reverse-define` as a syntactic form and binding the identifier `three` to 3. The situation may be different if the file is compiled with `compile-file`, however. Unless the system or programmer takes steps to assure that the first form is fully executed before the second expression is compiled, the syntax expander will not recognize `reverse-define` as a syntactic form and will generate code for a procedure call to `reverse-define` instead of generating code to define `three` to be 3. When the object file is subsequently loaded, the attempt to reference either `reverse-define` or `three` will fail.

As it happens, when a `define-syntax`, `module`, `import`, or `import-only` form appears at top level, as in the example above, the compiler does indeed arrange to evaluate it before going on to compile the remainder of the file. If the compiler encounters a variable definition for an identifier that was previously something else, it records that fact as well. The compiler also generates the appropriate code so that the bindings will be present as well when the object file is subsequently loaded. This solves most, but not all, problems of this nature, since most are related to the use of `define-syntax` and modules. Some problems are not so straightforwardly handled, however. For example, assume that the file contains the following definitions for `nodups?` and `mvlet`.

```
(define nodups?
  (lambda (ids)
    (define bound-id-member?
      (lambda (id ids)
        (and (not (null? ids))
```

```
            (or (bound-identifier=? id (car ids))
                (bound-id-member? id (cdr ids))))))
    (or (null? ids)
        (and (not (bound-id-member? (car ids) (cdr ids)))
             (nodups? (cdr ids))))))

(define-syntax mvlet
  (lambda (x)
    (syntax-case x ()
      [(_ ((x ...) expr) b1 b2 ...)
       (and (andmap identifier? #'(x ...))
            (nodups? #'(x ...)))
       #'(call-with-values
           (lambda () expr)
           (lambda (x ...) b1 b2 ...))])))

(mvlet ((a b c) (values 1 2 3))
  (list (* a a) (* b b) (* c c)))
```

When loaded directly, this results in the definition of `nodups?` as a procedure and `mvlet` as a syntactic abstraction before evaluation of the `mvlet` expression. Because `nodups?` is defined before the `mvlet` expression is expanded, the call to `nodups?` during the expansion of `mvlet` causes no difficulty. If instead this file were compiled, using `compile-file`, the compiler would arrange to define `mvlet` before continuing with the expansion and evaluation of the `mvlet` expression, but it would not arrange to define `nodups?`. Thus the expansion of the `mvlet` expression would fail.

In this case it does not help to evaluate the syntactic extension alone. A solution in this case would be to move the definition of `nodups?` inside the definition for `mvlet`, just as the definition for `bound-id-member?` is placed within `nodups?`, but this does not work for help routines shared among several syntactic definitions. Another solution is to label the `nodups?` definition a "meta" definition (see Section 11.8) but this does not work for helpers that are used both by syntactic abstractions and by run-time code.

A somewhat simpler problem occurs when setting parameters that affect compilation, such as `optimize-level` and `case-sensitive?`. If not set prior to compilation, their settings usually will not have the desired effect.

`eval-when` offers a solution to these problems by allowing the programmer to explicitly control what forms should or should not be evaluated during compilation. `eval-when` is a syntactic form and is handled directly by the expander. The action of `eval-when` depends upon the *situations* argument and whether or not the forms $form_1$ $form_2$ ... are being compiled via `compile-file` or are being evaluated directly. Let's consider each of the possible situation specifiers `eval`, `compile`, `load`, `visit`, and `revisit` in turn.

`eval`: The `eval` specifier is relevant only when the `eval-when` form is being evaluated directly, i.e., if it is typed at the keyboard or loaded from a source file. Its presence causes $form_1$ $form_2$ ... to be expanded and this expansion to be included in the expansion of the `eval-when` form. Thus, the forms will be evaluated directly as if not contained within an `eval-when` form.

`compile`: The `compile` specifier is relevant only when the `eval-when` form appears in a file currently being compiled. (Its presence is simply ignored otherwise.) Its presence forces $form_1$ $form_2$ ... to be expanded and evaluated immediately.

`load`: The `load` specifier is also relevant only when the `eval-when` form appears in a file currently being compiled. Its presence causes $form_1$ $form_2$ ... to be expanded and this expansion to be included in the expansion of the `eval-when` form. Any code necessary to record binding information and evaluate syntax transformers for definitions contained in the forms is marked for execution when the file is "visited," and any code necessary to compute the values of variable definitions and the expressions contained within the forms is marked for execution when the file is "revisited."

`visit`: The `visit` specifier is also relevant only when the `eval-when` form appears in a file currently being compiled. Its presence causes $form_1$ $form_2$ ... to be expanded and this expansion to be included in the expansion of the `eval-when` form, with an annotation that the forms are to be executed when the file is "visited."

`revisit`: The `revisit` specifier is also relevant only when the `eval-when` form appears in a file currently being compiled. Its presence causes $form_1$ $form_2$ ... to be expanded and this expansion to be included in the expansion of the `eval-when` form, with an annotation that the forms are to be executed when the file is "revisited."

A file is considered "visited" when it is brought in by either `load` or `visit` and "revisited" when it is brought in by either `load` or `revisit`.

Top-level expressions are treated as if they are wrapped in an `eval-when` with situations `load` and `eval`. This means that, by default, forms typed at the keyboard or loaded from a source file are evaluated, and forms appearing in a file to be compiled are not evaluated directly but are compiled for execution when the resulting object file is subsequently loaded.

The treatment of top-level definitions is slightly more involved. All definitions result in changes to the compile-time environment. For example, an identifier defined by `define` is recorded as a variable, and an identifier defined by `define-syntax` is recorded as a keyword and associated with the value of its right-hand-side (transformer) expression. These changes are made at eval, compile, and load time as if the definitions were wrapped in an `eval-when` with situations `eval`, `load`, and `compile`. (This behavior can be altered by changing the value of the parameter `eval-syntax-expanders-when`.) Some definitions also result in changes to the run-time environment. For example, a variable is associated with the value of its right-hand-side expression. These changes are made just at evaluation and load time as if the definitions were wrapped in an `eval-when` with situations `eval` and `load`.

The treatment of local expressions or definitions (those not at top level) that are wrapped in an `eval-when` depends only upon whether the situation `eval` is present in the list of situations. If the situation `eval` is present, the definitions and expressions are evaluated as if they were not wrapped in an `eval-when` form, i.e., the `eval-when` form is treated as a `begin` form. If the situation `eval` is not present, the forms are ignored; in a definition context, the `eval-when` form is treated as an empty `begin`, and in an expression context, the `eval-when` form is treated as a constant with an unspecified value.

Since top-level syntax bindings are established, by default, at compile time as well as eval and load time, top-level variable bindings needed by syntax transformers should be wrapped in an `eval-when` form with situations `compile`, `load`, and `eval`. We can thus `nodups?` problem above by enclosing the definition of `nodups?` in an `eval-when` as follows.

```
(eval-when (compile load eval)
  (define nodups?
    (lambda (ids)
      (define bound-id-member?
        (lambda (id ids)
          (and (not (null? ids))
               (or (bound-identifier=? id (car ids))
                   (bound-id-member? id (cdr ids))))))
      (or (null? ids)
          (and (not (bound-id-member? (car ids) (cdr ids)))
               (nodups? (cdr ids)))))))
```

This forces it to be evaluated before it is needed during the expansion of the `mvlet` expression.

Just as it is useful to add `compile` to the default `load` and `eval` situations, omitting options is also useful. Omitting one or more of `compile`, `load`, and `eval` has the effect of preventing the evaluation at the given time. Omitting all of the options has the effect of inhibiting evaluation altogether.

One common combination of situations is `(compile eval)`, which by the inclusion of `compile` causes the expression to be evaluated at compile time, and by the omission of `load` inhibits the generation of code by the compiler for execution when the file is subsequently loaded. This is typically used for the definition of syntactic extensions used only within the file in which they appear; in this case their presence in the object file is not necessary. It is also used to set compilation parameters that are intended to be in effect whether the file is loaded from source or compiled via `compile-file`

```
(eval-when (compile eval) (case-sensitive #t))
```

Another common situations list is `(compile)`, which might be used to set compilation options to be used only when the file is compiled via `compile-file`.

```
(eval-when (compile) (optimize-level 3))
```

Finally, one other common combination is `(load eval)`, which might be useful for inhibiting the double evaluation (during the compilation of a file and again when the resulting object file is loaded) of syntax definitions when the syntactic extensions are not needed within the file in which their definitions appear.

The behavior of `eval-when` is usually intuitive but can be understood precisely as follows. The `syntax-case` expander, which handles `eval-when` forms, maintains two state sets, one for compile-time forms and one for run-time forms. The set of possible states in each set are "L" for `load`, "C" for `compile`, "V" for `visit`, "R" for `revisit`, and "E" for `eval`.

When compiling a file, the compile-time set initially contains "L" and "C" and the run-

time set initially contains only "L." When not compiling a file (as when a form is evaluated by the read-eval-print loop or loaded from a source file), both sets initially contain only "E." The subforms of an `eval-when` form at top level are expanded with new compile- and run-time sets determined by the current sets and the situations listed in the `eval-when` form. Each element of the current set contributes zero or more elements to the new set depending upon the given situations according to the following table.

	`load`	`compile`	`visit`	`revisit`	`eval`
L	L	C	V	R	—
C	—	—	—	—	C
V	V	C	V	—	—
R	R	C	—	R	—
E	—	—	—	—	E

For example, if the current compile-time state set is {L} and the situations are `load` and `compile`, the new compile-time state set is {L, C}, since L/`load` contributes "L" and L/`compile` contributes "C."

The state sets determine how forms are treated by the expander. Compile-time forms such as syntax definitions are evaluated at a time or times determined by the compile-time state set, and run-time forms are evaluated at a time or times determined by the run-time state set. A form is evaluated immediately if "C" is in the state set. Code is generated to evaluate the form at visit or revisit time if "V" or "R" is present. If "L" is present in the compile-time set, it is treated as "V;" likewise, if "L" is present in the run-time set, it is treated as "R." If more than one of states is present in the state set, the form is evaluated at each specified time.

"E" can appear in the state set only when not compiling a file, i.e., when the expander is invoked from an evaluator such as `compile` or `interpret`. When it does appear, the expanded form is returned from the expander to be processed by the evaluator, e.g., `compile` or `interpret`, that invoked the expander.

The value of the parameter `eval-syntax-expanders-when` actually determines the initial compile-time state set. The parameter is bound to a list of situations, which defaults to `(compile load eval)`. When compiling a file, `compile` contributes "C" to the state set, `load` contributes "L," `visit` contributes "V," `revisit` contributes "R," and `eval` contributes nothing. When not compiling a file, `eval` contributes "E" to the state set, and the other situations contribute nothing. There is no corresponding parameter for controlling the initial value of the run-time state set.

For RNRS top-level programs, `eval-when` is essentially ineffective. The entire program is treated as a single expression, so `eval-when` becomes a local `eval-when` for which only the `eval` situation has any relevance. As for any local `eval-when` form, the subforms are ignored if the `eval` situation is not present; otherwise, they are treated as if the `eval-when` wrapper were absent.

`eval-syntax-expanders-when` **thread parameter**
libraries: `(chezscheme)`

This parameter must be set to a list representing a set of `eval-when` situations, e.g., a

list containing at most one occurrence of each of the symbols `eval`, `compile`, `load`, `visit`, and `revisit`. It is used to determine the evaluation time of syntax definitions, module forms, and import forms are expanded. (See the discussion of `eval-when` above.) The default value is `(compile load eval)`, which causes compile-time information in a file to be established when the file is loaded from source, when it is compiled via `compile-file`, and when a compiled version of the file is loaded via `load` or `visit`.

12.5. Source Directories and Files

`source-directories` **global parameter**
libraries: `(chezscheme)`

The value of `source-directories` must be a list of strings, each of which names a directory path. `source-directories` determines the set of directories searched for source or object files when a file is loaded via `load`, `load-library`, `load-program`, `include`, `visit`, or `revisit`, when a syntax error occurs, or when a source file is opened in the interactive inspector.

The default value is the list `(".")`, which means source files will be found only in or relative to the current directory, unless named with an absolute path.

This parameter is never altered by the system, with one exception. The expander temporarily adds (via `parameterize`) the directory in which a library file resides to the front of the `source-directories` list when it compiles (when `compile-imported-libraries` is true) or loads the library from source, which it does only if the library is not already defined.

`(with-source-path` *who name procedure*`)` **procedure**
libraries: `(chezscheme)`

The procedure `with-source-path` searches through the current source-directories path, in order, for a file with the specified *name* and invokes *procedure* on the result. If no such file is found, an exception is raised with condition types `&assertion` and `&who` with *who* as who value.

If *name* is an absolute pathname or one beginning with `./` (or `.\` under Windows) or `../` (or `..\` under Windows), or if the list of source directories contains only `"."`, the default, or `""`, which is equivalent to `"."`, no searching is performed and *name* is returned.

who must be a symbol, *name* must be a string, and *procedure* should accept one argument.

The following examples assumes that the file "pie" exists in the directory "../spam" but not in "../ham" or the current directory.

```
(define find-file
  (lambda (fn)
    (with-source-path 'find-file fn values)))

(find-file "pie")  ⇒  "pie"
```

```
(source-directories '("." "../ham"))
(find-file "pie")  ⇒  exception in find-file: pie not found

(source-directories '("." "../spam"))
(find-file "pie")  ⇒  "../spam/pie"

(source-directories '("." "../ham"))
(find-file "/pie")  ⇒  "/pie"

(source-directories '("." "../ham"))
(find-file "./pie")  ⇒  "./pie"

(source-directories '("." "../spam"))
(find-file "../pie")  ⇒  "../ham/pie"
```

12.6. Compiler Controls

`optimize-level` **thread parameter**

libraries: `(chezscheme)`

This parameter can take on one of the four values 0, 1, 2, and 3.

In theory, this parameter controls the amount of optimization performed by the compiler. In practice, it does so only indirectly, and the only difference is between optimize level 3, at which the compiler generates "unsafe" code, and optimize levels 0–2, at which the compiler generates "safe" code. Safe code performs full type and bounds checking so that, for example, an attempt to apply a non-procedure, an attempt to take the car of a non-pair, or an attempt to reference beyond the end of a vector each result in an exception being raised. With unsafe code, the same situations may result in invalid memory references, corruption of the Scheme heap (which may cause seemingly unrelated problems later), system crashes, or other undesirable behaviors. Unsafe code is typically faster, but optimize-level 3 should be used with caution and only on sections of well-tested code that must run as quickly as possible.

While the compiler produces the same code for optimize levels 0–2, user-defined macro transformers can differentiate among the different levels if desired.

One way to use optimize levels is on a per-file basis, using `eval-when` to force the use of a particular optimize level at compile time. For example, placing:

```
(eval-when (compile) (optimize-level 3))
```

at the front of a file will cause all of the forms in the file to be compiled at optimize level 3 when the file is compiled (using `compile-file`) but does not affect the optimize level used when the file is loaded from source. Since `compile-file` parameterizes `optimize-level` (see `parameterize`), the above expression does not permanently alter the optimize level in the system in which the `compile-file` is performed.

The optimize level can also be set via the `--optimize-level` command-line option (Section 2.9). This option is particularly useful for running RNRS top-level programs at

optimize-level 3 via the `--program` command-line option, since `eval-when` is ineffective for RNRS top-level programs as described on page 345.

`($primitive` *variable*`)` **syntax**
`#%`*variable* **syntax**
`($primitive 2` *variable*`)` **syntax**
`#2%`*variable* **syntax**
`($primitive 3` *variable*`)` **syntax**
`#3%`*variable* **syntax**
returns: the primitive value for *variable*
libraries: `(chezscheme)`

variable must name a primitive procedure. The `$primitive` syntactic form allows control over the optimize level at the granularity of individual primitive references, and it can be used to access the original value of a primitive, regardless of the lexical context or the current top-level binding for the variable originally bound to the primitive.

The expression `($primitive` *variable*`)` may be abbreviated as `#%`*variable*. The reader expands `#%` followed by an object into a `$primitive` expression, much as it expands `'`*object* into a `quote` expression.

If a `2` or `3` appears in the form or between the `#` and `%` in the abbreviated form, the compiler treats an application of the primitive as if it were compiled at the corresponding optimize level (see the `optimize-level` parameter). If no number appears in the form, an application of the primitive is treated as an optimize-level 3 application if the current optimize level is 3; otherwise, it is treated as an optimize-level 2 application.

```
(#%car '(a b c))  ⇒  a
(let ([car cdr]) (car '(a b c)))  ⇒  (b c)
(let ([car cdr]) (#%car '(a b c)))  ⇒  a
(begin (set! car cdr) (#%car '(a b c)))  ⇒  a
```

`debug-level` **thread parameter**
libraries: `(chezscheme)`

This parameter can take on one of the four values 0, 1, 2, and 3. It is used to tell the compiler how important the preservation of debugging information is, with 0 being least important and 3 being most important. The default value is 1. As of Version 9.0, it is used solely to determine whether an error-causing call encountered in nontail position is treated as if it were in tail position (thus causing the caller's frame not to appear in a stack backtrace); this occurs at debug levels below 2.

`generate-interrupt-trap` **thread parameter**
libraries: `(chezscheme)`

To support interrupts, including keyboard, timer, and collect request interrupts, the compiler inserts a short sequence of instructions at the entry to each nonleaf procedure (Section 12.2). This small overhead may be eliminated by setting `generate-interrupt-trap` to `#f`. The default value of this parameter is `#t`.

It is rarely a good idea to compile code without interrupt trap generation, since a tight loop in the generated code may completely prevent interrupts from being serviced, including the collect request interrupt that causes garbage collections to occur automatically. Disabling trap generation may be useful, however, for routines that act simply as "wrappers" for other routines for which code is presumably generated with interrupt trap generation enabled. It may also be useful for short performance-critical routines with embedded loops or recursions that are known to be short running and that make no other calls.

`compile-interpret-simple` **thread parameter**
libraries: `(chezscheme)`

At all optimize levels, when the value of `compile-interpret-simple` is set to a true value (the default), `compile` interprets simple expressions. A simple expression is one that creates no procedures. This can save a significant amount of time over the course of many calls to `compile` or `eval` (with `current-eval` set to `compile`, its default value). When set to false, `compile` compiles all expressions.

`generate-inspector-information` **thread parameter**
libraries: `(chezscheme)`

When this parameter is set to a true value (the default), information about the source and contents of procedures and continuations is generated during compilation and retained in tables associated with each code segment. This information allows the inspector to provide more complete information, at the expense of using more memory and producing larger object files (via `compile-file`). Although compilation and loading may be slower when inspector information is generated, the speed of the compiled code is not affected. If this parameter is changed during the compilation of a file, the original value will be restored. For example, if:

```
(eval-when (compile) (generate-inspector-information #f))
```

is included in a file, generation of inspector information will be disabled only for the remainder of that particular file.

`enable-cross-library-optimization` **thread parameter**
libraries: `(chezscheme)`

This parameter controls whether information is included with the object code for a compiled library to enable propagation of constants and inlining of procedures defined in the library into dependent libraries. When set to `#t` (the default), this information is included; when set to `#f`, the information is not included. Setting the parameter to `#f` potentially reduces the sizes of the resulting object files and the exposure of near-source information via the object file.

`generate-wpo-files` **thread parameter**
libraries: `(chezscheme)`

When this parameter is set to `#t` (the default is `#f`), `compile-file`, `compile-library`, `compile-program`, and `compile-script` produce whole-program optimization (wpo) files for use by `compile-whole-program`. The name of the `wpo` file is derived from the output-file name by replacing the object-file extension (normally `.so`) with `.wpo`, or adding the extension `.wpo` if the object filename has no extension or has the extension `.wpo`.

`compile-compressed` **thread parameter**
libraries: `(chezscheme)`

When this parameter is `#t`, the default, `compile-file`, `compile-library`, `compile-script`, `compile-program`, `compile-to-file`, `compile-whole-program`, and `strip-fasl-file` compress the object files they create.

`compile-file-message` **thread parameter**
libraries: `(chezscheme)`

When this parameter is set to true, the default, `compile-file`, `compile-library`, `compile-program`, and `compile-script` print a message of the form:

`compiling` *input-path* `with output to` *output-path*

When the parameter is set to `#f`, the message is not printed.

`run-cp0` **thread parameter**
`cp0-effort-limit` **thread parameter**
`cp0-score-limit` **thread parameter**
`cp0-outer-unroll-limit` **thread parameter**
libraries: `(chezscheme)`

These parameters control the operation of `cp0`, a source optimization pass that runs after macro expansion and prior to most other compiler passes. `cp0` performs procedure inlining, in which the code of one procedure is inlined at points where it is called by other procedures, as well as copy propagation, constant folding, useless code elimination, and several related optimizations. The algorithm used by the optimizer is described in detail in the paper "Fast and effective procedure inlining" [31].

When `cp0` is enabled, the programmer can count on the compiler to fold constants, eliminate unnecessary `let` bindings, and eliminate unnecessary and inaccessible code. This is particularly useful when writing macros, since the programmer can usually handle only the general case and let the compiler simplify the code when possible. For example, the programmer can define `case` as follows:

```
(define-syntax case
  (syntax-rules ()
    [(_ e [(k ...) a1 a2 ...] ... [else b1 b2 ...])
     (let ([t e])
       (cond
         [(memv t '(k ...)) a1 a2 ...]
         ...
         [else b1 b2 ...]))]
    [(_ e [(k ...) a1 a2 ...] ...)
     (let ([t e])
       (cond
         [(memv t '(k ...)) a1 a2 ...]
         ...))]))
```

and count on the introduce `let` expression to be eliminated if `e` turns out to be an unassigned variable, and count on the entire `case` expression to be folded if `e` turns out to be a constant.

