Chez Scheme Version 7.9.4 Release Notes
Copyright © 2009 Cadence Research Systems
All Rights Reserved
December 2009

Contents

1. Overview

This document outlines the changes made to *Chez Scheme* for Version 7.9.4 since Version 7.4, most of which are focused on converting *Chez Scheme* into an implementation of the new Scheme standard described in the Revised[6] Report on Scheme.

Version 7.9.4 is available for the following platforms:

- Linux x86, threaded and nonthreaded
- Linux x86_64, threaded and nonthreaded
- MacOS X x86, threaded and nonthreaded
- MacOS X x86_64, threaded and nonthreaded
- MacOS X PowerPC, threaded and nonthreaded
- Windows x86, threaded and nonthreaded
- OpenBSD x86, threaded and nonthreaded
- OpenBSD x86_64, threaded and nonthreaded
- FreeBSD x86, threaded and nonthreaded
- Solaris 32-bit Sparc, threaded and nonthreaded
- Solaris 64-bit Sparc, threaded and nonthreaded

This document contains three sections describing significant (1) functionality changes, (2) bugs fixed, and (3) performance enhancements. A version number listed in parentheses in the header for a change indicates the first minor release or internal prerelease to support the change. Many other changes, too numerous to list, have been made to improve reliability, performance, and the quality of error messages and source information produced by the system.

Version 7.9.4 is a prerelease of Version 8. Few additional changes are expected in Version 8, though some may be incompatible with Version 7.9.4. More information on *Chez Scheme* and *Petite Chez Scheme* can be found at http://www.scheme.com, and extensive documentation is available in *The Scheme Programming Language, 4th edition* (available directly from MIT Press or from online and local retailers) and the draft *Chez Scheme Version 8 User's Guide*. Online versions of both books can be found at http://www.scheme.com.

2. Functionality Changes

2.1. Automatic library compilation (7.9.4)

When the new parameter `compile-imported-libraries` is set to `#t`, the expander automatically calls `compile-library` on any imported library before it loads it from the file system, as described in the entry below. The default initial value of this parameter is `#f`. It is set to `#t` via the command-line option `--compile-imported-libraries`.

2.2. Library directories and extensions (7.9.4/7.9.3)

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. 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`.

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 below for the `--libdirs` and `--libexts` command-line options.

The `--libdirs` and `--libexts` options can be used to set `library-directories` and `library-extensions` parameters from the command line. The format of the arguments to these options 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 existing list; otherwise, the specified pairs replace the existing list. 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.

These parameters are consulted by the expander when it encounters an import for a library that has not previously been defined or loaded. The expander searches for the library source or object file in each pair of root directories, in order, trying each of the source extensions then each of the object extensions in turn before moving onto the next pair of root directories. If the expander finds a source file before it finds an object file, it loads the corresponding object file if the object file exists and is not older than the source file.

If this is not the case, and the parameter `compile-imported-libraries` is set to `#t`, the expander compiles the library via `compile-library`, then loads the resulting object file. Otherwise, the expander loads the source 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.

Whenever the expander determines it must compile a library to a file or loads 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 found in or relative to its own directory.

2.3. New top-level `begin` semantics (7.9.4)

Groups of definitions within a top-level `begin` are now treated in a manner similar to `lambda`, `library`, and top-level program bodies. This eliminates some ordering constraints on definitions appearing within a top-level `begin` expression. For example:

```
(begin
  (define-syntax a (identifier-syntax 3))
  (define x a))
```

and

```
(begin
  (define x a)
  (define-syntax a (identifier-syntax 3)))
```

are now equivalent. 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. More generally, this reduces the difference between groups of forms wrapped in a top-level `begin` form and groups of forms appearing with a `lambda`, `library`, or top-level program body, making it easier to write code or macros that expand into code that operates correctly in all contexts.