It is possible to see what `cp0` does with an expression via the procedure `expand/optimize`, which expands its argument and passes the result through `cp0`, as illustrated by the following transcript.

```
> (print-gensym #f)
> (expand/optimize
    '(lambda (x)
       (case x [(a) 1] [(b c) 2] [(d) 3] [else 4])))
(lambda (x)
  (if (#2%memv x '(a))
      1
      (if (#2%memv x '(b c)) 2 (if (#2%memv x '(d)) 3 4))))
> (expand/optimize
    '(+ (let ([f (lambda (x)
                   (case x [(a) 1] [(b c) 2] [(d) 3] [else 4]))])
          (f 'b))
        15))
17
```

In the first example, the `let` expression produced by `case` is eliminated, and in the second, the entire expression is optimized down to the constant `17`. Although not shown by `expand/optimize`, the `memv` calls in the output code for the first example will be replaced by calls to the less expensive `eq?` by a later pass of the compiler. Additional examples are given in the description of `expand/optimize`.

The value of `run-cp0` must be a procedure. Whenever the compiler is invoked on a Scheme form, the value p of this parameter is called to determine whether and how `cp0` is run. p receives two arguments: *cp0*, the entry point into `cp0`, and x, the form being compiled. The default value of `run-cp0` simply invokes *cp0* on x, then *cp0* again on the result. The second run is useful in some cases because the first run may not eliminate bindings for certain variables that appear to be referenced but are not actually referenced after inlining. The marginal benefit of the second run is usually minimal, but so is the cost.

Interesting variants include

```
(run-cp0 (lambda (cp0 x) x))
```

which bypasses (disables) `cp0`, and

```
(run-cp0 (lambda (cp0 x) (cp0 x)))
```

which runs `cp0` just once.

The value of `cp0-effort-limit` determines the maximum amount of effort spent on each inlining attempt. The time spent optimizing a program is a linear function of this limit and the number of calls in the program's source, so small values for this parameter enforce a tighter bound on compile time. When set to zero, inlining is disabled except when the name of a procedure is referenced only once. The value of `cp0-score-limit` determines the maximum amount of code produced per inlining attempt. Small values for this parameter limit the amount of overall code expansion. These parameters must be set to nonnegative fixnum values.

The parameter `cp0-outer-unroll-limit` controls the amount of inlining performed by the optimizer for recursive procedures. With the parameter's value set to the default value of `0`, recursive procedures are not inlined. A nonzero value for the outer unroll limit allows calls external to a recursive procedure to be inlined. For example, the expression

```
(letrec ([fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))])
  (fact 10))
```

would be left unchanged with the outer unroll limit set to zero, but would be converted into

```
(letrec ([fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))])
  (* 10 (fact 9)))
```

with the outer unroll limit set to one.

Interesting effects can be had by varying several of these parameters at once. For example, setting the effort and outer unroll limits to large values and the score limit to `1` has the effect of inlining even complex recursive procedures whose values turn out to be constant at compile time without risking any code expansion. For example,

```
(letrec ([fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))])
  (fact 10))
```

would be reduced to `3628800`, but

```
(letrec ([fact (lambda (x) (if (zero? x) 1 (* x (fact (- x 1)))))])
  (fact z))
```

would be left unchanged, although the optimizer may take a while to reach this decision if the effort and outer unroll limits are large.

`commonization-level` **thread parameter**
libraries: `(chezscheme)`

After running the main source optimization pass (cp0) for the last time, the compiler optionally runs a *commonization* pass. The pass commonizes the code for lambda expressions that have identical structure by abstracting differences at certain leaves of the program, namely constants, references to unassigned variables, and references to primitives. The parameter `commonization-level` controls whether commonization is run and, if so, how aggressive it is. Its value must be a nonnegative exact integer ranging from 0 through 9. When the parameter is set to 0, the default, commonization is not run. Otherwise, higher values result in more commonization.

Commonization can undo some of the effects of cp0's inlining, can add run-time overhead, and can complicate debugging, particularly at higher commonization levels, which is why it is disabled by default. On the other hand, for macros or other meta programs that can generate large, mostly similar lambda expressions, enabling commonization can result in significant savings in object-code size and even reduce run-time overhead by making more efficient use of instruction caches.

`undefined-variable-warnings` **thread parameter**
libraries: `(chezscheme)`

When `undefined-variable-warnings` is set to `#t`, the compiler issues a warning message whenever it cannot determine that a variable bound by `letrec`, `letrec*`, or an internal definition will not be referenced before it is defined. The default value is `#f`.

Regardless of the setting of this parameter, the compiler inserts code to check for the error, except at optimize level 3. The check is fairly inexpensive and does not typically inhibit inlining or other optimizations. In code that must be carefully tuned, however, it is sometimes useful to reorder bindings or make other changes to eliminate the checks. Enabling undefined-variable warnings can facilitate this process.

The checks are also visible in the output of `expand/optimize`.

`expand-output` **thread parameter**
`expand/optimize-output` **thread parameter**
libraries: `(chezscheme)`

The parameters `expand-output` and `expand/optimize-output` can be used to request that the compiler and interpreter print expander and source-optimizer output produced during the compilation or interpretation process. Each parameter must be set to either `#f` (the default) or a textual output port.

When `expand-output` is set to a textual output port, the output of the expander is printed to the port as a side effect of running `compile`, `interpret`, or any of the file compiling primitives, e.g., `compile-file` or `compile-library`. Similarly, when `expand/optimize-output` is set to a textual output port, the output of the source optimizer is printed.

See also `expand` (page 339) and `expand-optimize` (page 340), which can be used to run the expander or the expander and source optimizer directly on an individual form.

`(pariah` *expr*$_1$ *expr*$_2$ `...)` **syntax**
returns: the values of the last subexpression
libraries: `(chezscheme)`

A `pariah` expression is just like a `begin` expression except that it informs the compiler that the code is expected to be executed infrequently. The compiler uses this information to optimize code layout, register assignments, and other aspects of the generated code. The `pariah` form can be used in performance-critical code to mark the branches of a conditional (e.g., `if`, `cond`, or `case`) that are less likely to be executed than the others.

12.7. Profiling

ChezScheme supports two forms of profiling: source profiling and block profiling. With source profiling enabled, the compiler instruments the code it produces to count the number of times each source-code expression is executed. This information can be displayed in HTML format or packaged in a list for arbitrary user-defined processing. It can also be dumped to a file to be loaded subsequently into the compiler's database of profile information for use in source-level optimizations, such as reordering the clauses of a `case` or `exclusive-cond` form.

The association between source-code expressions and profile counts is usually established via annotations produced by the reader and present in the input to the expander (Section 11.11). It is also possible to explicitly identify source positions to be assigned profile counts via `profile` expressions. A `profile` expression has one subform, a source object, and returns an unspecified value. Its only effect is to cause the number of times the expression is executed to be accounted to the source object.

In cases where source positions explicitly identified by `profile` forms are the only ones whose execution counts should be tracked, the parameter `generate-profile-forms` can be set to `#f` to inhibit the expander's implicit generation of `profile` forms for all annotated source expressions. It is also possible to obtain finer control over implicit generation of `profile` forms by marking which annotations that should and should not be used for profiling (Section 11.11).

With block profiling enabled, the compiler similarly instruments the code it produces to count the number of times each "basic block" in the code it produces is executed. Basic blocks are the building blocks of the code produced by many compilers, including *Chez Scheme*'s compiler, and are sequences of straight-line code entered only at the top and exited only at the bottom. Counting the number of times each basic block is executed is equivalent to counting the number of times each instruction is executed, but more efficient. Block-profile information cannot be viewed, but it can be dumped to a file to be loaded subsequently into the compiler's database of profile information for use in block- and instruction-level optimizations. These optimizations include reordering blocks to push less frequently used sequences of code out-of-line, so they will not occupy space in the instruction cache, and giving registers to variables that are used in more frequently executed instructions.

Source profiling involves at least the following steps:

- compile the code with source profiling enabled,
- run the compiled code to generate source-profile information, and
- dump the profile information.

Source profiling is enabled by setting the parameter `compile-profile` to the symbol `source` or to the boolean value `#t`. The profile information can be dumped via:

`profile-dump-html` in HTML format to allow the programmer to visualize how often each expression is executed using a color-coding system that makes it easy to spot "hot spots,"

`profile-dump-list` in a form suitable for user-defined post-processing,

`profile-dump` in a form suitable for off-line processing by one of the methods above or by some custom means, or

`profile-dump-data` in a form suitable for loading into the compiler's database.

If the information is intended to be fed back into the compiler for optimization, the following additional steps are required, either in the same or a different Scheme process:

- load the profile information into the compiler's profile database, and
- recompile the code.

Profile information dumped by `profile-dump-data` is loaded into the compiler's profile database via `profile-load-data`. Profiling information is *not* available to the compiler unless it is explicitly dumped via `profile-dump-data` and loaded via `profile-load-data`.

When block-profile information is to be used for optimization, the steps are similar:

- compile the code with block profiling enabled,
- run the code to generate block-profile information,
- dump the profile information,
- load the profile information, and
- recompile the code.

Block profiling is enabled by setting the parameter `compile-profile` to the symbol `block` or to the boolean value `#t`. The profile information must be dumped via `profile-dump-data` and loaded via `profile-load-data`. As with source profile information, block profile information can be loaded in the same or in a different Scheme process as the one that dumped the information.

For block optimization, the code to be recompiled must be identical. In general, this means the files involved must not have been modified, and nothing else can change that indirectly affects the code produced by the compiler, e.g., settings for compiler parameters such as `optimize-level` or the contents of configuration files read by macros at compile time. Otherwise, the set of blocks or the instructions within them might be different, in which case the block profile information will not line up properly and the compiler will raise an exception.

For the same reason, when both source profiling and block profiling information is to be used for optimization, the source information must be gathered first and loaded before both the first and second compilation runs involved in block profiling. That is, the following steps must be used:

1 compile the code with source profiling enabled,

2 run the code to generate source-profile information,

2 dump the source-profile information,

3 load the source-profile information,

3 recompile the code with block profiling enabled,

4 run the code to generate block-profile information,

4 dump the block-profile information,

5 load the source- and block-profile information, and

5 recompile the code.

The numbers labeling each step indicate both the order of the steps and those that must be performed in the same Scheme process. (All of the steps can be performed in the same Scheme process, if desired.)

Both source and block profiling are disabled when `compile-profile` is set to `#f`, its default value.

The following example highlights the use of source profiling for identifying hot spots in the code. Let's assume that the file /tmp/fatfib/fatfib.ss contains the following source code.

```
(define fat+
  (lambda (x y)
    (if (zero? y)
        x
        (fat+ (1+ x) (1- y)))))

(define fatfib
  (lambda (x)
    (if (< x 2)
        1
        (fat+ (fatfib (1- x)) (fatfib (1- (1- x)))))))
```

We can load fatfib.ss with profiling enabled as follows.

```
(parameterize ([compile-profile 'source])
  (load "/tmp/fatfib/fatfib.ss"))
```

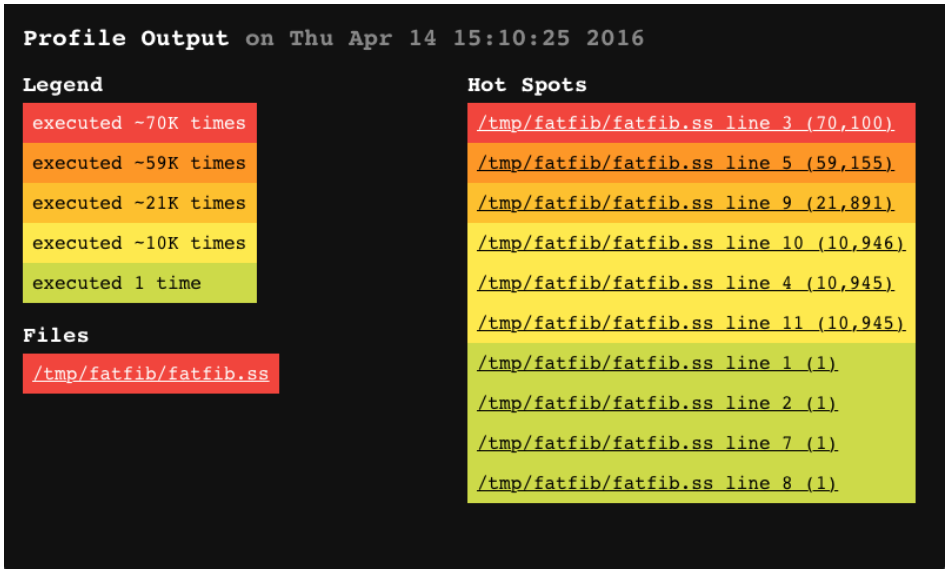
We then run the application as usual.

```
(fatfib 20)  ⇒  10946
```

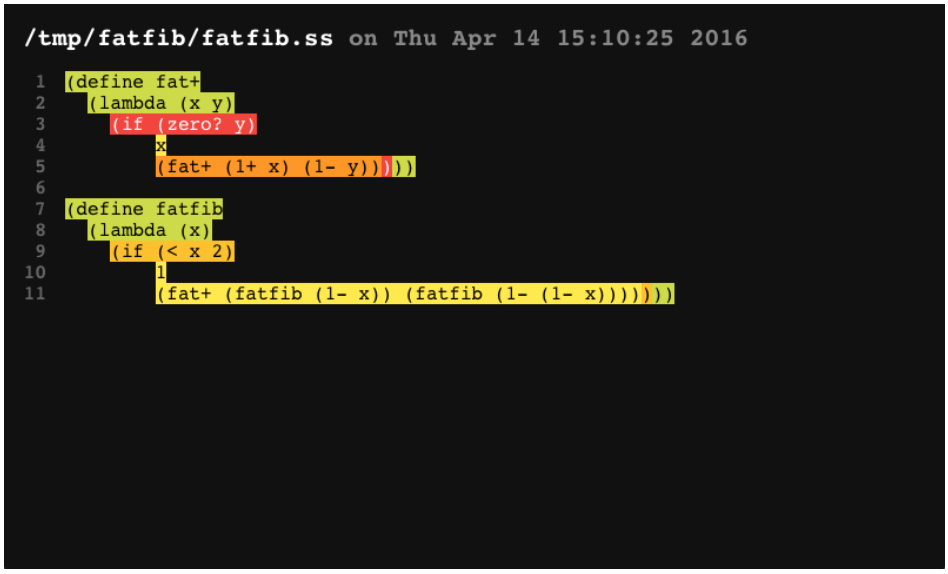
After the run (or multiple runs), we dump the profile information as a set of html files using `profile-dump-html`.

```
(profile-dump-html)
```

This creates a file named profile.html containing a summary of the profile information gathered during the run. If we view this file in a browser, we should see something like the following.



The most frequently executed code is highlighted in colors closer to red in the visible spectrum, while the least frequently executed code is highlighted in colors closer to violet. Each of the entries in the lists of files and hot spots are links into additional generated files, one per source file (provided `profile-dump-html` was able to locate an unmodified copy of the source file). In this case, there is only one, fatfib.ss.html. If we move to that file, we should see something like this:



As in the summary, the code is color-coded according to frequency of execution. Hovering over a color-coded section of code should cause a pop-up box to appear with the starting position and count of the source expression. If a portion of source code is not color-coded or is identified via the starting position as having inherited its color from some enclosing expression, it may have been recognized as dead code by the compiler or garbage collector and discarded, or the expander might not have been able to track it through the macro-expansion process.

`profile-dump` and `profile-dump-list` may be used to generate a list of profile entries, which may then be analyzed manually or via a custom profile-viewing application.

`compile-profile` **thread parameter**
libraries: `(chezscheme)`

When this parameter is set to the symbol `source` or the boolean value `#t`, the compiler instruments the code it generates with instructions that count the number of times each section of source code is executed. When set to the symbol `block`, the compiler similarly instruments the code it generates with instructions that count the number of times each block of code is executed. When set to `#f` (the default), the compiler does not insert these instructions.

The general description of profiling above describes how the source and block profile information can be viewed or used for optimization.

The code generated when `compile-profile` is non-false is larger and less efficient, so this parameter should be set only when profile information is needed.

`(profile` *source-object*`)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

A `profile` form has the effect of accounting to the source position identified by *source-object* the number of times the `profile` form is executed. Profile forms are generated implicitly by the expander for source expressions in annorated input, e.g., input read by the compiler or interpreter from a Scheme source file, so this form is typically useful only when unannotated source code is produced by the front end for some language that targets Scheme.

`(generate-profile-forms)` **thread parameter**
libraries: `(chezscheme)`

When this parameter is set to `#t`, the default, the expander implicitly introduces `profile` forms for each annotated input expression, unless the annotation has not been marked for use in profiling (Section 11.11). It can be set to `#f` to inhibit the expander's implicit generation of `profile` forms, typically when explicit `profile` forms are already present for all source positions that should be profiled.

`(profile-clear)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

Calling this procedure causes profile information to be cleared, i.e., the counts associated with each section of code are set to zero.

`(profile-dump)` **procedure**
returns: a list of pairs of source-object and count
libraries: `(chezscheme)`

This procedure produces a dump of all profile information gathered since startup or the last call to `profile-clear`. It returns a list of pairs, where the car of each pair is a source object (Section 11.11) and the cdr is an exact nonnegative integer count.

The list might contain more than one entry per source object due to macro expansion and procedure inlining, and it might contain more than one (non-eq) source object per file and source position due to separate compilation. In such cases, the counts are not overlapping and can be summed together to obtain the full count.

The advantage of `profile-dump` over `profile-dump-list` is that `profile-dump` performs only minimal processing and preserves complete source objects, including their embedded source-file descriptors. It might be used, for example, to dump profile information to a fasl file on one machine for subsequent processing on another.

(profile-dump-html) **procedure**
(profile-dump-html *prefix*) **procedure**
(profile-dump-html *prefix dump*) **procedure**
returns: unspecified
libraries: (chezscheme)

This procedure produces one or more HTML files, including profile.html, which contains color-coded summary information, and one file *source*.html for each source file *source* containing a color-coded copy of the source code, as described in the lead-in to this section. If *prefix* is specified, it must be a string and is prepended to the names of the generated HTML files. For example, if *prefix* is `"/tmp/"`, the generated files are placed in the directory /tmp. The raw profile information is obtained from *dump*, which defaults to the value returned by `profile-dump`.

(profile-palette) **thread parameter**
libraries: (chezscheme)

This value of this parameter must be a nonempty vector of at least three pairs. The car of each pair is a background color and the cdr is a foreground (text) color. Each color must be a string, and each string should contain an HTML cascading style sheet (CSS) color specifier. The first pair is used for unprofiled code, and the second is used for unexecuted profiled code. The third is used for code that is executed least frequently, the fourth for code executed next-least frequently, and so on, with the last being used for code that is executed most frequently. Programmers may wish to supply their own palette to enhance visibility or to change the number of colors used.

By default, a black background is used for unprofiled code, and a gray background is used for unexecuted profiled code. Background colors ranging from purple to red are used for executed profiled code, depending on frequency of execution, with red for the most frequently executed code.

```
(profile-palette) ⇒
  #(("#111111" . "white") ("#607D8B" . "white")
    ("#9C27B0" . "black") ("#673AB7" . "white")
    ("#3F51B5" . "white") ("#2196F3" . "black")
    ("#00BCD4" . "black") ("#4CAF50" . "black")
    ("#CDDC39" . "black") ("#FFEB3B" . "black")
    ("#FFC107" . "black") ("#FF9800" . "black")
    ("#F44336" . "white"))
(profile-palette
 ; set palette with rainbow colors and black text
 ; for all but unprofiled or unexecuted code
  '#(("#000000" . "white")    ; black
     ("#666666" . "white")    ; gray
     ("#8B00FF" . "black")    ; violet
     ("#6600FF" . "black")    ; indigo
     ("#0000FF" . "black")    ; blue
     ("#00FF00" . "black")    ; green
     ("#FFFF00" . "black")    ; yellow
```

```
("#FF7F00" . "black")    ; orange
("#FF0000" . "black")))  ; red
```

(profile-line-number-color) **thread parameter**
libraries: (chezscheme)

This value of this parameter must be a string or `#f`. If it is a string, the string should contain an HTML cascading style sheet (CSS) color specifier. If the parameter is set to string, `profile-dump-html` includes line numbers in its html rendering of each source file, using the specified color. If the parameter is set to `#f`, no line numbers are included.

(profile-dump-list) **procedure**
(profile-dump-list *warn?*) **procedure**
(profile-dump-list *warn? dump*) **procedure**
returns: a list of profile entries (see below)
libraries: (chezscheme)

This procedure produces a dump of all profile information present in *dump*, which defaults to the value returned by `profile-dump`. It returns a list of entries, each of which is itself a list containing the following elements identify one block of code and how many times it has been executed.

- execution count
- pathname
- beginning file position in characters (inclusive)
- ending file position in characters (exclusive)
- line number of beginning file position
- character position of beginning file position

`profile-dump-list` may be unable to locate an unmodified copy of the file in the current source directories or at the absolute address, if an absolute address was used when the file was compiled or loaded. If this happens, the line number and character position of the beginning file position are `#f` and the pathname is the pathname originally used. A warning is also issued (an exception with condition type `&warning` is raised) unless the `warn?` argument is provided and is false.

Otherwise, the pathname is the path to an unmodified copy of the source and the line and character positions are set to exact nonnegative integers.

In either case, the execution count, beginning file position, and ending file position are all exact nonnegative integers, and the pathname is a string.

For source positions in files that cannot be found, the list might contain more than one entry per position due to macro expansion, procedure inlining, and separate compilation. In such cases, the counts are not overlapping and can be summed together to obtain the full count.

The information returned by `profile-dump-list` can be used to implement a custom viewer or used as input for offline analysis of profile information.

The advantage of `profile-dump-list` over `profile-dump` is that it attempts to determine the line number and character position for each source point and, if successful, aggregates multiple counts for the source point into a single entry.

`(profile-dump-data` *path*`)` **procedure**
`(profile-dump-data` *path dump*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string.

This procedure writes, in a machine-readable form consumable by `profile-load-data`, profile counts represented by *dump* to the file named by *path*, replacing the file if it already exists. *dump* defaults to the value returned by `profile-dump`.

`(profile-load-data` *path* `...)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

Each *path* must be a string.

This procedure reads profile information from the files named by *path* ... and stores it in the compiler's internal database of profile information. The contents of the files must have been created originally by `profile-dump-data` using the same version of *Chez Scheme*.

The database stores a weight for each source expression or block rather than the actual count. When a single file is loaded into the database, the weight is the proportion of the actual count over the maximum count for all expressions or blocks represented in the file. When more than one file is loaded, either by one or multiple calls to `profile-load-data`, the weights are averaged.

`(profile-query-weight` *obj*`)` **procedure**
returns: *obj*'s profile weight, or `#f` if *obj* is not in the database
libraries: `(chezscheme)`

The compiler's profile database maps source objects (Section 11.11) to weights. If *obj* is a source object, the `profile-query-weight` returns the weight associated with the source object or `#f` if the database does not have a weight recorded for the source object. *obj* can also be an annotation or syntax object, in which case `profile-query-weight` first extracts the source object, if any, using `syntax->annotation` and `annotation-source`, returning `#f` if no source-object is found.

A weight is a flonum in the range 0.0 to 1.0, inclusive, and denotes the ratio of the actual count to the maximum count as described in the description of `profile-load-data`.

`profile-query-weight` can be used by a macro to determine the relative frequency with which its subexpressions were executed in the run or runs that generated the information

in the database. This information can be used to guide the generation of code that is likely to be more efficient. For example, the `case` macro uses profile information, when available, to order the clauses so that those whose keys matched more frequently are tested before those whose keys matched less frequently.

`(profile-clear-database)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedure clears the compiler's profile database. It has no impact on the counts associated with individual sections of instrumented code; `profile-clear` can be used to reset those counts.

12.8. Waiter Customization

`(new-cafe)` **procedure**
`(new-cafe` *eval-proc*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

Chez Scheme interacts with the user through a *waiter*, or read-eval-print loop (REPL). The waiter operates within a context called a *café*. When the system starts up, the user is placed in a café and given a waiter. `new-cafe` opens a new Scheme café, stacked on top of the old one. In addition to starting the waiter, `new-cafe` sets up the café's reset and exit handlers (see `reset-handler` and `exit-handler`). Exiting a café resumes the continuation of the call to `new-cafe` that created the café. Exiting from the initial café leaves Scheme altogether. A café may be exited from either by an explicit call to `exit` or by receipt of end-of-file ("control-D" on Unix systems) in response to the waiter's prompt. In the former case, any values passed to `exit` are returned from `new-cafe`.

If the optional *eval-proc* argument is specified, *eval-proc* is used to evaluate forms entered from the console. Otherwise, the value of the parameter `current-eval` is used. *eval-proc* must accept one argument, the expression to evaluate.

Interesting values for *eval-proc* include `expand`, which causes the macro expanded value of each expression entered to be printed and `(lambda (x) x)`, which simply causes each expression entered to be printed. An arbitrary procedure of one argument may be used to facilitate testing of a program on a series of input values.

```
> (new-cafe (lambda (x) x))
>> 3
3
>> (a . (b . (c . ())))
(a b c)
```

```
(define sum
  (lambda (ls)
    (if (null? ls)
        0
        (+ (car ls) (sum (cdr ls))))))
> (new-cafe sum)
>> (1 2 3)
6
```

The default waiter reader (see `waiter-prompt-and-read`) displays the current waiter prompt (see `waiter-prompt-string`) to the current value of `console-output-port` and reads from the current value of `console-input-port`. The default waiter printer (see `waiter-write`) sends output to the current value of `console-output-port`. These parameters, along with `current-eval`, can be modified to change the behavior of the waiter.

`waiter-prompt-string` **thread parameter**
libraries: `(chezscheme)`

The value of `waiter-prompt-string` must be a string. It is used by the default waiter prompter (see the parameter `waiter-prompt-and-read`) to print a prompt. Nested cafés are marked by repeating the prompt string once for each nesting level.

```
> (waiter-prompt-string)
">"
> (waiter-prompt-string "%")
% (waiter-prompt-string)
"%"
% (new-cafe)
%% (waiter-prompt-string)
"%"
```

`waiter-prompt-and-read` **thread parameter**
libraries: `(chezscheme)`

`waiter-prompt-and-read` must be set to a procedure. It is used by the waiter to print a prompt and read an expression. The value of `waiter-prompt-and-read` is called by the waiter with a positive integer that indicates the café nesting level. It should return an expression to be evaluated by the current evaluator (see `new-cafe` and `current-eval`).

`(default-prompt-and-read` *level*`)` **procedure**
libraries: `(chezscheme)`

level must be a positive integer indicating the cafeé nesting level as described above.

This procedure is the default value of the `waiter-prompt-and-read` parameter whenever the expression editor (Section 2.2, Chapter 14) is *not* enabled. It might be defined as follows.

```
(define default-prompt-and-read
  (lambda (n)
    (unless (and (integer? n) (>= n 0))
       (assertion-violationf 'default-prompt-and-read
         "~s is not a nonnegative integer"
         n))
    (let ([prompt (waiter-prompt-string)])
      (unless (string=? prompt "")
        (do ([n n (- n 1)])
            ((= n 0)
             (write-char #\space (console-output-port))
             (flush-output-port (console-output-port)))
            (display prompt (console-output-port))))
      (let ([x (read (console-input-port))])
         (when (and (eof-object? x) (not (string=? prompt "")))
            (newline (console-output-port))
            (flush-output-port (console-output-port)))
         x))))
```

`waiter-write` **thread parameter**
libraries: `(chezscheme)`

The value of `waiter-write` must be a procedure. The waiter uses the value of `waiter-write` to print the results of each expression read and evaluated by the waiter. The following example installs a procedure equivalent to the default `waiter-write`:

```
(waiter-write
  (lambda (x)
    (unless (eq? x (void))
      (pretty-print x (console-output-port)))
    (flush-output-port (console-output-port))))
```

`(reset)` **procedure**
returns: does not return
libraries: `(chezscheme)`

`reset` invokes the current reset handler (see `reset-handler`) without arguments.

`reset-handler` **thread parameter**
libraries: `(chezscheme)`

The value of this parameter must be a procedure and should accept zero arguments. The current reset handler is called by `reset`. The default reset handler resets to the current café.

`(exit` *obj* `...)` **procedure**
returns: does not return
libraries: `(chezscheme)`

`exit` invokes the current exit handler (see `exit-handler`), passing along its arguments, if any.

`exit-handler` **thread parameter**
libraries: `(chezscheme)`

The value of this parameter must be a procedure and should accept any number of arguments. The current exit handler is called by `exit`.

The default exit handler exits from the current café, returning its arguments as the values of the call to `new-cafe` that created the current café. If the current café is the original café, or if `exit` is called from a script, `exit` exits from Scheme. In this case, the exit code for the Scheme process is 0 if no arguments were supplied or if the first argument is void, the value of the first argument if it is a 32-bit exact integer, and -1 otherwise.

`(abort)` **procedure**
`(abort` *obj*`)` **procedure**
returns: does not return
libraries: `(chezscheme)`

`abort` invokes the current abort handler (see `abort-handler`), passing along its argument, if any.

`abort-handler` **thread parameter**
libraries: `(chezscheme)`

The value of this parameter must be a procedure and should accept either zero arguments or one argument. The current abort handler is called by `abort`.

The default abort handler exits the Scheme process. The exit code for the Scheme process is -1 if no arguments were supplied, 0 if the first argument is void, the value of the first argument if it is a 32-bit exact integer, and -1 otherwise.

`scheme-start` **global parameter**
libraries: `(chezscheme)`

The value of `scheme-start` is a procedure that determines the system's action upon start-up. The procedure receives zero or more arguments, which are strings representing the file names (or command-line arguments not recognized by the Scheme executable) after given on the command line. The default value first loads the files named by the arguments, then starts up the initial café:

```
(lambda fns
  (for-each load fns)
  (new-cafe))
```

`scheme-start` may be altered to start up an application or to perform customization prior to normal system start-up.

To have any effect, this parameter must be set within a boot file. (See Chapter 2.)

`scheme-script` **global parameter**
libraries: `(chezscheme)`

The value of `scheme-script` is a procedure that determines the system's action upon start-up, when the `--script` option is used. The procedure receives one or more arguments. The first is a string identifying the script filename and the remainder are strings representing the remaining file names (or command-line arguments not recognized by the Scheme executable) given on the command line. The default value of this parameter is a procedure that sets the `command-line` and `command-line-arguments` parameters, loads the script using `load`, and returns void, which is translated into a 0 exit status for the script process.

```
(lambda (fn . fns)
  (command-line (cons fn fns))
  (command-line-arguments fns)
  (load fn))
```

`scheme-script` may be altered to start up an application or to perform customization prior to normal system start-up.

To have any effect, this parameter must be set within a boot file. (See Chapter 2.)

`scheme-program` **global parameter**
libraries: `(chezscheme)`

The value of `scheme-program` is a procedure that determines the system's action upon start-up when the `--program` (RNRS top-level program) option is used. The procedure receives one or more arguments. The first is a string identifying the program filename and the remainder are strings representing the remaining file names (or command-line arguments not recognized by the Scheme executable) given on the command line. The default value of this parameter is a procedure that sets the `command-line` and `command-line-arguments` parameters, loads the program using `load-program`, and returns void, which is translated into a 0 exit status for the script process.

```
(lambda (fn . fns)
  (command-line (cons fn fns))
  (command-line-arguments fns)
  (load-program fn))
```

`scheme-program` may be altered to start up an application or to perform customization prior to normal system start-up.

To have any effect, this parameter must be set within a boot file. (See Chapter 2.)

`command-line` **global parameter**
libraries: `(chezscheme)`

This parameter is set by the default values of `scheme-script` and `scheme-program` to a list representing the command line, with the script name followed by the command-line arguments, when the `--script` or `--program` option is used on system startup.

`command-line-arguments` **global parameter**
libraries: `(chezscheme)`

This parameter is set by the default values of `scheme-script` and `scheme-program` to a list of the command-line arguments when the `--script` or `--program` option is used on system startup.

`suppress-greeting` **global parameter**
libraries: `(chezscheme)`

The value of `suppress-greeting` is a boolean value that determines whether *Chez Scheme* prints an identifying banner and copyright notice. The parameter defaults to `#f` but may be set to `#t` for use in batch processing applications where the banner would be disruptive.

To have any effect, this parameter must be set within a boot file. (See Chapter 2.)

12.9. Transcript Files

A transcript file is a record of an interactive session. It is also useful as a "quick-and-dirty" alternative to opening an output file and using explicit output operations.

`(transcript-on` *path*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

path must be a string.

`transcript-on` opens the file named by *path* for output, and it copies to this file all input from the current input port and all output to the current output port. An exception is raised with condition-type `i/o-filename` if the file cannot be opened for output.

`(transcript-off)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`transcript-off` ends transcription and closes the transcript file.

`(transcript-cafe` *path*`)` **procedure**
libraries: `(chezscheme)`

path must be a string. `transcript-cafe` opens a transcript file as with `transcript-on` and enters a new café; exiting from this café (see `exit`) also ends transcription and closes the transcript file. Invoking `transcript-off` while in a transcript café ends transcription and closes the transcript file but does not cause an exit from the café.

12.10. Times and Dates

This section documents procedures for handling times and dates. Most of the procedures described here are proposed in SRFI 19: Time Data Types and Procedures, by Will Fitzgerald.

Times are represented by time objects. Time objects record the nanosecond and second of a particular time or duration, along with a *time type* that identifies the nature of the time object. The time type is one of the following symbols:

`time-utc`: The time elapsed since the "epoch:" 00:00:00 UTC, January 1, 1970, subject to adjustment, e.g., to correct for leap seconds.

`time-monotonic`: The time elapsed since some arbitrary point in the past, ideally not subject to adjustment.

`time-duration`: The time elapsed between two times. When used as an argument to `current-time`, it behaves like `time-monotonic`, but may also used to represent the result of subtracting two time objects.

`time-process`: The amount of CPU time used by the current process.

`time-thread`: The amount of CPU time used by the current thread. It is the same as `time-process` if not running threaded or if the system does not allow individual thread times to be determined.

`time-collector-cpu`: The portion of the current process's CPU time consumed by the garbage collector.

`time-collector-real`: The portion of the current process's real time consumed by the garbage collector.

A time-object second is an exact integer (possibly negative), and a nanosecond is an exact nonnegative integer less than 10^9. The second and nanosecond of a time object may be converted to an aggregate nanosecond value by scaling the seconds by 10^9 and adding the nanoseconds. Thus, if the second and nanosecond of a time object are 5 and 10, the time object represents 5000000010 nanoseconds (5.000000010 seconds). If the second and nanosecond are -5 and 10, the time object represents -4999999990 nanoseconds (-4.999999990 seconds).

Dates are represented by date objects. A date object records the nanosecond, second, minute, hour, day, month, and year of a particular date, along with an offset that identifies the time zone.

As for time objects, a nanosecond is an exact integer less than 10^9. A date-object second is, however, an exact nonnegative integer less than 62. (The values 61 and 62 allow for leap seconds.) A minute is an exact nonnegative integer less than 60, and an hour is an exact nonnegative integer less than 24. A day is an exact nonnegative integer in ranging from 1 representing the first day of the month to n, where n is the number of days in the date's month and year. A month is an exact nonnegative integer ranging from 1 through 12, where 1 represents January, 2 represents February, and so on. A year must be an exact integer. Years less than 1970 or greater than 2038 may not be supported depending on limitations of the underlying implementation. A time-zone offset represents the time-zone offset, in seconds, from UTC. It is an exact integer in the range −86400 to +86400, inclusive. For example, Eastern Standard Time (EST), which is 5 hours east, has offset $5 \times 3600 = -18000$. The offset for Eastern Daylight Time (EDT) is −14400. UTC is represented by offset zero.

`(current-time)` **procedure**
`(current-time` *time-type*`)` **procedure**
returns: a time object representing the current time
libraries: `(chezscheme)`

time-type must be one of the time-type symbols listed above and defaults to `time-utc`.

```
(current-time)  ⇒  #<time-utc 1198815722.473668000>
(current-time 'time-process)  ⇒  #<time-process 0.120534264>
```

`(make-time` *type nsec sec*`)` **procedure**
returns: a time object
libraries: `(chezscheme)`

type must be one of the time-type symbols listed above. *nsec* represents nanoseconds and must be an exact nonnegative integer less than 10^9. *sec* represents seconds and must be an exact integer.

```
(make-time 'time-utc 787511000 1198783214)
(make-time 'time-duration 10 5)
(make-time 'time-duration 10 -5)
```

`(time?` *obj*`)` **procedure**
returns: `#t` if *obj* is a time object, `#f` otherwise
libraries: `(chezscheme)`

```
(time? (current-time))  ⇒  #t
(time? (make-time 'time-utc 0 0))  ⇒  #t
(time? "1400 hours")  ⇒  #f
```

`(time-type` *time*`)` **procedure**
returns: the time type of *time*
`(time-nanosecond` *time*`)` **procedure**
returns: the nanosecond of *time*
`(time-second` *time*`)` **procedure**
returns: the second of *time*
libraries: `(chezscheme)`

time must be a time object.

```
(time-type (current-time))  ⇒  time-utc
(time-type (current-time 'time-process))  ⇒  time-process
(time-type (make-time 'time-duration 0 50))  ⇒  time-duration
(time-second (current-time))  ⇒  1198816497
(time-nanosecond (current-time))  ⇒  2399000
(time-second (make-time 'time-duration 10 -5))  ⇒  -5
(time-nanosecond (make-time 'time-duration 10 -5))  ⇒  10
```

`(set-time-type!` *time type*`)` **procedure**
returns: unspecified
`(set-time-nanosecond!` *time nsec*`)` **procedure**
returns: unspecified
`(set-time-second!` *time sec*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

time must be a time object. *type* must be one of the time-type symbols listed above. *nsec* represents nanoseconds and must be an exact nonnegative integer less than 10^9. *sec* represents seconds and must be an exact integer.

Each of these procedures modifies the time object, changing one aspect while leaving the others unaffected. For example, `set-time-nanosecond!` changes the nanosecond of *time* without changing the second or type. In particular, no conversion of values is performed when the type of a time object is changed.

`(time=?` $time_1$ $time_2$`)` **procedure**
`(time<?` $time_1$ $time_2$`)` **procedure**
`(time<=?` $time_1$ $time_2$`)` **procedure**
`(time>=?` $time_1$ $time_2$`)` **procedure**
`(time>?` $time_1$ $time_2$`)` **procedure**
returns: `#t` if the relation holds, `#f` otherwise
libraries: `(chezscheme)`

$time_1$ and $time_2$ must be time objects and must have the same type.

```
(let ([t (current-time)])
  (time=? t t))  ⇒  #t
(let ([t (current-time)])
  (let loop ()
    (when (time=? (current-time) t))
      (loop))
  (time>? (current-time) t))  ⇒  #t
```

`(copy-time` *time*`)` **procedure**
returns: a copy of *time*
libraries: `(chezscheme)`

```
(define t1 (current-time))
(define t2 (copy-time t1))
(eq? t2 t1)  ⇒  #f
(eqv? (time-second t2) (time-second t1))  ⇒  #t
(eqv? (time-nanosecond t2) (time-nanosecond t1))  ⇒  #t
```

`(time-difference` $time_1$ $time_2$`)` **procedure**
returns: the result of subtracting $time_2$ from $time_1$
`(time-difference!` $time_1$ $time_2$`)` **procedure**
returns: the result of subtracting $time_2$ from $time_1$
`(add-duration` *time* $time_d$`)` **procedure**
returns: the result of adding $time_d$ to `time`
`(add-duration!` *time* $time_d$`)` **procedure**
returns: the result of adding $time_d$ to `time`
`(subtract-duration` *time* $time_d$`)` **procedure**
returns: the result of subtracting $time_d$ from `time`
`(subtract-duration!` *time* $time_d$`)` **procedure**
returns: the result of subtracting $time_d$ from `time`
libraries: `(chezscheme)`

For `time-difference`, $time_1$ and $time_2$ must have the same time type, and the result is a time object with time type `time-duration`. For `add-duration`, `add-duration!`, `subtract-duration`, and `subtract-duration!`, $time_d$ must have time type `time-duration`, and the result is a time object with the same time type as *time*. `time-difference!`, `add-duration!`, and `subtract-duration!` are potentially destructive, i.e., each might modify and return its first argument, or it might allocate a new time object.

```
(let ([delay (make-time 'time-duration 0 1)])
  (let ([t1 (current-time 'time-monotonic)])
    (sleep delay)
    (let ([t2 (current-time 'time-monotonic)])
      (let ([t3 (time-difference t2 t1)])
        (and
          (eq? (time-type t3) 'time-duration)
          (time>=? t3 delay)
```

```
(time=? (add-duration t1 t3) t2)
(time=? (subtract-duration t2 t3) t1))))))  ⇒  #t
```

(current-date) **procedure**
(current-date *offset*) **procedure**
returns: a date object representing the current date
libraries: `(chezscheme)`

offset represents the time-zone offset in seconds east of UTC, as described above. It must be an exact integer in the range −86400 to +86400, inclusive and defaults to the local time-zone offset. UTC may be obtained by passing an offset of zero.

If *offset* is not provided, then the current time zone's offset is used, and `date-dst?` and `date-zone-name` report information about the time zone. If *offset* is provided, then `date-dst?` and `date-zone-name` on the resulting date object produce `#f`.

The following examples assume the local time zone is EST.

```
(current-date)  ⇒  #<date Thu Dec 27 23:23:20 2007>
(current-date 0)  ⇒  #<date Fri Dec 28 04:23:20 2007>

(date-zone-name (current-date))  ⇒  "EST" or other system-provided string
(date-zone-name (current-date 0))  ⇒  #f
```

(make-date *nsec sec min hour day mon year*) **procedure**
(make-date *nsec sec min hour day mon year offset*) **procedure**
returns: a date object
libraries: `(chezscheme)`

nsec represents nanoseconds and must be an exact nonnegative integer less than 10^9. *sec* represents seconds and must be an exact nonnegative integer less than 62. *min* represents minutes and must be an exact nonnegative integer less than 60. *hour* must be an exact nonnegative integer less than 24. *day* must be an exact integer, $1 \leq day \leq 31$. (The actual upper limit may be less depending on the month and year.) *mon* represents the month must be an exact integer, $1 \leq mon \leq 12$. *year* must be an exact integer. It should be at least 1970. *offset* represents the time-zone offset in seconds east of UTC, as described above. It must be an exact integer in the range −86400 to +86400, inclusive. UTC may be specified by passing an offset of zero.