2.4. New foreign-data manipulation routines (7.9.4)

For new procedures have been added for manipulating foreign data: `foreign-alloc`, which allocates a block of memory outside of the Scheme heap, `foreign-free`, which frees a block previously allocated by `foreign-alloc`, `foreign-set!`, which writes a new value to a foreign memory location, and `foreign-ref`, which reads a value from a foreign memory location. The type of data written or read from a foreign memory location is specified by a type symbol, one of the machine-dependent "C" types `short`, `unsigned-short`, `int`, `unsigned`, `unsigned-int` (an alias for `unsigned`), `long`, `unsigned-long`, `long-long`, `unsigned-long-long`, `char`, `wchar`, `float`, `double`, `iptr` (an integer the size of a pointer), `uptr` (an unsigned integer the size of a

pointer), or `void*` (an alias for `uptr`); one of the fixed-sized types `double-float`, `single-float`, `integer-8`, `unsigned-8`, `integer-16`, `unsigned-16`, `integer-32`, `unsigned-32`, `integer-64`, or `unsigned-64`; or one of the additional types `boolean` (a zero or nonzero `int` on the C side, `#f` or `#t` on the Scheme side) or `fixnum` (an `iptr` limited to fixnum range).

The new procedure `foreign-sizeof` can be used to determine the size (in bytes) of a foreign type. It accepts a symbolic type argument and returns the size as an exact nonnegative integer.

2.5. Extended foreign-procedure types (7.9.4)

The set of `foreign-procedure` and `foreign-callable` argument and result types has been extended to include all of the types listed above for `foreign-ref` and `foreign-set!`. That is, to the set supported by previous versions, Version 7.9.4 adds `short`, `unsigned-short`, `long-long`, `unsigned-long-long`, `char`, `wchar`, `float`, `double`, `void*`, `integer-8`, `unsigned-8`, `integer-16`, and `unsigned-16`. It also makes `integer-64` and `unsigned-64` available even on 32-bit machines. Version 7.9.4 also adds `ptr` as an alias for `scheme-object`.

A set of new types for passing Scheme bytevectors and strings between Scheme and C have been added: `u8*`, `u16*`, `u32*`, `utf-8`, `utf-16le`, `utf-16be`, `utf-32le`, and `utf-32be`.

The types `u8*`, `u16*`, and `u32*` are used to pass or return the address of a bytevector's data from Scheme to C or to pass or return a null-terminated string of 8-, 16-, or 32-bit integers from C to Scheme. Null termination is required only when passing or returning data from C to Scheme and is used to determine the length of the returned sequence of integers. Going the other direction, if the receiving C code requires the data to be null-terminated, the null terminator should be included explicitly in the bytevector. Because Scheme bytevectors and strings can include zero elements, it is often better to pass the length as an explicit additional argument.

The types `utf-8`, `utf-16le`, `utf-16be`, `utf-32le`, and `utf-32be` are used to pass Unicode strings between C and Scheme. When going from Scheme to C, the Scheme string is converted to a Scheme bytevector, using the appropriate conversion with the appropriate null terminator, and a pointer to the bytevector's data is passed to C. When going from C to Scheme, the value should be a null-terminated sequence of 8-, 16-, or 32-bit integers and is converted via the appropriate conversion to a Scheme string.

The old `string` type is now an alias for `utf-8`. While previous releases did not allow `string` to be used as a `foreign-callable` return type, Version 7.9.4 allows `string` and all of the other bytevector and string types to be used as `foreign-callable` return types. Because the C code receives a pointer into a Scheme object, and Scheme objects can be moved or destroyed, with their storage reclaimed for other purposes by the Scheme garbage collector, the C code should not retain pointers it receives via any of the bytevector and string types after it returns to or calls into Scheme.

A new `wstring` type has also been added and is an alias for `utf-16le`, `utf-16be`, `utf-32le`, or `utf-32be` depending on the size of a C `wchar_t` and the endianness of the host machine.

The valid set of result types also still includes `void`.

In an incompatible change from previous prereleases, bytevectors are no longer permissible as `string` inputs, and bytevectors are no longer implicitly null terminated.