If *offset* is not provided, then the current time zone's offset is used, and `date-dst?` and `date-zone-name` report information about the time zone. If *offset* is provided, then `date-dst?` and `date-zone-name` on the resulting date object produce `#f`.

```
(make-date 0 0 0 0 1 1 1970 0)  ⇒  #<date Thu Jan  1 00:00:00 1970>
(make-date 0 30 7 9 23 9 2007 -14400)  ⇒  #<date Sun Sep 23 09:07:30 2007>

(date-zone-name (make-date 0 30 7 9 23 9 2007 -14400))  ⇒  #f
(string? (date-zone-name (make-date 0 30 7 9 23 9 2007)))  ⇒  #t
```

`(date? obj)` **procedure**
returns: `#t` if *obj* is a date object, `#f` otherwise
libraries: `(chezscheme)`

```
(date? (current-date))
(date? (make-date 0 30 7 9 23 9 2007 -14400))
(date? "Sun Sep 23 09:07:30 2007")  ⇒  #f
```

`(date-nanosecond date)` **procedure**
returns: the nanosecond of *date*
`(date-second date)` **procedure**
returns: the second of *date*
`(date-minute date)` **procedure**
returns: the minute of *date*
`(date-hour date)` **procedure**
returns: the hour of *date*
`(date-day date)` **procedure**
returns: the day of *date*
`(date-month date)` **procedure**
returns: the month of *date*
`(date-year date)` **procedure**
returns: the year of *date*
`(date-zone-offset date)` **procedure**
returns: the time-zone offset of *date*
libraries: `(chezscheme)`

date must be a date object.

```
(define d (make-date 0 30 7 9 23 9 2007 -14400))
(date-nanosecond d)  ⇒  0
(date-second d)  ⇒  30
(date-minute d)  ⇒  7
(date-hour d)  ⇒  9
(date-day d)  ⇒  23
(date-month d)  ⇒  9
(date-year d)  ⇒  2007
(date-zone-offset d)  ⇒  -14400
```

`(date-week-day date)` **procedure**
returns: the week-day of *date*
`(date-year-day date)` **procedure**
returns: the year-day of *date*
libraries: `(chezscheme)`

These procedures allow the day-of-week or day-of-year to be determined for the date represented by *date*. A week-day is an exact nonnegative integer less than 7, where 0 represents Sunday, 1 represents Monday, and so on. A year-day is an exact nonnegative integer less

than 367, where 0 represents the first day of the year (January 1), 1 the second day, 2 the third, and so on.

```
(define d1 (make-date 0 0 0 0 1 1 1970 -18000))
d1  ⇒  #<date Thu Jan  1 00:00:00 1970>
(date-week-day d1)  ⇒  4
(date-year-day d1)  ⇒  0

(define d2 (make-date 0 30 7 9 23 9 2007 -14400))
d2  ⇒  #<date Sun Sep 23 09:07:30 2007>
(date-week-day d2)  ⇒  0
(date-year-day d2)  ⇒  265
```

`(date-dst?` *date*`)` **procedure**
returns: whether *date* is in Daylight Saving Time
`(date-zone-name` *date*`)` **procedure**
returns: `#f` or a string naming the time zone of *date*
libraries: `(chezscheme)`

These procedures report time-zone information for the date represented by *date* for a date object that is constructed without an explicit time-zone offset. When a date object is created instead with explicit time-zone offset, these procedures produce `#f`.

Daylight Saving Time status for the current time zone and a name string for the time zone are computed using platform-specific routines. In particular, the format of the zone name is platform-specific.

```
(define d (make-date 0 30 7 9 23 9 2007))
(date-zone-offset d)  ⇒  -14400 assuming Eastern U.S. time zone
(date-dst? d)  ⇒  #t
(date-zone-name d)  ⇒  "EDT" or some system-provided string
```

`(time-utc->date` *time*`)` **procedure**
`(time-utc->date` *time offset*`)` **procedure**
returns: a date object corresponding to *time*
`(date->time-utc` *date*`)` **procedure**
returns: a time object corresponding to *date*
libraries: `(chezscheme)`

These procedures are used to convert between time and date objects. The *time* argument to `time-utc->date` must have time-type `utc`, and `date->time-utc` always returns a time object with time-type `utc`.

For `time-utc->date`, *offset* represents the time-zone offset in seconds east of UTC, as described at the beginning of this section. It must be an exact integer in the range −86400 to +86400, inclusive and defaults to the local time-zone offset. UTC may be obtained by passing an offset of zero.

If *offset* is not provided to `time-utc->date`, then the current time zone's offset is used, and `date-dst?` and `date-zone-name` report information about the time zone. If *offset* is provided, then `date-dst?` and `date-zone-name` on the resulting date object produce `#f`.

```
(define d (make-date 0 30 7 9 23 9 2007 -14400))
(date->time-utc d)  ⇒  #<time-utc 1190552850.000000000>
(define t (make-time 'time-utc 0 1190552850))
(time-utc->date t)  ⇒  #<date Sun Sep 23 09:07:30 2007>
(time-utc->date t 0)  ⇒  #<date Sun Sep 23 13:07:30 2007>
```

`(date-and-time)` **procedure**
`(date-and-time` *date*`)` **procedure**
returns: a string giving the current date and time
libraries: `(chezscheme)`

The string is always in the format illustrated by the examples below and always has length 24.

```
(date-and-time)  ⇒  "Fri Jul 13 13:13:13 2001"
(define d (make-date 0 0 0 0 1 1 2007 0))
(date-and-time d)  ⇒  "Mon Jan 01 00:00:00 2007"
```

`(sleep` *time*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

time must be a time object with type `time-duration`. *sleep* causes the invoking thread to suspend operation for approximately the amount of time indicated by the time object, unless the process receives a signal that interrupts the sleep operation. The actual time slept depends on the granularity of the system clock and how busy the system is running other threads and processes.

12.11. Timing and Statistics

This section documents procedures for timing computations. The `current-time` procedure described in Section 12.10 may also be used to time computations.

`(time` *expr*`)` **syntax**
returns: the values of *expr*
libraries: `(chezscheme)`

`time` evaluates *expr* and, as a side-effect, prints (to the console-output port) the amount of cpu time, the amount of real time, the number of bytes allocated, and the amount of collection overhead associated with evaluating *expr*.

```
> (time (collect))
(time (collect))
    1 collection
    1 ms elapsed cpu time, including 1 ms collecting
    1 ms elapsed real time, including 1 ms collecting
    160 bytes allocated, including 8184 bytes reclaimed
```

`(display-statistics)` **procedure**
`(display-statistics` *textual-output-port*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

This procedure displays a running total of the amount of cpu time, real time, bytes allocated, and collection overhead. If *textual-output-port* is not supplied, it defaults to the current output port.

`(cpu-time)` **procedure**
returns: the amount of cpu time consumed since system start-up
libraries: `(chezscheme)`

The amount is in milliseconds. The amount includes "system" as well as "user" time, i.e., time spent in the kernel on behalf of the process as well as time spent in the process itself.

See also `current-time`, which returns more precise information.

`(real-time)` **procedure**
returns: the amount of real time that has elapsed since system start-up
libraries: `(chezscheme)`

The amount is in milliseconds.

See also `current-time`, which returns more precise information.

`(bytes-allocated)` **procedure**
`(bytes-allocated` *g*`)` **procedure**
returns: the number of bytes currently allocated
libraries: `(chezscheme)`

If *g* is supplied, `bytes-allocated` returns the number of bytes currently allocated for Scheme objects in the specified generation. *g* must be a nonnegative exact integer no greater than the maximum nonstatic generation, i.e., the value returned by `collect-maximum-generation`, or the symbol `static`. If *g* is not supplied, `bytes-allocated` returns the total number of bytes allocated in all generations.

`(initial-bytes-allocated)` **procedure**
returns: the total number of bytes allocated after loading boot files
libraries: `(chezscheme)`

`(bytes-deallocated)` **procedure**
returns: the total number of bytes deallocated by the garbage collector
libraries: `(chezscheme)`

The total number of bytes allocated by the current process, whether still in use or not, can be obtained by summing `(bytes-deallocated)` and `(bytes-allocated)` and possibly subtracting `(initial-bytes-allocated)`.

`(current-memory-bytes)` **procedure**
returns: the total number of bytes currently allocated, including overhead
libraries: `(chezscheme)`

`current-memory-bytes` returns the total size of the heap in bytes, including not only the bytes occupied for Scheme objects but also various forms of overhead, including fragmentation and reserved but not currently occupied memory, and is thus an accurate measure of the amount of heap memory currently reserved from the operating system for the current process.

`(maximum-memory-bytes)` **procedure**
returns: the maximum number of bytes ever allocated, including overhead
libraries: `(chezscheme)`

`maximum-memory-bytes` returns the maximum size of the heap in bytes, i.e., the maximum value that `current-memory-bytes` returned or could have returned, since the last call to `reset-maximum-memory-bytes!` or, if there has been no such call, since the process started.

`(reset-maximum-memory-bytes!)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`reset-maximum-memory-bytes!` resets the maximum recorded size of the heap to the current size of the heap.

`(collections)` **procedure**
returns: the number garbage collections so far
libraries: `(chezscheme)`

`(statistics)` **procedure**
returns: a sstats record containing current statistics
libraries: `(chezscheme)`

`statistics` packages together various timing and allocation statistics into a single `sstats` record. A `sstats` record has the following fields:

`cpu`, the cpu time consumed,

`real`, the elapsed real time,

`bytes`, the number of bytes allocated,

`gc-count`, the number of collections,

`gc-cpu`, the cpu time consumed during collections,

`gc-real`, the elapsed real time during collections, and

`gc-bytes`, the number of bytes reclaimed by the collector.

All values are computed since system start-up. The time values are time objects (Section 12.10), and the bytes and count values are exact integers.

`statistics` might be defined as follows:

```
(define statistics
  (lambda ()
    (make-sstats
      (current-time 'time-thread)
      (current-time 'time-monotonic)
      (- (+ (bytes-allocated) (bytes-deallocated))
         (initial-bytes-allocated))
      (collections)
      (current-time 'time-collector-cpu)
      (current-time 'time-collector-real)
      (bytes-deallocated))))
```

`(make-sstats` *cpu real bytes gc-count gc-cpu gc-real gc-bytes*`)` **procedure**
returns: a sstats record
libraries: `(chezscheme)`

The time arguments (*cpu*, *real*, *gc-cpu*, and *gc-real*) must be time objects. The other arguments must be exact integers.

`(sstats?` *obj*`)` **procedure**
returns: `#t` if *obj* is a sstats record, otherwise `#f`
libraries: `(chezscheme)`

(sstats-cpu *s*) **procedure**
(sstats-real *s*) **procedure**
(sstats-bytes *s*) **procedure**
(sstats-gc-count *s*) **procedure**
(sstats-gc-cpu *s*) **procedure**
(sstats-gc-real *s*) **procedure**
(sstats-gc-bytes *s*) **procedure**
returns: the value of the corresponding field of *s*
libraries: (chezscheme)

s must be a sstats record.

(set-sstats-cpu! *s new-value*) **procedure**
(set-sstats-real! *s new-value*) **procedure**
(set-sstats-bytes! *s new-value*) **procedure**
(set-sstats-gc-count! *s new-value*) **procedure**
(set-sstats-gc-cpu! *s new-value*) **procedure**
(set-sstats-gc-real! *s new-value*) **procedure**
(set-sstats-gc-bytes! *s new-value*) **procedure**
returns: unspecified
libraries: (chezscheme)

s must be a sstats record, the *new-value* arguments for the time fields (*cpu*, *real*, *gc-cpu*, and *gc-real*) must be time objects, and the other *new-value* arguments must be exact integers. Each procedure sets the value of the corresponding field of *s* to *new-value*.

(sstats-difference s_1 s_2) **procedure**
returns: a sstats record representing the difference between s_1 and s_2
libraries: (chezscheme)

s_1 and s_2 must be sstats records. `sstats-difference` subtracts each field of s_2 from the corresponding field of s_1 to produce the resulting `sstats` record.

(sstats-print *s*) **procedure**
(sstats-print *s textual-output-port*) **procedure**
returns: unspecified
libraries: (chezscheme)

s must be a `sstats` record. If *textual-output-port* is not supplied, it defaults to the current output port. `sstats-print` displays the fields of `s` in a manner similar to `display-statistics` and `time`.

enable-object-counts **global parameter**
libraries: (chezscheme)

The value of `enable-object-counts` is a boolean value that determines whether the col-

lector records object counts as it runs and hence whether the object counts returned by the procedure `object-counts` are accurate. The parameter is set to `#f` by default, since enabling object counts adds overhead to collection.

Counts for the static generation are always correct. Counts for a nonstatic generation n are correct immediately after a collection of generation $m \geq n$ (regardless of whether the target generation is m or $m + 1$) if `enable-object-counts` was set to `#t` during the collection.

One strategy for collecting object counts with minimal overhead is to enable object counts only while collecting the maximum nonstatic generation and to obtain the object counts immediately after that collection.

`(object-counts)` **procedure**
returns: see below
libraries: `(chezscheme)`

The procedure `object-counts` returns a nested association list representing object counts and bytes allocated for each heap-allocated primitive type and record type with at least one live instance in one or more generations. (Heap-allocated primitive types include, e.g., pairs and vectors, but not, e.g., fixnums or characters.) Object counts are gathered by the collector only when `enable-object-counts` is `#t`. The description of `enable-object-counts` details the circumstances under which the counts are accurate.

The association list returned by `object-counts` has the following structure:

`((`*type* `(`*generation count* `.` *bytes*`) ...) ...)`

type is either the name of a primitive type, represented as a symbol, e.g., `pair`, or a record-type descriptor (rtd). *generation* is a nonnegative fixnum between 0 and the value of `(collect-maximum-generation)`, inclusive, or the symbol `static` representing the static generation. *count* and *bytes* are nonnegative fixnums.

```
(collect-request-handler void)
(enable-object-counts #t)
(define-record-type frob (fields x))
(define x (make-frob (make-frob #f)))
(collect 3 3)
(cdr (assoc 3
       (cdr (assoc (record-type-descriptor frob)
              (object-counts)))))                ⇒  (2 . 16)
```

12.12. Cost Centers

Cost centers are used to track the bytes allocated, instructions executed, and/or cpu time elapsed while evaluating selected sections of code. Cost centers are created via the procedure `make-cost-center`, and costs are tracked via the procedure `with-cost-center`.

Allocation and instruction counts are tracked only for code instrumented for that purpose. This instrumentation is controlled by two parameters: `generate-allocation-counts` and

`generate-instruction-counts`. Instrumentation is disabled by default. Built in procedures are not instrumented, nor is interpreted code or non-Scheme code. Elapsed time is tracked only when the optional `timed?` argument to `with-cost-center` is provided and is not false.

The `with-cost-center` procedure accurately tracks costs, subject to the caveats above, even when reentered with the same cost center, used simultaneously in multiple threads, and exited or reentered one or more times via continuation invocation.

`generate-allocation-counts` **thread parameter**
libraries: `(chezscheme)`

When this parameter has a true value, the compiler inserts a short sequence of instructions at each allocation point in generated code to track the amount of allocation that occurs. This parameter is initially false.

`generate-instruction-counts` **thread parameter**
libraries: `(chezscheme)`

When this parameter has a true value, the compiler inserts a short sequence of instructions in each block of generated code to track the number of instructions executed by that block. This parameter is initially false.

`(make-cost-center)` **procedure**
returns: a new cost center
libraries: `(chezscheme)`

The recorded costs of the new cost center are initialized to zero.

`(cost-center?` *obj*`)` **procedure**
returns: `#t` if *obj* is a cost center, otherwise `#f`
libraries: `(chezscheme)`

`(with-cost-center` *cost-center thunk*`)` **procedure**
`(with-cost-center` *timed? cost-center thunk*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

thunk must be a procedure that accepts zero arguments. `with-cost-center` invokes *thunk* without arguments and returns its values. It also tracks, dynamically, the bytes allocated, instructions executed, and cpu time elapsed while evaluating the invocation of *thunk* and adds the tracked costs to the cost center's running record of these costs.

As described above, allocation counts are tracked only for code compiled with the parameter `generate-allocation-counts` set to true, and instruction counts are tracked only for code compiled with `generate-instruction-counts` set to true. Cpu time is tracked only if *timed?* is provided and not false and includes cpu time spent in instrumented, uninstrumented, and non-Scheme code.

(cost-center-instruction-count *cost-center*) **procedure**
returns: the number of instructions tracked by *cost-center*
libraries: (chezscheme)

(cost-center-allocation-count *cost-center*) **procedure**
returns: the number of allocated bytes tracked by *cost-center*
libraries: (chezscheme)

(cost-center-time *cost-center*) **procedure**
returns: the cpu time tracked by *cost-center*
libraries: (chezscheme)

The cpu time is returned as a time object with time-type `time-duration`.

(reset-cost-center! *cost-center*) **procedure**
returns: unspecified
libraries: (chezscheme)

This procedure resets the costs recorded by *cost-center* to zero.

12.13. Parameters

This section describes mechanisms for creating and manipulating parameters. New parameters may be created conveniently with `make-parameter`. Nothing distinguishes parameters from other procedures, however, except for their behavior. If more complicated actions must be taken when a parameter is invoked than can be accommodated easily through the `make-parameter` mechanism, the parameter may be defined directly with `case-lambda`.

(make-parameter *object*) **procedure**
(make-parameter *object procedure*) **procedure**
returns: a parameter (procedure)
libraries: (chezscheme)

`make-parameter` accepts one or two arguments. The first argument is the initial value of the internal variable, and the second, if present, is a *filter* applied to the initial value and all subsequent values. The filter should accept one argument. If the value is not appropriate, the filter should raise an exception or convert the value into a more appropriate form.

For example, the default value of `print-length` is defined as follows:

```
(define print-length
  (make-parameter
    #f
    (lambda (x)
      (unless (or (not x) (and (fixnum? x) (fx>= x 0)))
        (assertion-violationf 'print-length
          "~s is not a positive fixnum or #f"
          x))
      x)))

(print-length)   ⇒  #f
(print-length 3)
(print-length)   ⇒  3
(format "~s" '(1 2 3 4 5 6))   ⇒  "(1 2 3 ...)"
(print-length #f)
(format "~s" '(1 2 3 4 5 6))   ⇒  "(1 2 3 4 5 6)"
```

The definition of `make-parameter` is straightforward using `case-lambda`:

```
(define make-parameter
  (case-lambda
    [(init guard)
     (let ([v (guard init)])
       (case-lambda
         [() v]
         [(u) (set! v (guard u))]))]
    [(init)
     (make-parameter init (lambda (x) x))]))
```

In threaded versions of *Chez Scheme*, `make-parameter` creates global parameters. The procedure `make-thread-parameter`, described in Section 15.6, may be used to make thread parameters.

`(parameterize ((`*param expr*`) ...)` *body*$_1$ *body*$_2$ `...)` **syntax**
returns: the values of the body *body*$_1$ *body*$_2$ `...`
libraries: `(chezscheme)`

Using the syntactic form `parameterize`, the values of parameters can be changed in a manner analogous to `fluid-let` for ordinary variables. Each *param* is set to the value of the corresponding *expr* while the body is evaluated. When control leaves the body by normal return or by the invocation of a continuation created outside of the body, the parameters are restored to their original values. If control returns to the body via a continuation created during the execution of the body, the parameters are again set to their temporary values.

```
(define test
  (make-parameter 0))
(test)   ⇒  0
(test 1)
```

```
(test)   ⇒  1
(parameterize ([test 2])
  (test))   ⇒  2
(test)   ⇒  1
(parameterize ([test 2])
  (test 3)
  (test))   ⇒  3
(test)   ⇒  1
(define k (lambda (x) x))
(begin (set! k (call/cc k))
       'k)   ⇒  k
(parameterize ([test 2])
  (test (call/cc k))
  (test))   ⇒  k
(test)   ⇒  1
(k 3)   ⇒  3
(test)   ⇒  1
```

The definition of `parameterize` is similar to the definition of `fluid-let` (page 117):

```
(define-syntax parameterize
  (lambda (x)
    (syntax-case x ()
      [(_ () b1 b2 ...) #'(begin b1 b2 ...)]
      [(_ ((x e) ...) b1 b2 ...)
       (with-syntax ([(p ...) (generate-temporaries #'(x ...))]
                     [(y ...) (generate-temporaries #'(x ...))])
         #'(let ([p x] ... [y e] ...)
             (let ([swap (lambda ()
                           (let ([t (p)]) (p y) (set! y t))
                           ...)])
               (dynamic-wind swap (lambda () b1 b2 ...) swap))))])))
```

12.14. Virtual registers

A limited set of *virtual registers* is supported by the compiler for use by programs that require high-speed, global, and mutable storage locations. Referencing or assigning a virtual register is potentially faster and never slower than accessing an assignable local or global variable, and the code sequences for doing so are generally smaller. Assignment is potentially significantly faster because there is no need to track pointers from the virtual registers to young objects, as there is for variable locations that might reside in older generations. On threaded versions of the system, virtual registers are "per thread" and thus serve as thread-local storage in a manner that is less expensive than thread parameters.

The interface consists of three procedures: `virtual-register-count`, which returns the number of virtual registers, `set-virtual-register!`, which sets the value of a specified virtual register, and `virtual-register`, which retrieves the value of a specified virtual register.

A virtual register is specified by a nonnegative fixnum index less than the number of virtual registers. To get optimal performance for `set-virtual-register!` and `virtual-register`, the index should be a constant embedded right in the call (or propagatable via optimization to the call). To avoid putting these constants in the source code, programmers should consider using identifier macros to give names to virtual registers, e.g.:

```
(define-syntax current-state
  (identifier-syntax
    [id (virtual-register 0)]
    [(set! id e) (set-virtual-register! 0 e)]))
(set! current-state 'start)
current-state  ⇒  start
```

A more elaborate macro could dole out indices at compile time and complain when no more indices are available.

Virtual-registers must be treated as an application-level resource, i.e., libraries intended to be used by multiple applications should generally not use virtual registers to avoid conflicts with an application's use of the registers.

`(virtual-register-count)` **procedure**
returns: the number of virtual registers
libraries: `(chezscheme)`

As of Version 9.0, the number of virtual registers is set at 16. It cannot be changed except by recompiling *Chez Scheme* from source.

`(set-virtual-register!` *k x*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`set-virtual-register!` stores *x* in virtual register *k*. *k* must be a nonnegative fixnum less than the value of `(virtual-register-count)`.

`(virtual-register` *k*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

`virtual-register` returns the value most recently stored in virtual register *k* (on the current thread, in threaded versions of the system).

12.15. Environmental Queries and Settings

`(scheme-version)` **procedure**
returns: a version string
libraries: `(chezscheme)`

The version string is in the form

`"Chez Scheme Version` *version*`"`

for *Chez Scheme*, and

`"Petite Chez Scheme Version` *version*`"`

for *Petite Chez Scheme*.

`(scheme-version-number)` **procedure**
returns: three values: the major, minor, and sub-minor version numbers
libraries: `(chezscheme)`

Each of the three return values is a nonnegative fixnum.

In *Chez Scheme* Version 7.9.4:

```
(scheme-version-number)  ⇒  7
                            9
                            4
```

`(petite?)` **procedure**
returns: `#t` if called in *Petite Chez Scheme*, `#f` otherwise
libraries: `(chezscheme)`

The only difference between *Petite Chez Scheme* and *Chez Scheme* is that the compiler is not available in the former, so this predicate can serve as a way to determine if the compiler is available.

`(threaded?)` **procedure**
returns: `#t` if called in a threaded version of the system, `#f` otherwise
libraries: `(chezscheme)`

`(interactive?)` **procedure**
returns: `#t` if system is run interactively, `#f` otherwise
libraries: `(chezscheme)`

This predicate returns `#t` if the Scheme process's stdin and stdout are connected to a tty (Unix-based systems) or console (Windows). Otherwise, it returns `#f`.

`(get-process-id)` **procedure**
returns: the operating system process id if the current process
libraries: `(chezscheme)`

`(getenv` *key*`)` **procedure**
returns: environment value of *key* or `#f`
libraries: `(chezscheme)`

key must be a string. `getenv` returns the operating system shell's environment value associated with *key*, or `#f` if no environment value is associated with *key*.

```
(getenv "HOME")  ⇒  "/u/freddy"
```

`(putenv` *key value*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

key and *value* must be strings.

`putenv` stores the *key*, *value* pair in the environment of the process, where it is available to the current process (e.g., via *getenv*) and any spawned processes. The key and value are copied into storage allocated outside of the Scheme heap; this space is never reclaimed on non-Windows systems.

```
(putenv "SCHEME" "rocks!")
(getenv "SCHEME")  ⇒  "rocks!"
```

`(get-registry` *key*`)` **procedure**
returns: registry value of *key* or `#f`
`(put-registry!` *key val*`)` **procedure**
`(remove-registry!` *key*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

key and *val* must be strings.

`get-registry` returns a string containing the registry value of *key* if the value exists. If no registry value for *key* exists, `get-registry` returns `#f`.

`put-registry!` sets the registry value of *key* to *val*. It raises an exception with condition type `&assertion` if the value cannot be set, which may happen if the user has insufficient access.

`remove-registry!` removes the registry key or value named by *key*. It raises an exception with condition type `&assertion` if the value cannot be removed. Reasons for failure include the key not being present, the user having insufficient access, or *key* being a key with subkeys.

These routines are defined for Windows only.

```
(get-registry "hkey_local_machine\\Software\\North\\South")  ⇒  #f
(put-registry! "hkey_local_machine\\Software\\North\\South" "east")
(get-registry "hkey_local_machine\\Software\\North\\South")  ⇒  "east"
(remove-registry! "hkey_local_machine\\Software\\North")
(get-registry "hkey_local_machine\\Software\\North\\South")  ⇒  #f
```

12.16. Subset Modes

subset-mode **thread parameter**
libraries: (chezscheme)

The value of this parameter must be `#f` (the default) or the symbol `system`. Setting `subset-mode` to `system` allows the manipulation of various undocumented system variables, data structures, and settings. It is typically used only for system debugging.

13. Storage Management

This chapter describes aspects of the storage management system and procedures that may be used to control its operation.

13.1. Garbage Collection

Scheme objects such as pairs, strings, and procedures are never explicitly deallocated by a Scheme program. Instead, the storage management system automatically reclaims the storage associated with an object once it proves the object is no longer accessible. In order to reclaim this storage, *Chez Scheme* employs a garbage collector which runs periodically as a program runs. Starting from a set of known *roots*, e.g., the machine registers, the garbage collector locates all accessible objects, copies them (in most cases) in order to eliminate fragmentation between accessible objects, and reclaims storage occupied by inaccessible objects.

Collections are triggered automatically by the default collect-request handler, which is invoked via a collect-request interrupt that occurs after approximately n bytes of storage have been allocated, where n is the value of the parameter `collect-trip-bytes`. The default collect-request handler causes a collection by calling the procedure `collect` without arguments. The collect-request handler can be redefined by changing the value of the parameter `collect-request-handler`. A program can also cause a collection to occur between collect-request interrupts by calling `collect` directly.

Chez Scheme's collector is a *generation-based* collector. It segregates objects based on their age (roughly speaking, the number of collections survived) and collects older objects less frequently than younger objects. Since younger objects tend to become inaccessible more quickly than older objects, the result is that most collections take little time. The system also maintains a *static* generation from which storage is never reclaimed. Objects are placed into the static generation only when a heap is compacted (see `Scompact_heap` in Section 4.8) or when the target-generation argument to `collect` is the symbol `static`.

Nonstatic generations are numbered starting at zero for the youngest generation up through the current value of `collect-maximum-generation`. The storage manager places newly allocated objects into generation 0. During a generation 0 collection, objects in generation 0 that survive the collection move, by default, to generation 1. Similarly, during a generation 1 collection, objects in generations 0 and 1 that survive move to generation 2, and so on. During the collection of the maximum nonstatic collection, all surviving nonstatic objects

move (possibly back) into the maximum nonstatic generation. With this mechanism, it is possible for an object to skip one or more generations, but this is not likely to happen to many objects, and if the objects become inaccessible, their storage is reclaimed eventually.

An internal counter, gc-trip, is maintained to control when each generation is collected. Each time `collect` is called without arguments (as from the default collect-request handler), gc-trip is incremented by one. With a collect-generation radix of r, the collected generation is the highest numbered generation g for which gc-trip is a multiple of r^g. If `collect-generation-radix` is set to 4, the system thus collects generation 0 every time, generation 1 every 4 times, generation 2 every 16 times, and so on.

Each time `collect` is called with a single generation argument g, generation g is collected and gc-trip is advanced to the next r^g boundary, but not past the next r^{g+1} boundary, where r is again the value of `collect-generation-radix`.

If `collect` is called with a second generation argument, tg, tg determines the target generation. When g is the maximum nonstatic generation, tg must be g or the symbol `static`. Otherwise, tg must be g or $g+1$. When the target generation is the symbol `static`, all data in the nonstatic generations are moved to the static generation. Objects in the static generation are never collected. This is primarily useful after an application's permanent code and data structures have been loaded and initialized, to reduce the overhead of subsequent collections.

It is possible to make substantial adjustments in the collector's behavior by setting the parameters described in this section. It is even possible to completely override the collector's default strategy for determining when each generation is collected by redefining the collect-request handler to call `collect` with explicit g and tg arguments. For example, the programmer can redefine the handler to treat the maximum nonstatic generation as a static generation over a long period of time by calling `collect` with explicit g and tg arguments that are never equal to the maximum nonstatic generation during that period of time.

Additional information on *Chez Scheme*'s collector can be found in the report "Don't stop the BiBOP: Flexible and efficient storage management for dynamically typed languages" [13].

`(collect)` **procedure**
`(collect` *g*`)` **procedure**
`(collect` *g tg*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

g must be a nonnegative fixnum no greater than the maximum nonstatic generation, i.e., the value returned by `collect-maximum-generation`. If g is the maximum nonstatic generation, tg must be a fixnum equal to g or the symbol `static`. Otherwise, tg must be a fixnum equal to or one greater than g.

This procedure causes the storage manager to perform a garbage collection. `collect` is invoked periodically via the collect-request handler, but it may also be called explicitly to force collection at a particular time, e.g., before timing a computation. In the threaded versions of *Chez Scheme*, the thread that invokes `collect` must be the only active thread.

The system determines which generations to collect, based on *g* and *tg* if provided, as described in the lead-in to this section.

`(collect-rendezvous)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

Requests a garbage collection in the same way as when the system determines that a collection should occur. All running threads are coordinated so that one of them calls the collect-request handler, while the other threads pause until the handler returns.

Note that if the collect-request handler (see `collect-request-handler`) does not call `collect`, then `collect-rendezvous` does not actualy perform a garbage collection.

`collect-notify` **global parameter**
libraries: `(chezscheme)`

If `collect-notify` is set to a true value, the collector prints a message whenever a collection is run. `collect-notify` is set to `#f` by default.

`collect-trip-bytes` **global parameter**
libraries: `(chezscheme)`

This parameter determines the approximate amount of storage that is allowed to be allocated between garbage collections. Its value must be a positive fixnum.

Chez Scheme allocates memory internally in large chunks and subdivides these chunks via inline operations for efficiency. The storage manager determines whether to request a collection only once per large chunk allocated. Furthermore, some time may elapse between when a collection is requested by the storage manager and when the collect request is honored, especially if interrupts are temporarily disabled via `with-interrupts-disabled` or `disable-interrupts`. Thus, `collect-trip-bytes` is an approximate measure only.

`collect-generation-radix` **global parameter**
libraries: `(chezscheme)`

This parameter determines how often each generation is collected when `collect` is invoked without arguments, as by the default collect-request handler. Its value must be a positive fixnum. Generations are collected once every r^g times a collection occurs, where r is the value of `collect-generation-radix` and g is the generation number.

Setting `collect-generation-radix` to one forces all generations to be collected each time a collection occurs. Setting `collect-generation-radix` to a very large number effectively delays collection of older generations indefinitely.

`collect-maximum-generation` **global parameter**
libraries: `(chezscheme)`

This parameter determines the maximum nonstatic generation, hence the total number of generations, currently in use. Its value is an exact integer in the range 1 through 254. When set to 1, only two nonstatic generations are used; when set to 2, three nonstatic generations are used, and so on. When set to 254, 255 nonstatic generations are used, plus the single static generation for a total of 256 generations. Increasing the number of generations effectively decreases how often old objects are collected, potentially decreasing collection overhead but potentially increasing the number of inaccessible objects retained in the system and thus the total amount of memory required.

`collect-request-handler` **global parameter**
libraries: `(chezscheme)`

The value of `collect-request-handler` must be a procedure. The procedure is invoked without arguments whenever the system determines that a collection should occur, i.e., some time after an amount of storage determined by the parameter `collect-trip-bytes` has been allocated since the last collection.

By default, `collect-request-handler` simply invokes `collect` without arguments.

Automatic collection may be disabled by setting `collect-request-handler` to a procedure that does nothing, e.g.:

```
(collect-request-handler void)
```

Collection can also be temporarily disabled using `critical-section`, which prevents any interrupts from occurring.

`release-minimum-generation` **global parameter**
libraries: `(chezscheme)`

This parameter's value must be between 0 and the value of `collect-maximum-generation`, inclusive, and defaults to the value of `collect-maximum-generation`.

As new data is allocated and collections occur, the storage-management system automatically requests additional virtual memory address space from the operating system. Correspondingly, in the event the heap shrinks significantly, the system attempts to return some of the virtual-memory previously obtained from the operating system back to the operating system. By default, the system attempts to do so only after a collection that targets the maximum nonstatic generation. The system can be asked to do so after collections targeting younger generations as well by altering the value `release-minimum-generation` to something less than the value of `collect-maximum-generation`. When the generation to which the parameter is set, or any older generation, is the target generation of a collection, the storage management system attempts to return unneeded virtual memory to the operating system following the collection.

When `collect-maximum-generation` is set to a new value g, `release-minimum-generation` is implicitly set to g as well if (a) the two parameters have the same value before the change, or (b) `release-minimum-generation` has a value greater than g.

`heap-reserve-ratio` **global parameter**
libraries: `(chezscheme)`

This parameter determines the approximate amount of memory reserved (not returned to the O/S as described in the entry for `release-minimum-generation`) in proportion to the amount currently occupied, excluding areas of memory that have been made static. Its value must be an inexact nonnegative flonum value; if set to an exact real value, the exact value is converted to an inexact value. The default value, 1.0, reserves one page of memory for each currently occupied nonstatic page. Setting it to a smaller value may result in a smaller average virtual memory footprint, while setting it to a larger value may result in fewer calls into the operating system to request and free memory space.

13.2. Weak Pairs, Ephemeron Pairs, and Guardians

Weak pairs allow programs to maintain *weak pointers* to objects. A weak pointer to an object does not prevent the object from being reclaimed by the storage management system, but it does remain valid as long as the object is otherwise accessible in the system.

Ephemeron pairs are like weak pairs, but ephemeron pairs combine two pointers where the second is retained only as long as the first is retained.

Guardians allow programs to protect objects from deallocation by the garbage collector and to determine when the objects would otherwise have been deallocated.

Weak pairs, ephemeron pairs, and guardians allow programs to retain information about objects in separate data structures (such as hash tables) without concern that maintaining this information will cause the objects to remain indefinitely in the system. Ephemeron pairs allow such data structures to retain key–value combinations where a value may refer to its key, but the combination can be reclaimed if neither must be saved otherwise. In addition, guardians allow objects to be saved from deallocation indefinitely so that they can be reused or so that clean-up or other actions can be performed using the data stored within the objects.

The implementation of guardians and weak pairs used by *Chez Scheme* is described in [12]. Ephemerons are described in [23], but the implementation in *Chez Scheme* avoids quadratic-time worst-case behavior.

`(weak-cons` obj_1 obj_2`)` **procedure**
returns: a new weak pair
libraries: `(chezscheme)`

obj_1 becomes the car and obj_2 becomes the cdr of the new pair. Weak pairs are indistinguishable from ordinary pairs in all but two ways:

- weak pairs can be distinguished from pairs using the `weak-pair?` predicate, and
- weak pairs maintain a weak pointer to the object in the car of the pair.

The weak pointer in the car of a weak pair is just like a normal pointer as long as the object to which it points is accessible through a normal (nonweak) pointer somewhere in the system. If at some point the garbage collector recognizes that there are no nonweak pointers to the object, however, it replaces each weak pointer to the object with the "broken weak-pointer" object, `#!bwp`, and discards the object.

The cdr field of a weak pair is *not* a weak pointer, so weak pairs may be used to form lists of weakly held objects. These lists may be manipulated using ordinary list-processing operations such as `length`, `map`, and `assv`. (Procedures like `map` that produce list structure always produce lists formed from nonweak pairs, however, even when their input lists are formed from weak pairs.) Weak pairs may be altered using `set-car!` and `set-cdr!`; after a `set-car!` the car field contains a weak pointer to the new object in place of the old object. Weak pairs are especially useful for building association pairs in association lists or hash tables.

Weak pairs are printed in the same manner as ordinary pairs; there is no reader syntax for weak pairs. As a result, weak pairs become normal pairs when they are written and then read.

```
(define x (cons 'a 'b))
(define p (weak-cons x '()))
(car p)  ⇒  (a . b)
```

```
(define x (cons 'a 'b))
(define p (weak-cons x '()))
(set! x '*)
(collect)
(car p)  ⇒  #!bwp
```

The latter example above may in fact return `(a . b)` if a garbage collection promoting the pair into an older generation occurs prior to the assignment of `x` to `*`. It may be necessary to force an older generation collection to allow the object to be reclaimed. The storage management system guarantees only that the object will be reclaimed eventually once all nonweak pointers to it are dropped, but makes no guarantees about when this will occur.

`(weak-pair?` *obj*`)` **procedure**
returns: `#t` if obj is a weak pair, `#f` otherwise
libraries: `(chezscheme)`

```
(weak-pair? (weak-cons 'a 'b))  ⇒  #t
(weak-pair? (cons 'a 'b))  ⇒  #f
(weak-pair? "oops")  ⇒  #f
```

`(ephemeron-cons` obj_1 obj_2`)` **procedure**
returns: a new ephemeron pair
libraries: `(chezscheme)`

obj_1 becomes the car and obj_2 becomes the cdr of the new pair. Ephemeron pairs are indistinguishable from ordinary pairs in all but two ways:

- ephemeron pairs can be distinguished from pairs using the `ephemeron-pair?` predicate, and
- ephemeron pairs maintain a weak pointer to the object in the car of the pair, and the cdr of the pair is preserved only as long as the car of the pair is preserved.

An ephemeron pair behaves like a weak pair, but the cdr is treated specially in addition to the car: the cdr of an ephemeron is set to `#!bwp` at the same time that the car is set to `#!bwp`. Since the car and cdr fields are set to `#!bwp` at the same time, then the fact that the car object may be referenced through the cdr object does not by itself imply that car must be preserved (unlike a weak pair); instead, the car must be saved for some reason independent of the cdr object.

Like weak pairs and other pairs, ephemeron pairs may be altered using `set-car!` and `set-cdr!`, and ephemeron pairs are printed in the same manner as ordinary pairs; there is no reader syntax for ephemeron pairs.

```
(define x (cons 'a 'b))
(define p (ephemeron-cons x x))
(car p)  ⇒  (a . b)
(cdr p)  ⇒  (a . b)
```

```
(define x (cons 'a 'b))
(define p (ephemeron-cons x x))
(set! x '*)
(collect)
(car p)  ⇒  #!bwp
(cdr p)  ⇒  #!bwp
```

```
(define x (cons 'a 'b))
(define p (weak-cons x x)) ; not an ephemeron pair
(set! x '*)
(collect)
(car p)  ⇒  (a . b)
(cdr p)  ⇒  (a . b)
```

As with weak pairs, the last two expressions of the middle example above may in fact return `(a . b)` if a garbage collection promoting the pair into an older generation occurs prior to the assignment of `x` to `*`. In the last example above, however, the results of the last two expressions will always be `(a . b)`, because the cdr of a weak pair holds a non-weak reference, and that non-weak reference prevents the car field from becoming `#!bwp`.

`(ephemeron-pair?` *obj*`)` **procedure**
returns: `#t` if obj is a ephemeron pair, `#f` otherwise
libraries: `(chezscheme)`

```
(ephemeron-pair? (ephemeron-cons 'a 'b))  ⇒  #t
(ephemeron-pair? (cons 'a 'b))  ⇒  #f
(ephemeron-pair? (weak-cons 'a 'b))  ⇒  #f
(ephemeron-pair? "oops")  ⇒  #f
```

`(bwp-object?` *obj*`)` **procedure**
returns: `#t` if obj is the broken weak-pair object, `#f` otherwise
libraries: `(chezscheme)`

```
(bwp-object? #!bwp)  ⇒  #t
(bwp-object? 'bwp)  ⇒  #f
```

```
(define x (cons 'a 'b))
(define p (weak-cons x '()))
(set! x '*)
(collect (collect-maximum-generation))
(car p)  ⇒  #!bwp
(bwp-object? (car p))  ⇒  #t
```

`(make-guardian)` **procedure**
returns: a new guardian
libraries: `(chezscheme)`

Guardians are represented by procedures that encapsulate groups of objects registered for preservation. When a guardian is created, the group of registered objects is empty. An object is registered with a guardian by passing the object as an argument to the guardian:

```
(define G (make-guardian))
(define x (cons 'aaa 'bbb))
x  ⇒  (aaa . bbb)
(G x)
```

It is also possible to specify a "representative" object when registering an object. Continuing the above example:

```
(define y (cons 'ccc 'ddd))
y  ⇒  (ccc . ddd)
(G y 'rep)
```

The group of registered objects associated with a guardian is logically subdivided into two disjoint subgroups: a subgroup referred to as "accessible" objects, and one referred to "inaccessible" objects. Inaccessible objects are objects that have been proven to be inaccessible (except through the guardian mechanism itself or through the car field of a

weak or ephemeron pair), and accessible objects are objects that have not been proven so. The word "proven" is important here: it may be that some objects in the accessible group are indeed inaccessible but that this has not yet been proven. This proof may not be made in some cases until long after the object actually becomes inaccessible (in the current implementation, until a garbage collection of the generation containing the object occurs).

Objects registered with a guardian are initially placed in the accessible group and are moved into the inaccessible group at some point after they become inaccessible. Objects in the inaccessible group are retrieved by invoking the guardian without arguments. If there are no objects in the inaccessible group, the guardian returns `#f`. Continuing the above example:

```
(G)  ⇒  #f
(set! x #f)
(set! y #f)
(collect)
(G)  ⇒  (aaa . bbb)    ; this might come out second
(G)  ⇒  rep            ; and this first
(G)  ⇒  #f
```

The initial call to `G` returns `#f`, since the pairs bound to `x` and `y` are the only object registered with `G`, and the pairs are still accessible through those binding. When `collect` is called, the objects shift into the inaccessible group. The two calls to `G` therefore return the pair previously bound to `x` and the representative of the pair previously bound to `y`, though perhaps in the other order from the one shown. (As noted above for weak pairs, the call to collect may not actually be sufficient to prove the object inaccessible, if the object has migrated into an older generation.)

Although an object registered without a representative and returned from a guardian has been proven otherwise inaccessible (except possibly via the car field of a weak or ephemeron pair), it has not yet been reclaimed by the storage management system and will not be reclaimed until after the last nonweak pointer to it within or outside of the guardian system has been dropped. In fact, objects that have been retrieved from a guardian have no special status in this or in any other regard. This feature circumvents the problems that might otherwise arise with shared or cyclic structure. A shared or cyclic structure consisting of inaccessible objects is preserved in its entirety, and each piece registered for preservation with any guardian is placed in the inaccessible set for that guardian. The programmer then has complete control over the order in which pieces of the structure are processed.

An object may be registered with a guardian more than once, in which case it will be retrievable more than once:

```
(define G (make-guardian))
(define x (cons 'aaa 'bbb))
(G x)
(G x)
(set! x #f)
(collect)
(G)  ⇒  (aaa . bbb)
(G)  ⇒  (aaa . bbb)
```

It may also be registered with more than one guardian, and guardians themselves can be registered with other guardians.

An object that has been registered with a guardian without a representative and placed in the car field of a weak or ephemeron pair remains in the car field of the weak or ephemeron pair until after it has been returned from the guardian and dropped by the program or until the guardian itself is dropped.

```
(define G (make-guardian))
(define x (cons 'aaa 'bbb))
(define p (weak-cons x '()))
(G x)
(set! x #f)
(collect)
(set! y (G))
y  ⇒  (aaa . bbb)
(car p)  ⇒  (aaa . bbb)
(set! y #f)
(collect 1)
(car p)  ⇒  #!bwp
```

(The first collector call above would promote the object at least into generation 1, requiring the second collector call to be a generation 1 collection. This can also be forced by invoking `collect` several times.)