2.6. UTF-16LE, UTF-16BE codecs and changes to utf-16-codec (7.9.4)

Support for UTF-16LE and UTF-16BE codecs has been added. The procedure `utf-16le-codec` returns a codec that can be used to read or write files in the little-endian form of UTF-16, with no byte-order mark (BOM), and the procedure `utf-16be-codec` returns a codec that can be used to read or write files in the big-endian form of UTF-16, also with no BOM.

For the UTF-16 codec returned by `utf-16-codec`, Version 7.9.4 handles byte-order marks differently from earlier prereleases. First, Version 7.9.4 emits a BOM just before the first character written to a port

transcoded with a transcoder based on the UTF-16 codec. For textual ports created via `transcoded-port`, the BOM 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. Second, when the system can determine that a transcoded port is positioned at the start of a data stream or file, it automatically positions the port at the start of the data beyond the BOM when an attempt is made to reposition the port at the beginning of the file, so that the BOM is preserved. Finally, 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.

The `utf-16-codec` procedure can now be called with an optional argument that specifies the default endiannes. The argument defaults to the symbol `big` but can also be the symbol `little`. The codec returned by `(utf-16-codec 'little)` behaves just like its statndard counterpart, but if no BOM is present when the file is first read, the file is assumed to be encoded in little-endian format rather than big-endian format. This allows programs to read files claimed to be Unicode files but actually encoded in UTF-16LE with no BOM, while still handling those that might have BOMs. It also allows programs to create BOM-prefixed files that use little-endian rather than the default big-endian format.

2.7. Support for non-Unicode character sets (7.9.4)

Support for handling non-Unicode character sets has been added.

For non-Windows systems that provide built-in support for `iconv` and for Windows systems upon which an `iconv` dynamic-link library (DLL) can be loaded, the `iconv-codec` procedure takes a string naming a code page and returns a codec that can be used as the basis of a transcoder used to transcode files or convert between bytevectors and strings. 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.

Two additional procedures are available under Windows. The procedure `multibyte->string` is a wrapper for the Windows API `MultiByteToWideChar` function. It takes a code-page identifier and a bytevector containing a sequence of multibyte characters and converts them to a Scheme string containing the corresponding sequence of whole characters. Similarly, `string->multibyte` is a wrapper for the Windows API `WideCharToMultiByte` function. It takes a code-page identifier and a string and returns a bytevector containing a sequence of the corresponding multibyte characters. A code-page identifier is an exact nonnegative integer 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 like-named API constants.

2.8. New `current-transcoder` parameter (7.9.4)

The new parameter `current-transcoder`, which defaults to the value of `(native-transcoder)`, determines the transcoder used when textual files are opened by operations for which an explicit transcoder is not specified, e.g., `open-input-file`, `open-output-file`, `load`, `compile-file`, and `include`.

2.9. New `transcoder?` predicate (7.9.4)

The new procedure `transcoder?`, which should be standard but is not, returns `#t` if its argument is a transcoder and `#f` otherwise.

2.10. New port and file operations (7.9.4)

The new procedure `file-port?` returns true if its port argument is a file port, which includes 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 port). The new procedure `port-file-descriptor` returns the file descriptor assocated with a file port.

The new procedure `get-mode` retrieves the permissions on the file named by its string argument and returns them as a exact integer in the same form as the mode argument to `chmod`. If the optional *follows?* argument is supplied and is `#f`, If *follow?* is specified and `#f`, symbolic links are not followed; othewise, they are followed.

Three procedures have been added to allow the times recorded for a file to be retrieved:

- `file-access-time`, which returns a time record representing the time of last access of a file,
- `file-modification-time`, which returns a time record representing the time of last modification to a file, and
- `file-change-time`, which returns a time record representing the time of last change to a file, which might include a change in permissions.

Each of these takes a string pathname or file port. Each also takes an optional *follow?* argument, as for `get-mode`. The *follow?* argument is ignored when the first argument is a port.