On the other hand, if a representative (other than the object itself) is specified, the guarded object is dropped from the car field of the weak or ephemeron pair at the same time as the representative becomes available from the guardian.

```
(define G (make-guardian))
(define x (cons 'aaa 'bbb))
(define p (weak-cons x '()))
(G x 'rep)
(set! x #f)
(collect)
(G)  ⇒  rep
(car p)  ⇒  #!bwp
```

The following example illustrates that the object is deallocated and the car field of the weak pair set to `#!bwp` when the guardian itself is dropped:

```
(define G (make-guardian))
(define x (cons 'aaa 'bbb))
(define p (weak-cons x '()))
(G x)
(set! x #f)
(set! G #f)
(collect)
(car p)  ⇒  #!bwp
```

The example below demonstrates how guardians might be used to deallocate external

storage, such as storage managed by the C library "malloc" and "free" operations.

```
(define malloc
  (let ([malloc-guardian (make-guardian)])
    (lambda (size)
      ; first free any storage that has been dropped.  to avoid long
      ; delays, it might be better to deallocate no more than, say,
      ; ten objects for each one allocated
      (let f ()
        (let ([x (malloc-guardian)])
          (when x
            (do-free x)
            (f))))
      ; then allocate and register the new storage
      (let ([x (do-malloc size)])
        (malloc-guardian x)
        x))))
```

`do-malloc` must return a Scheme object "header" encapsulating a pointer to the external storage (perhaps as an unsigned integer), and all access to the external storage must be made through this header. In particular, care must be taken that no pointers to the external storage exist outside of Scheme after the corresponding header has been dropped. `do-free` must deallocate the external storage using the encapsulated pointer. Both primitives can be defined in terms of `foreign-alloc` and `foreign-free` or the C-library "malloc" and "free" operators, imported as foreign procedures. (See Chapter 4.)

If it is undesirable to wait until `malloc` is called to free dropped storage previously allocated by `malloc`, a collect-request handler can be used instead to check for and free dropped storage, as shown below.

```
(define malloc)
(let ([malloc-guardian (make-guardian)])
  (set! malloc
    (lambda (size)
      ; allocate and register the new storage
      (let ([x (do-malloc size)])
        (malloc-guardian x)
        x)))
  (collect-request-handler
    (lambda ()
      ; first, invoke the collector
      (collect)
      ; then free any storage that has been dropped
      (let f ()
        (let ([x (malloc-guardian)])
          (when x
            (do-free x)
            (f)))))))
```

With a bit of refactoring, it would be possible to register the encapsulated foreign address

as a representative with each header, in which `do-free` would take just the foreign address as an argument. This would allow the header to be dropped from the Scheme heap as soon as it becomes inaccessible.

13.3. Locking Objects

All pointers from C variables or data structures to Scheme objects should generally be discarded before entry (or reentry) into Scheme. When this guideline cannot be followed, the object may be *locked* via `lock-object` or via the equivalent C library procedure `Slock_object` (Section 4.8).

`(lock-object` *obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

Locking an object prevents the storage manager from reclaiming or relocating the object. Locking should be used sparingly, as it introduces memory fragmentation and increases storage management overhead.

Locking can also lead to accidental retention of storage if objects are not unlocked. Objects may be unlocked via `unlock-object` or the equivalent C library procedure `Sunlock_object`.

Locking immediate values, such as fixnums, booleans, and characters, or objects that have been made static is unnecessary but harmless.

`(unlock-object` *obj*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

An object may be locked more than once by successive calls to `lock-object`, `Slock_object`, or both, in which case it must be unlocked by an equal number of calls to `unlock-object` or `Sunlock_object` before it is truly unlocked.

An object contained within a locked object, such as an object in the car of a locked pair, need not also be locked unless a separate C pointer to the object exists. That is, if the inner object is accessed only via an indirection of the outer object, it should be left unlocked so that the collector is free to relocate it during collection.

Unlocking immediate values, such as fixnums, booleans, and characters, or objects that have been made static is unnecessary and ineffective but harmless.

`(locked-object?` *obj*`)` **procedure**
returns: `#t` if *obj* is locked, immediate, or static
libraries: `(chezscheme)`

This predicate returns true if *obj* cannot be relocated or reclaimed by the collector, including immediate values, such as fixnums, booleans, and characters, and objects that have

been made static.

14. Expression Editor

When the expression editor (expeditor) is enabled as described in Section 2.2, it allows the user to edit expressions entered into the system and move backwards and forwards through a history of entered expressions. This chapter describes a set of parameters that may be used to control various aspects of the expression editor's behavior (Section 14.1), a procedure for binding key sequences to editing commands (Section 14.2), the built-in editing commands (Section 14.3), and mechanisms for creating new editing commands (Section 14.4).

These mechanisms are available through the `expression-editor` module.

`expression-editor` **module**

libraries: `(chezscheme)`

The `expression-editor` module exports a set of bindings for parameters and other procedures that can be used to modify how the expression editor interacts with the user, including the particular keys used to invoke the various editing commands.

Basic use of the expression editor is described in Section 2.2.

14.1. Expression Editor Parameters

`ee-auto-indent` **global parameter**

The value of `ee-auto-indent` is a boolean value that determines whether the expression editor indents expressions as they are entered. Its default value is `#t`.

`ee-standard-indent` **global parameter**

The value of `ee-standard-indent` is a nonnegative fixnum value that determines the amount (in single spaces) by which each expression is indented relative to the enclosing expression, if not aligned otherwise by one of the indenter's other heuristics, when `ee-auto-indent` is true or when one of the indentation commands is invoked explicitly. It's default value is `2`.

ee-auto-paren-balance **global parameter**

The value of `ee-auto-paren-balance` is a boolean value that determines whether the expression editor automatically corrects a close parenthesis or bracket, when typed, to match the corresponding open parenthesis or bracket, if any. Its default value is `#t`.

ee-flash-parens **global parameter**

The value of `ee-flash-parens` is a boolean value that determines whether the expression editor briefly moves the cursor when an open or close parenthesis or bracket is typed to the matching close or open parenthesis or bracket (if any). Its default value is `#t`.

ee-paren-flash-delay **global parameter**

The value of `ee-paren-flash-delay` is a nonnegative fixnum value that determines the amount of time (in milliseconds) that the expression editor pauses when the cursor is moved to the matching parenthesis or bracket, if any, when a parenthesis or bracket is entered. The value is ignored if the `ee-flash-parens` is false. Its default value is `100`.

ee-default-repeat **global parameter**

The value of `ee-default-repeat` is a nonnegative fixnum value that determines the number of times the next command is repeated after the `ee-command-repeat` editing command (bound to `Esc-^U` by default) is used and *not* followed by a sequence of digits. It's default value is `4`.

ee-noisy **global parameter**

The value of `ee-noisy` is a boolean value that determines whether the expression editor emits a beep (bell) when an error occurs, such as an attempt to find the matching delimiter for a non-delimiter character. Its default value is `#f`.

ee-history-limit **global parameter**

The value of `ee-history-limit` is a nonnegative fixnum value that determines the number of history entries retained by the expression editor during and across sessions. Only the last `(ee-history-limit)` entries are retained.

ee-common-identifiers **global parameter**

The value of `ee-common-identifiers` is list of symbols that are considered common enough that they should appear early when one of the incremental identifier-completion editing commands is invoked. Its default value contains a few dozen entries. They are all more than a few characters long (under the theory that users will most likely type short ones out fully) and all would appear later than they likely should when incremental identifier-completion is used.

14.2. Key Binding

Key bindings are established via `ee-bind-key`. The default key bindings are described in Section 14.3.

`(ee-bind-key` *key procedure*`)` **procedure**
returns: unspecified

The `ee-bind-key` procedure is used to add to or change the set of key bindings recognized by the expression editor.

The *key* must be a character or string; if it is a string, it must have the following form.

⟨key-string⟩ ⟶ `"`⟨key-char⟩`+"`

where

⟨key-char⟩ ⟶ `\\e` (specifying an escape character)
| `^`*x* (specifying control-*x*)
| `\\^` (specifying caret)
| `\\\\` (specifying back slash)
| `plain char` (any character other than `\` or `^`)

Note that each double-backslash in the syntax actually denotes just one backslash in the string.

For example, the *key* `"\\eX"` represents the two-character sequence Escape-x, i.e., the "escape" key followed by the (capital) "X" key. Similarly, they *key* `"\\e^X"` represents the two-character sequence Escape-Control-x, i.e., the "escape" key followed by Control-X.

Character keys and string keys consisting of a single plain character always represent a single keystroke.

The *procedure* argument should normally be one of the built-in editing commands described below. It is also possible to define new editing commands with `ee-string-macro` and `ee-compose`.

14.3. Editing Commands

The editing commands are grouped into sections according to usage. Each is listed along with the default character sequence or sequences by which it may be invoked.

Insertion commands

command: `ee-insert-self`
key(s): most printing characters

Inserts the entered character into the entry.

command: `ee-insert-paren`
key(s): `(`, `)`, `[`, `]`

Inserts the entered parenthesis or bracket into the entry.

If the parameter `ee-auto-paren-balance` is true, the editor corrects close delimiters if necessary to balance existing open delimiters, when a matching open delimiter can be found.

If the parameter `ee-flash-parens` is true, the editor briefly moves the cursor to the matching delimiter, if one can be found, pausing for an amount of time controlled by the parameter `ee-paren-flash-delay`. If the matching delimiter is not presently displayed, the cursor is flashed to the upper-left or lower-left corner of the displayed portion of the entry, as appropriate.

The behavior of this command is undefined if used for something other than a parenthesis or bracket. parentheses and brackets.

command: `ee-newline`
key(s): none

Inserts a newline at the cursor position, moves to the next line, and indents that line if the parameter `ee-auto-indent` is true. Does nothing if the entry is empty. See also `ee-newline/accept`.

command: `ee-open-line`
key(s): `^O`

Inserts a newline at the cursor position and indents the next line, but does not move to the next line.

command: `ee-yank-kill-buffer`
key(s): `^Y`

Inserts the contents of the kill buffer, which is set by the deletion commands described below.

command: `ee-yank-selection`
key(s): `^V`

Inserts the contents of the window system's current selection or paste buffer. When running in a shell window under X Windows, this command requires that the DISPLAY environment variable be set to the appropriate display.

Cursor movement commands

command: `ee-backward-char`
key(s): leftarrow, `^B`

Moves the cursor left one character.

command: `ee-forward-char`
key(s): rightarrow, `^F`

Moves the cursor right one character.

command: `ee-next-line`
key(s): downarrow, `^N`

Moves the cursor down one line (and to the left if necessary so that the cursor does not sit beyond the last possible position). If the cursor is at the end of the current entry, and the current entry has not been modified, this command behaves like `ee-history-fwd`.

command: `ee-previous-line`
key(s): uparrow, `^P`

Moves the cursor up one line (and to the left if necessary so that the cursor does not sit beyond the last possible position). If the cursor is at the top of the current entry, and the current entry has not been modified, this command behaves like `ee-history-bwd`.

command: `ee-beginning-of-line`
key(s): home, `^A`

Moves the cursor to the first character of the current line.

command: `ee-end-of-line`
key(s): end, `^E`

Moves the cursor to the right of the last character of the current line.

command: `ee-beginning-of-entry`
key(s): escape-`<`

Moves the cursor to the first character of the entry.

command: `ee-end-of-entry`
key(s): escape-`>`

Moves the cursor to the right of the last character of the entry.

command: `ee-goto-matching-delimiter`
key(s): escape-`]`

Moves the cursor to the matching delimiter. Has no effect if the character under the cursor is not a parenthesis or bracket or if no matching delimiter can be found.

command: `ee-flash-matching-delimiter`
key(s): `^]`

Moves the cursor briefly to the matching delimiter, if one can be found, pausing for an amount of time controlled by the parameter `ee-paren-flash-delay`. If the matching delimiter is not presently displayed, the cursor is flashed to the upper-left or lower-left corner of the displayed portion of the entry, as appropriate.

command: `ee-exchange-point-and-mark`
key(s): `^X-^X`

Moves the cursor to the mark and leaves the mark at the old cursor position. (The mark can be set with `ee-set-mark`.)

command: `ee-forward-sexp`
key(s): escape-`^F`

Moves the cursor to the start of the next expression.

command: `ee-backward-sexp`
key(s): escape-`^B`

Moves the cursor to the start of the preceding expression.

command: `ee-forward-word`
key(s): escape-f, escape-`F`

Moves the cursor to the end of the next word.

command: `ee-backward-word`
key(s): escape-b, escape-`B`

Moves the cursor to the start of the preceding word.

command: `ee-forward-page`
key(s): pagedown, `^X-]`

Moves the cursor down one screen page.

command: `ee-backward-page`
key(s): pageup, `^X-[`

Moves the cursor up one screen page.

Deletion commands

command: `ee-delete-char`
key(s): delete

Deletes the character under the cursor.

See also `ee-eof/delete-char`.

command: `ee-backward-delete-char`
key(s): backspace (rubout), `^H`

Deletes the character to the left of the cursor.

command: `ee-delete-line`
key(s): `^U`

Deletes the contents of the current line, leaving behind an empty line. When used on the first line of a multiline entry of which only the first line is displayed, i.e., immediately after history movement, `ee-delete-line` deletes the contents of the entire entry, like `ee-delete-entry` (described below).

command: `ee-delete-to-eol`
key(s): `^K`, escape-`K`

If the cursor is at the end of a line, joins the line with the next line, otherwise deletes from the cursor position to the end of the line.

command: `ee-delete-between-point-and-mark`
key(s): `^W`

Deletes text between the current cursor position and the mark. (The mark can be set with `ee-set-mark`.)

command: `ee-delete-entry`
key(s): `^G`

Deletes the contents of the current entry.

command: `ee-reset-entry`
key(s): `^C`

Deletes the contents of the current entry and moves to the end of the history.

command: `ee-delete-sexp`
key(s): escape-`^K`, escape-delete

Deletes the expression that starts under the cursor, or if no expression starts under the cursor, deletes up to the next expression.

command: `ee-backward-delete-sexp`
key(s): escape-backspace (escape-rubout), escape-`^H`

Deletes the expression to the left of the cursor.

Identifier/filename completion commands

These commands perform either identifier or filename completion. Identifier completion is performed outside of a string constant, and filename completion is performed within a string constant. (In determining whether the cursor is within a string constant, the expression editor looks only at the current line and so can be fooled by string constants that span multiple lines.)

command: `ee-id-completion`
key(s): none

Inserts the common prefix of possible completions of the identifier or filename immediately to the left of the cursor. Identifier completion is based on the identifiers defined in the interaction environment. When there is exactly one possible completion, the common prefix is the completion. This command has no effect if no filename or identifier prefix is immediately the left of the cursor or if the possible completions have no common prefix. If run twice in succession, a list of possible completions is displayed.

See also `ee-id-completion/indent`.

command: `ee-next-id-completion`
key(s): `^R`

Inserts one of the possible completions of the identifier or filename immediately to the left of the cursor. Identifier completion is based on the identifiers defined in the interaction environment. If run twice or more in succession, this command cycles through all of the possible completions. The order is determined by the following heuristics: appearing first are identifiers whose names appear in the list value of the parameter `ee-common-identifiers`; appearing second are identifiers bound in the interaction environment but not bound in the scheme-environment (i.e., identifiers defined by the user), and appearing last are those in the scheme environment. Within the set of matches appearing in the `ee-common-identifiers` list, those listed earliest are shown first; the order is alphabetical within the other two sets.

See also `ee-next-id-completion/indent`.

History movement commands

The expression editor maintains a history of entries during each session. It also saves the history across sessions unless this behavior is disabled via the command-line argument "`--eehistory off`."

When moving from one history entry to another, only the first line of each multi-line entry is displayed. The redisplay command (which `^L` is bound to by default) can be used to display the entire entry. It is also possible to move down one line at a time to expose just part of the rest of the entry.

command: `ee-history-bwd`
key(s): escape-uparrow, escape-`^P`

Moves to the preceding history entry if the current entry is empty or has not been modified; otherwise, has no effect.

See also `ee-previous-line`.

command: `ee-history-fwd`
key(s): escape-downarrow, escape-`^N`

Moves to the next history entry if the current entry is empty or has not been modified; otherwise, has no effect.

See also `ee-next-line`.

command: `ee-history-bwd-prefix`
key(s): escape-`p`

Moves to the closest previous history entry, if any, that starts with the sequence of characters that makes up the current entry. May be used multiple times to search for same prefix.

command: `ee-history-fwd-prefix`
key(s): escape-`n`

Moves to the closest following history entry, if any, that starts with the sequence of characters that makes up the current entry. May be used multiple times to search for same prefix.

command: `ee-history-bwd-contains`
key(s): escape-`P`

Moves to the closest previous history entry, if any, that contains within it the sequence of characters that makes up the current entry. May be used multiple times to search for same content.

command: `ee-history-fwd-contains`
key(s): escape-`N`

Moves to the closest following history entry, if any, that contains within it the sequence of characters that makes up the current entry. May be used multiple times to search for same content.

Indentation commands

command: `ee-indent`
key(s): escape-tab

Re-indents the current line.

See also `ee-next-id-completion/indent`.

command: `ee-indent-all`
key(s): escape-`q`, escape-`Q`, escape-`^Q`

Re-indents each line of the entire entry.

Miscellaneous commands

command: `ee-accept`
key(s): `^J`

Causes the expression editor to invoke the Scheme reader on the contents of the entry. If the read is successful, the expression is returned to the waiter; otherwise, an error message is printed, the entry redisplayed, and the cursor left (if possible) at the start of the invalid subform.

See also `ee-newline/accept`.

command: `ee-eof`
key(s): none

Causes end-of-file to be returned from the expression editor, which in turn causes the waiter to exit. Ignored unless entry is empty.

See also `ee-eof/delete-char`.

command: `ee-redisplay`
key(s): `^L`

Redisplays the current expression. If run twice in succession, clears the screen and redisplays the expression at the top of the screen.

command: `ee-suspend-process`
key(s): `^Z`

Suspends the current process in shells that support job control.

command: `ee-set-mark`
key(s): `^@`, ^space

Sets the mark to the current cursor position.

command: `ee-command-repeat`
key(s): escape-`^U`

Repeats the next command n times. If the next character typed is a digit, n is determined by reading up the sequence of the digits typed and treating it as a decimal number. Otherwise, n is the value of the parameter `ee-default-repeat`.

Combination commands

command: `ee-newline/accept`
key(s): enter, `^M`

Behaves like `ee-accept` if run at the end (not including whitespace) of an entry that starts with a balanced expression; otherwise, behaves like `ee-newline`.

command: `ee-id-completion/indent`
key(s): tab

Behaves like `ee-id-completion` if an identifier (outside a string constant) or filename (within a string constant) appears just to the left of the cursor and the last character of that identifier or filename was just entered; otherwise, behaves like `ee-indent`.

If an existing identifier or filename, i.e., not one just typed, appears to the left of the cursor, the first use of this command behaves like `ee-newline`, the second consecutive use behaves like `ee-id-completion`, and the third behaves like a second consecutive use of `ee-id-completion`.

command: `ee-next-id-completion/indent`
key(s): none

Behaves like `ee-next-id-completion` if an identifier (outside a string constant) or filename (within a string constant) appears just to the left of the cursor and the last character of that identifier or identifier was just entered; otherwise, behaves like `ee-indent`.

command: `ee-eof/delete-char`
key(s): `^D`

Behaves like `ee-delete-char` if the entry is nonempty; otherwise, behaves like `ee-eof`. If the entry is nonempty and this command is run twice or more in succession, it does nothing once the entry becomes empty. This is to prevent accidental exit from the waiter in cases

where the command is run repeatedly (perhaps with the help of a keyboard's auto-repeat feature) to delete all of the characters in an entry.

14.4. Creating New Editing Commands

`(ee-string-macro` *string*`)` **procedure**
returns: a new editing command

The new editing command produced inserts *string* before the current cursor position.

Two string macros are predefined:

```
(ee-string-macro "(define ")   escape-d
(ee-string-macro "(lambda ")   escape-l
```

`(ee-compose` *ecmd* `...)` **procedure**
returns: a new editing command

Each *ecmd* must be an editing command.

The new editing command runs each of the editing commands *ecmd* `...` in sequence.

For example, the following expression binds `^X`-p to an editing command that behaves like `ee-history-bwd-prefix` but leaves the cursor at the end of the expression rather than at the end of the first line, causing the entire entry to be displayed.

```
(let ()
  (import expression-editor)
  (ee-bind-key "^Xp"
    (ee-compose ee-history-bwd ee-end-of-entry)))
```

A command such as `ee-id-completion` that performs a different action when run twice in succession will not recognize that it has been run twice in succession if run as part of a composite command.

15. Thread System

This chapter describes the *Chez Scheme* thread-system procedures and syntactic forms. With the exception of locks, locked increment, and locked decrement, the features of the thread system are implemented on top of the Posix thread system (pthreads) on non-Windows-based system and directly using the Windows API on Windows-based systems. Consult the appropriate documentation on your system for basic details of thread creation and interaction.

Most primitive Scheme procedures are *thread-safe*, meaning that they can be called concurrently from multiple threads. This includes allocation operations like *cons* and `make-string`, accessors like `car` and `vector-ref`, numeric operators like `+` and `sqrt`, and nondestructive higher-level primitive operators like `append` and `map`.

Simple mutation operators, like `set-car!`, `vector-set!`, and record field mutators are thread-safe. Likewise, assignments to local variables, including assignments to (unexported) library and top-level program variables are thread-safe.

Other destructive operators are thread safe only if they are used to operate on different objects from those being read or modified by other threads. For example, assignments to global variables are thread-safe only as long as one thread does not assign the same variable another thread references or assigns. Similarly, `putprop` can be called in one thread while another concurrently calls `putprop` or `getprop` if the symbols whose property lists are being modified or accessed differ.

In this context, most I/O operations should be considered destructive, since they might modify a port's internal structure; see also Section 15.7 for information on buffered ports.

Use of operators that are not thread-safe without proper synchronization can corrupt the objects upon which they operate. This corruption can lead to incorrect behavior, memory faults, and even unrecoverable errors that cause the system to abort.

The compiler and interpreter are thread-safe to the extent that user code evaluated during the compilation and evaluation process is thread-safe or properly synchronized. Thus, two or more threads can call any of the compiler or interpreter entry points, i.e., `compile`, `compile-file`, `compile-program`, `compile-script`, `compile-port`, or `interpret` at the same time. Naturally, the object-file targets of two file compilation operations that run at the same time should be different. The same is true for `eval` and `load` as long as the default evaluator is used or is set explicitly to `compile`, `interpret`, or some other thread-safe evaluator.

One restriction should be observed when one of multiple threads creates or loads compiled

code, however, which is that only that thread or subsequently created children, or children of subsequently created children, etc., should run the code. This is because multiple-processor systems upon which threaded code may run might not guarantee that the data and instruction caches are synchronized across processors.

15.1. Thread Creation

`(fork-thread` *thunk*`)` **procedure**
returns: a thread object
libraries: `(chezscheme)`

thunk must be a procedure that accepts zero arguments.

`fork-thread` invokes *thunk* in a new thread and returns a thread object.

Nothing can be done with the thread object returned by `fork-thread`, other than to print it.

Threads created by foreign code using some means other than `fork-thread` must call `Sactivate_thread` (Section 4.8) before touching any Scheme data or calling any Scheme procedures.

`(thread?` *obj*`)` **procedure**
returns: `#t` if *obj* is a thread object, `#f` otherwise
libraries: `(chezscheme)`

`(get-thread-id)` **procedure**
returns: the thread id of the current thread
libraries: `(chezscheme)`

The thread id is a thread number assigned by thread id, and has no relationship to the process id returned by `get-process-id`, which is the same in all threads.

15.2. Mutexes

`(make-mutex)` **procedure**
returns: a new mutex object
libraries: `(chezscheme)`

`(mutex?` *obj*`)` **procedure**
returns: `#t` if *obj* is a mutex, `#f` otherwise
libraries: `(chezscheme)`

`(mutex-acquire` *mutex*`)` **procedure**
`(mutex-acquire` *mutex block?*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

mutex must be a mutex.

mutex-acquire acquires the mutex identified by *mutex*. The optional boolean argument *block?* defaults to `#t` and specifies whether the thread should block waiting for the mutex. If *block?* is omitted or is true, the thread blocks until the mutex has been acquired, and an unspecified value is returned.

If `block?` is false and the mutex currently belongs to a different thread, the current thread does not block. Instead, `mutex-acquire` returns immediately with the value `#f` to indicate that the mutex is not available. If *block?* is false and the mutex is successfully acquired, `mutex-acquire` returns `#t`.

Mutexes are *recursive* in Posix threads terminology, which means that the calling thread can use `mutex-acquire` to (re)acquire a mutex it already has. In this case, an equal number of `mutex-release` calls is necessary to release the mutex.

`(mutex-release` *mutex*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

mutex must be a mutex.

`mutex-release` releases the mutex identified by *mutex*. Unpredictable behavior results if the mutex is not owned by the calling thread.

`(with-mutex` *mutex* $body_1$ $body_2$ `...)` **syntax**
returns: the values of the body $body_1$ $body_2$ ...
libraries: `(chezscheme)`

`with-mutex` evaluates the expression *mutex*, which must evaluate to a mutex, acquires the mutex, evaluates the body $body_1$ $body_2$ ..., and releases the mutex. The mutex is released whether the body returns normally or via a control operation (that is, throw to a continuation, perhaps because of an error) that results in a nonlocal exit from the `with-mutex` form. If control subsequently returns to the body via a continuation invocation, the mutex is reacquired.

Using `with-mutex` is generally more convenient and safer than using `mutex-acquire` and `mutex-release` directly.

15.3. Conditions

`(make-condition)` **procedure**
returns: a new condition object
libraries: `(chezscheme)`

`(thread-condition?` *obj*`)` **procedure**
returns: `#t` if *obj* is a condition object, `#f` otherwise
libraries: `(chezscheme)`

`(condition-wait` *cond mutex*`)` **procedure**
`(condition-wait` *cond mutex timeout*`)` **procedure**
returns: `#t` if the calling thread was awakened by the condition, `#f` if the calling thread timed out waiting
libraries: `(chezscheme)`

cond must be a condition object, and *mutex* must be a mutex. The optional argument *timeout* is a time record of type `time-duration` or `time-utc`, or `#f` for no timeout. It defaults to `#f`.

`condition-wait` waits up to the specified *timeout* for the condition identified by the condition object *cond*. The calling thread must have acquired the mutex identified by the mutex *mutex* at the time `condition-wait` is called. *mutex* is released as a side effect of the call to `condition-wait`. When a thread is later released from the condition variable by one of the procedures described below or the timeout expires, *mutex* is reacquired and `condition-wait` returns.

`(condition-signal` *cond*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

cond must be a condition object.

`condition-signal` releases one of the threads waiting for the condition identified by *cond*.

`(condition-broadcast` *cond*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

cond must be a condition object.

`condition-broadcast` releases all of the threads waiting for the condition identified by *cond*.

15.4. Locks

Locks are more primitive but more flexible and efficient than mutexes and can be used in situations where the added mutex functionality is not needed or desired. They can also be used independently of the thread system (including in nonthreaded versions of *Chez Scheme*) to synchronize operations running in separate Scheme processes as long as the lock is allocated in memory shared by the processes.

A lock is simply a word-sized integer, i.e., an `iptr` or `uptr` foreign type (Section 4.5) with the native endiannes of the target machine, possibly part of a larger structure defined using `define-ftype` (page 77). It must be explicitly allocated in memory that resides outside the Scheme heap and, when appropriate, explicitly deallocated. When just threads are involved (i.e., when multiple processes are not involved), the memory can be allocated via `foreign-alloc`. When multiple processes are involved, the lock should be allocated in some area shared by the processes that will interact with the lock.

Once initialized using `ftype-init-lock!`, a process or thread can attempt to lock the lock via `ftype-lock!` or `ftype-spin-lock!`. Once the lock has been locked and before it is unlocked, further attempts to lock the lock fail, even by the process or thread that most recently locked it. Locks can be unlocked, via `ftype-unlock!`, by any process or thread, not just by the process or thread that most recently locked the lock.

The lock mechanism provides little structure, and mistakes in allocation and use can lead to memory faults, deadlocks, and other problems. Thus, it is usually advisable to use locks only as part of a higher-level abstraction that ensures locks are used in a disciplined manner.

```
(define lock
  (make-ftype-pointer uptr
    (foreign-alloc (ftype-sizeof uptr))))

(ftype-init-lock! uptr () lock)
(ftype-lock! uptr () lock)  ⇒  #t
(ftype-lock! uptr () lock)  ⇒  #f
(ftype-unlock! uptr () lock)
(ftype-spin-lock! uptr () lock)
(ftype-lock! uptr () lock)  ⇒  #f
(ftype-unlock! uptr () lock)
```

(ftype-init-lock! *ftype-name* (*a* ...) *fptr-expr*) **syntax**
(ftype-init-lock! *ftype-name* (*a* ...) *fptr-expr index*) **syntax**
returns: unspecified
(ftype-lock! *ftype-name* (*a* ...) *fptr-expr*) **syntax**
(ftype-lock! *ftype-name* (*a* ...) *fptr-expr index*) **syntax**
returns: `#t` if the lock is not already locked, `#f` otherwise
(ftype-spin-lock! *ftype-name* (*a* ...) *fptr-expr*) **syntax**
(ftype-spin-lock! *ftype-name* (*a* ...) *fptr-expr index*) **syntax**
returns: unspecified
(ftype-unlock! *ftype-name* (*a* ...) *fptr-expr*) **syntax**
(ftype-unlock! *ftype-name* (*a* ...) *fptr-expr index*) **syntax**
returns: unspecified
libraries: `(chezscheme)`

Each of these has a syntax like and behaves similarly to `ftype-set!` (page 86), though with an implicit *val-expr*. In particular, the restrictions on and handling of *fptr-expr* and the accessors *a* ... is similar, with one important restriction: the field specified by the last accessor, upon which the form operates, must be a word-size integer, i.e., an `iptr`, `uptr`, or the equivalent, with the native endianness.

`ftype-init-lock!` should be used to initialize the lock prior to the use of any of the other operators; if this is not done, the behavior of the other operators is undefined.

`ftype-lock!` can be used to lock the lock. If it finds the lock unlocked at the time of the operation, it locks the lock and returns `#t`; if it finds the lock already locked, it returns `#f` without changing the lock.

`ftype-spin-lock!` can also be used to lock the lock. If it finds the lock unlocked at the time of the operation, it locks the lock and returns; if it finds the lock already locked, it waits until the lock is unlocked, then locks the lock and returns. If no other thread or process unlocks the lock, the operation does not return and cannot be interrupted by normal means, including by the storage manager for the purpose of initiating a garbage collection. There are also no guarantees of fairness, so a process might hang indefinitely even if other processes are actively locking and unlocking the lock.

`ftype-unlock!` is used to unlock a lock. If it finds the lock locked, it unlocks the lock and returns. Otherwise, it returns without changing the lock.

15.5. Locked increment and decrement

The locked operations described here can be used when just an atomic increment or decrement is required.

`(ftype-locked-incr!` *ftype-name* `(`*a* `...)` *fptr-expr*`)` **syntax**
`(ftype-locked-incr!` *ftype-name* `(`*a* `...)` *fptr-expr index*`)` **syntax**
returns: `#t` if the updated value is 0, `#f` otherwise
`(ftype-locked-decr!` *ftype-name* `(`*a* `...)` *fptr-expr*`)` **syntax**
`(ftype-locked-decr!` *ftype-name* `(`*a* `...)` *fptr-expr index*`)` **syntax**
returns: `#t` if the updated value is 0, `#f` otherwise
libraries: `(chezscheme)`

Each of these has a syntax like and behaves similarly to `ftype-set!` (page 86), though with an implicit *val-expr*. In particular, the restrictions on and handling of *fptr-expr* and the accessors *a* `...` is similar, with one important restriction: the field specified by the last accessor, upon which the form operates, must be a word-size integer, i.e., an `iptr`, `uptr`, or the equivalent, with the native endianness.

`ftype-locked-incr!` atomically reads the value of the specified field, adds 1 to the value, and writes the new value back into the field. Similarly, `ftype-locked-decr!` atomically reads the value of the specified field, subtracts 1 from the value, and writes the new value back into the field. Both return `#t` if the new value is 0, otherwise `#f`.

15.6. Thread Parameters

`(make-thread-parameter` *object*`)` **procedure**
`(make-thread-parameter` *object procedure*`)` **procedure**
returns: a new thread parameter
libraries: `(chezscheme)`

See Section 12.13 for a general discussion of parameters and the use of the optional second argument.

When a thread parameter is created, a separate location is set aside in each current and future thread to hold the value of the parameter's internal state variable. (This location may be eliminated by the storage manager when the parameter becomes inaccessible.) Changes to the thread parameter in one thread are not seen by any other thread.

When a new thread is created (see `fork-thread`), the current value (not location) of each thread parameter is inherited from the forking thread by the new thread. Similarly, when a thread created by some other means is activated for the first time (see `Sactivate_thread` in Section 4.8), the current value (not location) of each thread parameter is inherited from the main (original) thread by the new thread.

Most built-in parameters are thread parameters, but some are global. All are marked as global or thread where they are defined. There is no distinction between built-in global and thread parameters in the nonthreaded versions of the system.

15.7. Buffered I/O

Chez Scheme buffers file I/O operations for efficiency, but buffered I/O is not thread safe. Two threads that write to or read from the same buffered port concurrently can corrupt the port, resulting in buffer overruns and, ultimately, invalid memory references.

Buffering on binary output ports can be disabled when opened with buffer-mode `none`. Buffering on input ports cannot be completely disabled, however, due to the need to support lookahead, and buffering on textual ports, even textual output ports, cannot be disabled completely because the transcoders that convert between characters and bytes sometimes require some lookahead.

Two threads should thus *never* read from or write to the same port concurrently, except in the special case of a binary output port opened buffer-mode `none`. Alternatives include appointing one thread to perform all I/O for a given port and providing a per-thread generic-port wrapper that forwards requests to the port only after acquiring a mutex.

The initial console and current input and output ports are thread-safe, as are transcript ports, so it is safe for multiple threads to print error and/or debugging messages to the console. The output may be interleaved, even within the same line, but the port will not become corrupted. Thread safety for these ports is accomplished at the high cost of acquiring a mutex for each I/O operation.

15.8. Example: Bounded Queues

The following code, taken from the article "A Scheme for native threads [10]," implements a bounded queue using many of the thread-system features. A bounded queue has a fixed number of available slots. Attempting to enqueue when the queue is full causes the calling thread to block. Attempting to dequeue from an empty queue causes the calling thread to block.

```
(define-record-type bq
  (fields
    (immutable data)
    (mutable head)
    (mutable tail)
    (immutable mutex)
    (immutable ready)
    (immutable room))
  (protocol
    (lambda (new)
      (lambda (bound)
        (new (make-vector bound) 0 0 (make-mutex)
          (make-condition) (make-condition))))))

(define dequeue!
  (lambda (q)
    (with-mutex (bq-mutex q)
      (let loop ()
```

```
        (let ([head (bq-head q)])
          (cond
            [(= head (bq-tail q))
             (condition-wait (bq-ready q) (bq-mutex q))
             (loop)]
            [else
             (bq-head-set! q (incr q head))
             (condition-signal (bq-room q))
             (vector-ref (bq-data q) head)])))))

(define enqueue!
  (lambda (item q)
    (with-mutex (bq-mutex q)
      (let loop ()
        (let* ([tail (bq-tail q)] [tail^ (incr q tail)])
          (cond
            [(= tail^ (bq-head q))
             (condition-wait (bq-room q) (bq-mutex q))
             (loop)]
            [else
             (vector-set! (bq-data q) tail item)
             (bq-tail-set! q tail^)
             (condition-signal (bq-ready q))])))))

(define incr
  (lambda (q i)
    (modulo (+ i 1) (vector-length (bq-data q)))))
```

The code below demonstrates the use of the bounded queue abstraction with a set of threads that act as consumers and producers of the data in the queue.

```
(define job-queue)
(define die? #f)

(define make-job
  (let ([count 0])
    (define fib
      (lambda (n)
        (if (< n 2)
            n
            (+ (fib (- n 2)) (fib (- n 1))))))
    (lambda (n)
      (set! count (+ count 1))
      (printf "Adding job #~s = (lambda () (fib ~s))\n" count n)
      (cons count (lambda () (fib n))))))

(define make-producer
  (lambda (n)
    (rec producer
      (lambda ()
        (printf "producer ~s posting a job\n" n)
```

```
        (enqueue! (make-job (+ 20 (random 10))) job-queue)
        (if die?
            (printf "producer ~s dying\n" n)
            (producer))))))

(define make-consumer
  (lambda (n)
    (rec consumer
      (lambda ()
        (printf "consumer ~s looking for a job~%" n)
        (let ([job (dequeue! job-queue)])
          (if die?
              (printf "consumer ~s dying\n" n)
              (begin
                (printf "consumer ~s executing job #~s~%" n (car job))
                (printf "consumer ~s computed:  ~s~%" n ((cdr job)))
                (consumer)))))))))

(define (bq-test np nc)
  (set! job-queue (make-bq (max nc np)))
  (do ([np np (- np 1)])
      ((<= np 0))
      (fork-thread (make-producer np)))
  (do ([nc nc (- nc 1)])
      ((<= nc 0))
      (fork-thread (make-consumer nc))))
```

Here are a possible first several lines of output from a sample run of the example program.

```
> (begin
    (bq-test 3 4)
    (system "sleep 3")
    (set! die? #t))
producer 3 posting a job
Adding job #1 = (lambda () (fib 29))
producer 3 posting a job
Adding job #2 = (lambda () (fib 26))
producer 3 posting a job
Adding job #3 = (lambda () (fib 22))
producer 3 posting a job
Adding job #4 = (lambda () (fib 21))
producer 2 posting a job
Adding job #5 = (lambda () (fib 29))
producer 1 posting a job
Adding job #6 = (lambda () (fib 29))
consumer 4 looking for a job
producer 3 posting a job
Adding job #7 = (lambda () (fib 24))
consumer 4 executing job #1
consumer 3 looking for a job
producer 2 posting a job
```

```
Adding job #8 = (lambda () (fib 26))
consumer 3 executing job #2
consumer 3 computed:  121393
consumer 3 looking for a job
producer 1 posting a job
Adding job #9 = (lambda () (fib 26))
...
```

Additional examples, including definitions of suspendable threads and threads that automatically terminate when they become inaccessible, are given in "A Scheme for native threads [10]."

16. Compatibility Features

This chapter describes several items that are included with current versions of *Chez Scheme* primarily for compatibility with older versions of the system.

Section 16.1 describes a hash-table interface that has since been replaced by the R6RS hashtable interface. Section 16.2 describes `extend-syntax` macros. These features are supported directly by current versions of *Chez Scheme*, but support may be dropped in future versions. New programs should use the standard mechanisms described in *The Scheme Programming Language, 4th Edition* [11] instead.

Section 16.3 describes a mechanism for defining record-like structures as vectors instead of new unique types. New programs should use `define-record`, which is described in Section 7.15, instead.

Section 16.4 describes a compatibility file distributed with *Chez Scheme* that contains definitions for forms and procedures no longer supported directly by *Chez Scheme*.

16.1. Hash Tables

The hash table procedures here are obviated by the new hash table procedures listed in Section 7.12.

`(make-hash-table)` **procedure**
`(make-hash-table` *weak?*`)` **procedure**
returns: a new hash table
libraries: `(chezscheme)`

If *weak?* is provided and is non-false, the hash table is a weak hash table, which means that it does not protect keys from the garbage collector. Keys reclaimed by the garbage collector are removed from the table, and their associated values are dropped the next time the table is modified, if not sooner.

`(hash-table?` *obj*`)` **procedure**
returns: `#t` if *obj* is a hash table, otherwise `#f`
libraries: `(chezscheme)`

`(put-hash-table!` *ht k v*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

ht must be a hash table. *k* and *v* may be any Scheme values.

`put-hash-table!` associates the value *v* with the key *k* in *ht*.

`(get-hash-table` *ht k d*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

`get-hash-table` returns the value associated with *k* in *ht*. If no value is associated with *k* in *ht*, `get-hash-table` returns *d*.

Key comparisons are performed with *eq?*.

Because objects may be moved by the garbage collector, `get-hash-table` may need to rehash some objects and therefore cause side effects in the hash table. Thus, it is not safe to perform concurrent accesses of the same hash table from multiple threads using `get-hash-table`.

`(remove-hash-table!` *ht k*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`remove-hash-table!` drops any association for *k* from *ht*.

`(hash-table-map` *ht p*`)` **procedure**
returns: see below
libraries: `(chezscheme)`

`hash-table-map` applies *p* to each key, value association in *ht*, in no particular order, and returns a list of the resulting values, again in no particular order. *p* should accept two arguments, a key and a value.

`(hash-table-for-each` *ht p*`)` **procedure**
returns: unspecified
libraries: `(chezscheme)`

`hash-table-for-each` applies *p* to each key, value association in *ht*, in no particular order. Unlike `hash-table-map`, it does not create a list of the values; instead, it's value is unspecified. *p* should accept two arguments, a key and a value.

16.2. Extend-Syntax Macros

This section describes `extend-syntax`, a powerful yet easy to use syntactic extension facility based on pattern matching [27]. Syntactic transformations written using `extend-syntax` are similar to those written using a `define-syntax` with `syntax-case`, except that the transformations produced by `extend-syntax` do not automatically respect lexical scoping.

It is not typically possible to mix syntactic abstractions written using `syntax-case` with those written using `extend-syntax` seamlessly; it is generally preferable to use one or the other wherever possible. Support for `extend-syntax` within the `syntax-case` expander is provided only as an aid to migrating to `syntax-case`.

`(extend-syntax (`*name key* `...) (`*pat fender template*`) ...)` **syntax**
returns: unspecified
libraries: `(chezscheme)`

The identifier *name* is the name, or syntax keyword, for the syntactic extension to be defined. When the system expander processes any list expression whose car is *name*, the syntactic transformation procedure generated by `extend-syntax` is invoked on this expression. The remaining identifiers *key* ... are additional keywords to be recognized within input expressions during expansion (such as `else` in `cond` or `case`).

Each clause after the list of keys consists of a pattern *pat*, an optional *fender*, and a *template*. The optional *fender* is omitted more often than not. The *pat* specifies the syntax the input expression must have for the clause to be chosen. Identifiers within the pattern that are not keywords (*pattern variables*) are bound to corresponding pieces of the input expression. If present, the *fender* is a Scheme expression that specifies additional constraints on the input expression (accessed through the pattern variables) that must be satisfied in order for the clause to be chosen. The *template* specifies what form the output takes, usually in terms of the pattern variables.

During expansion, the transformation procedure `extend-syntax` generates attempts to match the input expression against each pattern in the order the clauses are given. If the input expression matches the pattern, the pattern variables are bound to the corresponding pieces of the input expression and the fender for the clause, if any, is evaluated. If the fender returns a true value, the given expansion is performed. If input does not match the pattern or if the fender returns a false value, the transformation procedure tries the next clause. An exception is raised with condition type `&assertion` if no clause can be chosen.

Within the pattern, *ellipsis* (...) may be used to specify zero or more occurrences of the preceding pattern fragment, or prototype. Similarly, ellipses may be used in the output to specify the construction of zero or more expansion prototypes. In this case, the expansion prototype must contain part of an input pattern prototype. The use of patterns, templates, ellipses within patterns and templates, and fenders is illustrated in the following sequence of examples.

The first example, defining `rec`, uses a single keyword, a single clause with no fender, and no ellipses.

```
(extend-syntax (rec)
  [(rec id val)
   (let ([id #f])
     (set! id val)
     id)])
```

The second example, defining `when`, shows the use of ellipses.

```
(extend-syntax (when)
  [(when test exp1 exp2 ...)
   (if test (begin exp1 exp2 ...) #f)])
```

The next example shows the definition of `let`. The definition of `let` shows the use of multiple ellipses, employing one for the identifier/value pairs and one for the expressions in the body. It also shows that the prototype need not be a single identifier, and that pieces of the prototype may be separated from one another in the template.

```
(extend-syntax (let)
  [(let ([x e] ...) b1 b2 ...)
   ((lambda (x ...) b1 b2 ...) e ...)])
```

The next example shows `let*`, whose syntax is the same as for `let`, but which is defined recursively in terms of `let` with two clauses (one for the base case, one for the recursion step) since it must produce a nested structure.

```
(extend-syntax (let*)
  [(let* () b1 b2 ...)
   (let () b1 b2 ...)]
  [(let* ([x e] more ...) b1 b2 ...)
   (let ([x e]) (let* (more ...) b1 b2 ...))])
```

The first pattern/template pair matches any `let*` expression with no identifier/value pairs and maps it into the equivalent `begin` expression. This is the base case. The second pattern/template pair matches any `let*` expression with one or more identifier/value pairs and transforms it into a `let` expression binding the first pair whose body is a `let*` expression binding the remaining pairs. This is the recursion step, which will eventually lead us to the base case because we remove one identifier/value pair at each step. Notice that the second pattern uses the pattern variable `more` for the second and later identifier/value pairs; this makes the pattern and template less cluttered and makes it clear that only the first identifier/value pair is dealt with explicitly.

The definition for `and` requires three clauses. The first clause is necessary to recognize `(and)`, and the second two define all other `and` forms recursively.

```
(extend-syntax (and)
  [(and) #t]
  [(and x) x]
  [(and x y ...) (if x (and y ...) #f)])
```

The definition for `cond` requires four clauses. As with `let*`, `cond` must be described re-

cursively, partly because it produces nested `if` expressions, and partly because one ellipsis prototype would not be sufficient to describe all possible `cond` clauses. The definition of `cond` also requires that we specify `else` as a keyword, in addition to `cond`. Here is the definition:

```
(extend-syntax (cond else)
  [(cond) #f]
  [(cond (else e1 e2 ...))
   (begin e1 e2 ...)]
  [(cond (test) more ...)
   (or test (cond more ...))]
  [(cond (test e1 e2 ...) more ...)
   (if test
       (begin e1 e2 ...)
       (cond more ...))])
```

Two of the clauses are base cases and two are recursion steps. The first base case is an empty `cond`. The value of `cond` in this case is unspecified, so the choice of `#f` is somewhat arbitrary. The second base case is a `cond` containing only an `else` clause; this is transformed to the equivalent `begin` expression. The two recursion steps differ in the number of expressions in the `cond` clause. The value of `cond` when the first true clause contains only the test expression is the value of the test. This is similar to what `or` does, so we expand the `cond` clause into an `or` expression. On the other hand, when there are expressions following the test expression, the value of the last expression is returned, so we use `if` and `begin`.

To be absolutely correct about the syntax of `let`, we actually must require that the bound identifiers in the input are symbols. If we typed something like `(let ([3 x]) x)` we would not get an error from `let` because it does not check to verify that the objects in the identifier positions are symbols. Instead, `lambda` may complain, or perhaps the system evaluator long after expansion is complete. This is where fenders are useful.

```
(extend-syntax (let)
  [(let ([x e] ...) b1 b2 ...)
   (andmap symbol? '(x ...))
   ((lambda (x ...) b1 b2 ...) e ...)])
```

The `andmap` of `symbol?` over `'(x ...)` assures that each bound identifier is a symbol. A fender is simply a Scheme expression. Within that expression, any quoted object is first expanded by the same rules as the template part of the clause. In this case, `'(x ...)` is expanded to the list of identifiers from the identifier/value pairs.

`extend-syntax` typically handles everything you need it for, but some syntactic extension definitions require the ability to include the result of evaluating an arbitrary Scheme expression. This ability is provided by `with`.

`(with ((`*pat expr*`) ...)` *template*`)` **syntax**
returns: processed *template*

`with` is valid only within an template inside of `extend-syntax`. `with` patterns are the same

as `extend-syntax` patterns, `with` expressions are the same as `extend-syntax` fenders, and `with` templates are the same as `extend-syntax` templates.

`with` can be used to introduce new pattern identifiers bound to expressions produced by arbitrary Scheme expressions within `extend-syntax` templates. That is, `with` allows an escape from the declarative style of `extend-syntax` into the procedural style of full Scheme.

One common use of `with` is the introduction of a temporary identifier or list of temporary identifiers into a template. `with` is also used to perform complex transformations that might be clumsy or inefficient if performed within the `extend-syntax` framework.

For example, `or` requires the use of a temporary identifier. We could define `or` as follows.

```
(extend-syntax (or)
  [(or) #f]
  [(or x) x]
  [(or x y ...)
   (let ([temp x])
     (if temp temp (or y ...)))])
```

This would work until we placed an `or` expression within the scope of an occurrence of `temp`, in which case strange things could happen, since `extend-syntax` does not respect lexical scoping. (This is one of the reasons that `define-syntax` is preferable to `extend-syntax`.)

```
(let ([temp #t])
  (or #f temp))  ⇒  #f
```

One solution is to use `gensym` and `with` to create a temporary identifier, as follows.

```
(extend-syntax (or)
  [(or) #f]
  [(or x) x]
  [(or x y ...)
   (with ([temp (gensym)])
     (let ([temp x])
       (if temp temp (or y ...))))])
```

Also, `with` can be used to combine elements of the input pattern in ways not possible directly with `extend-syntax`, such as the following `folding-plus` example.

```
(extend-syntax (folding-plus)
  [(folding-plus x y)
   (and (number? 'x) (number? 'y))
   (with ([val (+ 'x 'y)])
      val)]
  [(folding-plus x y) (+ x y)])
```

`folding-plus` collapses into the value of `(+ x y)` if both `x` and `y` are numeric constants. Otherwise, `folding-plus` is transformed into `(+ x y)` for later evaluation. The fender checks that the operands are numbers at expansion time, and the `with` performs the evaluation. As with fenders, expansion is performed only within a quoted expressions, since `quote` sets the data apart from the remainder of the Scheme expression.

The example below binds a list of pattern variables to a list of temporary symbols, taking advantage of the fact that `with` allows us to bind patterns to expressions. This list of temporaries helps us to implement the `sigma` syntactic extension. `sigma` is similar to `lambda`, except it assigns the identifiers in the identifier list instead of creating new bindings. It may be used to perform a series of assignments in parallel.

```
(extend-syntax (sigma)
  [(sigma (x ...) e1 e2 ...)
   (with ([(t ...) (map (lambda (x) (gensym)) '(x ...))])
     (lambda (t ...)
       (set! x t) ...
       e1 e2 ...))])

(let ([x 'a] [y 'b])
  ((sigma (x y) (list x y)) y x))  ⇒  (b a)
```

The final example below uses `extend-syntax` to implement `define-structure`, following a similar example using `syntax-case` in Section 8.4 of *The Scheme Programming Language, 4th Edition*.

The definition of `define-structure` makes use of two pattern/template clauses. Two clauses are needed to handle the optionality of the second subexpression. The first clause matches the form without the second subexpression and merely converts it into the equivalent form with the second subexpression present, but empty.

The definition also makes heavy use of `with` to evaluate Scheme expressions at expansion time. The first four `with` clauses are used to manufacture the identifiers that name the automatically defined procedures. (The procedure `format` is particularly useful here, but it could be replaced with `string-append!`, using `symbol->string` as needed.) The first two clauses yield single identifiers (for the constructor and predicate), while the next two yield lists of identifiers (for the field access and assignment procedures). The fifth `with` clause (the final clause in the outer `with`) is used to count the total length vector needed for each instance of the structure, which must include room for the name and all of the fields. The final `with` clause (the only clause in the inner `with`) is used to create a list of vector indexes, one for each field (starting at 1, since the structure name occupies position 0).

```
(extend-syntax (define-structure)
  [(define-structure (name id1 ...))
   (define-structure (name id1 ...) ())]
  [(define-structure (name id1 ...) ([id2 val] ...))
   (with ([constructor
           (string->symbol (format "make-~a" 'name))]
          [predicate
           (string->symbol (format "~a?" 'name))]
          [(access ...)
           (map (lambda (x)
                  (string->symbol
                    (format "~a-~a" 'name x)))
                '(id1 ... id2 ...))]
          [(assign ...)
```

```
      (map (lambda (x)
             (string->symbol
               (format "set-~a-~a!" 'name x)))
           '(id1 ... id2 ...))]
     [count (length '(name id1 ... id2 ...))])
(with ([(index ...)
        (let f ([i 1])
          (if (= i 'count)
              '()
              (cons i (f (+ i 1)))))])
  (begin
    (define constructor
      (lambda (id1 ...)
        (let* ([id2 val] ...)
          (vector 'name id1 ... id2 ...))))
    (define predicate
      (lambda (obj)
        (and (vector? obj)
             (= (vector-length obj) count)
             (eq? (vector-ref obj 0) 'name))))
    (define access
      (lambda (obj)
        (vector-ref obj index)))
    ...
    (define assign
      (lambda (obj newval)
        (vector-set! obj index newval)))
    ...))])
```

16.3. Structures

This section describes a mechanism, similar to the record-defining mechanisms of Section 7.15, that permits the creation of data structures with fixed sets of named fields. Unlike record types, structure types are not unique types, but are instead implemented as vectors. Specifically, a structure is implemented as a vector whose length is one more than the number of fields and whose first element contains the symbolic name of the structure.

The representation of structures as vectors simplifies reading and printing of structures somewhat as well as extension of the structure definition facility. It does, however, have some drawbacks. One is that structures may be treated as ordinary vectors by mistake in situations where doing so is inappropriate. When dealing with both structures and vectors in a program, care must be taken to look for the more specific structure type before checking for the more generic vector type, e.g., in a series of `cond` clauses. A similar drawback is that structure instances are easily "forged," either intentionally or by accident. It is also impossible to control how structures are printed and read.

Structures are created via `define-structure`. Each structure definition defines a constructor procedure, a type predicate, an access procedure for each of its fields, and an assignment

procedure for each of its fields. `define-structure` allows the programmer to control which fields are arguments to the generated constructor procedure and which fields are explicitly initialized by the constructor procedure.

`define-structure` is simple yet powerful enough for most applications, and it is easily extended to handle many applications for which it is not sufficient. The definition of `define-structure` given at the end of this section can serve as a starting point for more complicated variants.

`(define-structure (`*name* id_1 `...) ((`id_2 *expr*`) ...))` **syntax**
returns: unspecified
libraries: `(chezscheme)`

A `define-structure` form is a definition and may appear anywhere and only where other definitions may appear.

`define-structure` defines a new data structure, *name*, and creates a set of procedures for creating and manipulating instances of the structure. The identifiers id_1 ... and id_2 ... name the fields of the data structure.

The following procedures are defined by `define-structure`:

- a constructor procedure whose name is `make-`*name*,
- a type predicate whose name is *name*`?`,
- an access procedure whose name is *name*-*field* for each field name id_1 ... and id_2 ..., and
- an assignment procedure whose name is `set-`*name*-*field*`!` for each field name id_1 ... and id_2 ....

The fields named by the identifiers id_1 ... are initialized by the arguments to the constructor procedure. The fields named by the identifiers id_2 ... are initialized explicitly to the values of the expressions *expr* .... Each expression is evaluated within the scope of the identifiers id_1 ... (bound to the corresponding field values) and any of the identifiers id_2 ... (bound to the corresponding field values) appearing before it (as if within a `let*`).

To clarify, the constructor behaves as if defined as

```
(define make-name
  (lambda (id1 ...)
    (let* ([id2 expr] ...)
      body)))
```

where *body* builds the structure from the values of the identifiers id_1 ... and id_2 ....

If no fields other than those initialized by the arguments to the constructor procedure are needed, the second subexpression, `((`id_2 *expr*`) ...)`, may be omitted.

The following simple example demonstrates how pairs might be defined in Scheme if they did not already exist. Both fields are initialized by the arguments to the constructor procedure.

```
(define-structure (pare car cdr))
(define p (make-pare 'a 'b))

(pare? p)  ⇒  #t
(pair? p)  ⇒  #f
(pare? '(a . b))  ⇒  #f

(pare-car p)  ⇒  a
(pare-cdr p)  ⇒  b

(set-pare-cdr! p (make-pare 'b 'c))

(pare-car (pare-cdr p))  ⇒  b
(pare-cdr (pare-cdr p))  ⇒  c
```

The following example defines a handy string data structure, called a *stretch-string*, that grows as needed. This example uses a field explicitly initialized to a value that depends on the value of the constructor argument fields.

```
(define-structure (stretch-string length fill)
  ([string (make-string length fill)]))

(define stretch-string-ref
  (lambda (s i)
    (let ([n (stretch-string-length s)])
      (when (>= i n) (stretch-stretch-string! s (+ i 1) n))
      (string-ref (stretch-string-string s) i))))

(define stretch-string-set!
  (lambda (s i c)
    (let ([n (stretch-string-length s)])
      (when (>= i n) (stretch-stretch-string! s (+ i 1) n))
      (string-set! (stretch-string-string s) i c))))

(define stretch-string-fill!
  (lambda (s c)
    (string-fill! (stretch-string-string s) c)
    (set-stretch-string-fill! s c)))

(define stretch-stretch-string!
  (lambda (s i n)
    (set-stretch-string-length! s i)
    (let ([str (stretch-string-string s)]
          [fill (stretch-string-fill s)])
      (let ([xtra (make-string (- i n) fill)])
        (set-stretch-string-string! s
          (string-append str xtra))))))
```

As often happens, most of the procedures defined automatically are used only to define more specialized procedures, in this case the procedures `stretch-string-ref` and `stretch-string-set!`. `stretch-string-length` and `stretch-string-string` are the only automatically defined procedures that are likely to be called directly in code that uses

stretch strings.

```
(define ss (make-stretch-string 2 #\X))

(stretch-string-string ss)  ⇒  "XX"
(stretch-string-ref ss 3)  ⇒  #\X
(stretch-string-length ss)  ⇒  4
(stretch-string-string ss)  ⇒  "XXXX"

(stretch-string-fill! ss #\@)
(stretch-string-string ss)  ⇒  "@@@@"
(stretch-string-ref ss 5)  ⇒  #\@
(stretch-string-string ss)  ⇒  "@@@@@@"

(stretch-string-set! ss 7 #\=)
(stretch-string-length ss)  ⇒  8
(stretch-string-string ss)  ⇒  "@@@@@@@="
```

Section 8.4 of *The Scheme Programming Language, 4th Edition* defines a simplified variant of `define-structure` as an example of the use of `syntax-case`. The definition given below implements the complete version.

`define-structure` expands into a series of definitions for names generated from the structure name and field names. The generated identifiers are created with `datum->syntax` to make the identifiers visible where the `define-structure` form appears. Since a `define-structure` form expands into a `begin` containing definitions, it is itself a definition and can be used wherever definitions are valid.

```
(define-syntax define-structure
  (lambda (x)
    (define gen-id
      (lambda (template-id . args)
        (datum->syntax template-id
          (string->symbol
            (apply string-append
                   (map (lambda (x)
                          (if (string? x)
                              x
                              (symbol->string
                                (syntax->datum x))))
                        args))))))
    (syntax-case x ()
      ((_ (name field1 ...))
       (andmap identifier? #'(name field1 ...))
       #'(define-structure (name field1 ...) ()))
      ((_ (name field1 ...) ((field2 init) ...))
       (andmap identifier? #'(name field1 ... field2 ...))
       (with-syntax
         ((constructor (gen-id #'name "make-" #'name))
          (predicate (gen-id #'name #'name "?"))
          ((access ...)
```

```
  (map (lambda (x) (gen-id x #'name "-" x))
       #'(field1 ... field2 ...)))
 ((assign ...)
  (map (lambda (x) (gen-id x "set-" #'name "-" x "!"))
       #'(field1 ... field2 ...)))
 (structure-length
  (+ (length #'(field1 ... field2 ...)) 1))
 ((index ...)
  (let f ([i 1] [ids #'(field1 ... field2 ...)])
    (if (null? ids)
        '()
        (cons i (f (+ i 1) (cdr ids)))))))
#'(begin
    (define constructor
      (lambda (field1 ...)
        (let* ([field2 init] ...)
          (vector 'name field1 ... field2 ...))))
    (define predicate
      (lambda (x)
        (and (vector? x)
             (#3%fx= (vector-length x) structure-length)
             (eq? (vector-ref x 0) 'name))))
    (define access (lambda (x) (vector-ref x index)))
    ...
    (define assign
      (lambda (x update) (vector-set! x index update)))
    ...))))))
```

16.4. Compatibility File

Current versions of *Chez Scheme* are distributed with a compatibility file containing definitions of various syntactic forms and procedures supported by earlier versions of *Chez Scheme* for which support has since been dropped. This file, `compat.ss`, is typically installed in the library subdirectory of the *Chez Scheme* installation directory.

In some cases, the forms and procedures found in `compat.ss` have been dropped because they were infrequently used and easily written directly in Scheme. In other cases, the forms and procedures have been rendered obsolete by improvements in the system. In such cases, new code should be written to use the newer features, and older code should be rewritten if possible to use the newer features as well.

References

[1] Michael Adams and R. Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 179–188, September 2008.

[2] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, 140–149, June 1994.

[3] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, 99–107, May 1996.

[4] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, 108–116, May 1996.

[5] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proceedings of the IEEE Computer Society 1998 International Conference on Computer Languages*, 240–251, May 1998.

[6] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, 130–138, June 1995.

[7] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina, Chapel Hill, April 1987.

[8] R. Kent Dybvig. Writing hygienic macros in scheme with syntax-case. Technical Report 356, Indiana Computer Science Department, June 1992.

[9] R. Kent Dybvig. The development of Chez Scheme. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 1–12, September 2006.

[10] R. Kent Dybvig. A Scheme for native threads. In *Symposium in Honor of Mitchell Wand*, August 2009. *http://www.ccs.neu.edu/events/wand-symposium/*.

[11] R. Kent Dybvig. *The Scheme Programming Language*, 4th edition. MIT Press, 2009.

[12] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 207–216, June 1993.

[13] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BiBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana Computer Science Department, March 1994.

[14] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[15] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[16] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, September 1990.

[17] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[18] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Indiana Computer Science Department, February 1990.

[19] Abdulaziz Ghuloum. *Implicit phasing for library dependencies*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2008. Adviser-Dybvig, R. Kent.

[20] Abdulaziz Ghuloum and R. Kent Dybvig. Generation-friendly eq hash tables. In *2007 Workshop on Scheme and Functional Programming*, 27–35, 2007. *http://sfp2007.ift.ulaval.ca/programme.html*.

[21] Abdulaziz Ghuloum and R. Kent Dybvig. Implicit phasing for R6RS libraries. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, 303–314, 2007. *http://doi.acm.org/10.1145/1291220.1291197*.

[22] Abdulaziz Ghuloum and R. Kent Dybvig. Fixing letrec (reloaded). In *2009 Workshop on Scheme and Functional Programming*, August 2009. *http://www.schemeworkshop.org/2009/*.

[23] Barry Hayes. Ephemerons: a new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Languages, Programming, Systems, and Applications*, 176–183, 1997. *https://doi.org/10.1145/263700.263733*.

[24] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[25] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, 66–77, June 1990.

[26] IEEE Computer Society. *IEEE Standard for the Scheme Programming Language*, May 1991. IEEE Std 1178-1990.

[27] Eugene Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, August 1986.

[28] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007. *http://www.r6rs.org/*.

[29] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 report on the algorithmic language Scheme—non-normative appendices, September 2007. *http://www.r6rs.org/*.

[30] Guy L. Steele Jr. *Common Lisp, the Language*, second edition. Digital Press, 1990.

[31] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In *Fourth International Symposium on Static Analysis*, volume 1302 of *Springer-Verlag Lecture Notes in Computer Science*, 35–52. Springer-Verlag, 1997.

[32] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of the 26th Annual ACM Symposium on Principles of Programming Languages*, 203–213, January 1999.

[33] Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-order and symbolic computation*, 18(3/4):299–326, 2005.

Summary of Forms

The table that follows summarizes the syntactic forms and procedures described in this book along with standard Scheme syntactic forms and procedures. It shows each item's category and the page number where it is defined. Page numbers prefixed by "t" refer to *The Scheme Programming Language, 4th Edition* (TSPL4).

Form	*Category*	*Page*
'*obj*	syntax	t141
`*obj*	syntax	t142
,*obj*	syntax	t142
,@*obj*	syntax	t142
=>	syntax	t112
_	syntax	t297
...	syntax	t297
#'*template*	syntax	t300
#`*template*	syntax	t305
#,*template*	syntax	t305
#,@*template*	syntax	t305
#%*variable*	syntax	348
#2%*variable*	syntax	348
#3%*variable*	syntax	348
($primitive *variable*)	syntax	348
($primitive 2 *variable*)	syntax	348
($primitive 3 *variable*)	syntax	348
$system	module	305
&assertion	syntax	t366
&condition	syntax	t362
&continuation	syntax	319
&error	syntax	t367
&format	syntax	318
&i/o	syntax	t371
&i/o-decoding	syntax	t375
&i/o-encoding	syntax	t376
&i/o-file-already-exists	syntax	t374
&i/o-file-does-not-exist	syntax	t374
&i/o-file-is-read-only	syntax	t374
&i/o-file-protection	syntax	t373
&i/o-filename	syntax	t373
&i/o-invalid-position	syntax	t372

Index

This index is a unified index for this book and *The Scheme Programming Language, 4th Edition* (TSPL4). Page numbers prefixed by "t" refer the latter document. Italicized page numbers refer to the primary description of a syntactic form or procedure.