2.11. New and improved pathname operations (7.9.4)

The new procedures `path-first` and `path-rest` return the first component of a pathname and the remaining components of the pathname respectively. For example, `(path-first "a/b/c")` returns `"a"` and `(path-last "a/b/c")` returns `"b/c"`. For pathnames that start with a root directory (including drives and shares under Windows), home directory (e.g., `~/abc` or `~user/abc`), or a reference to the current or parent directory (`.` or `..`), the first component is that directory. The existing procedures `path-last`, `path-parent` `path-root`, and `path-extension` have been made more sensitive to the overall structure of a pathname to avoid giving nonsensical or ambiguous results.

2.12. Console error port (7.9.4)

The new parameter `console-error-port` determines the port to which conditions and other messages are displayed by `default-exception-handler`, which is initial value of the `base-exception-handler` parameter that determines how uncaught exceptions are handled. When the system determines that the standard output and standard error streams are directed to the same place (tty/console, file, pipe, etc.) the initial value of this parameter is the same as the initial value of `console-output-port`. Otherwise, the initial value of this parameter is a different port tied to the standard error stream of the Scheme process. The initial value of `current-error-port` is the same as the initial value of `console-error-port`.

2.13. Debug changes (7.9.4)

The continuation of a failed `assert` or a call to `assertion-violation`, `assertion-violationf`, `error`, `errorf`, or `syntax-error` is now included within the conditions they pass to `raise`. Conditions encapsulating raise continuations (conditions of type `&continuation`) can also be created explicitly by the procedure `make-continuation-condition`. The predicate `continuation-condition?` returns `#t` if its argument is a continuation condition, and the procedure `condition-continuation` returns the continuation encapsulated within a continuation condition.

When the default exception handler is invoked with any serious or non-warning condition, including one that encapsulates a continuation, it displays the condition and also saves it for possible later use by the debugger in the `debug-condition` parameter. The `debug` procedure in turn allows the condition and/or the continuation encapsulated within it to be inspected.

In previous prereleases, the continuation made available to the debugger was the continuation of the call to the default exception handler, but this is often the wrong continuation. For example, if a guard has caught

and reraised the exception, perhaps after performing some clean-up action like closing a file, the continuation is that of the `raise` call in the guard, not of the original raise.

Ordinarily, the user must type `debug` explicitly to enter the debugger and inspect the last debug condition or its continuation. If, however, the parameter `debug-on-exception` is set to `#t`, the default exception handler directly enters the debugger, from which the continuation (control stack) of the exception can be inspected. The `--debug-on-exception` 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` options.

2.14. More programmer control over collection (7.9.4)

The storage manager now gives the programmer more control over the garbage collector. First, the `collect` procedure allows the programmer to specify the target generation for each collection, and even cause objects to be collected into a static generation previously used only when compacting the heap. Second, `collect-maximum-generation` is now a parameter that can be used to alter the number of generations used by the collector. Third, `bytes-allocated` now takes an optional generation argument that can be used to monitor generation sizes.

When called without arguments, `collect` behaves as it did before. That is, it collects generations g and younger every r^g times it is called, where r is the value of the parameter `collect-generation-radix`. The target generation into which generations g and younger are collected is $g + 1$, except when g is the maximum nonstatic generation, in which case the target generation is g itself. (Younger generations have lower numbers, with generation 0 being the youngest generation into which objects are placed when they are created.)

When called with one argument, which must be a generation g in the range 0 through m, where m is the maximum nonstatic generation, the oldest generation collected is g. The target generation is g for the maximum nonstatic generation and $g + 1$ otherwise. In previous releases, the oldest generation collected might be greater than g, if the older generation would ordinarily have been collected the next time g is collected.

When called with two arguments, the first, g, determines the oldest generation to be collected, as if only one argument were provided. The second, *gtarget*, determines the target generation. When g is the maximum nonstatic generation, *gtarget* must be g or the symbol `static`. Otherwise, *gtarget* 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, from which objects 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.

The procedure `collect-maximum-generation` is now a parameter. When called without arguments, it returns the current maximum nonstatic generation. This is a change from previous releases, in which `(collect-maximum-generation)` returns a number one greater than the maximum nonstatic generation, i.e., the number assigned to the static generation.

A more significant change is that `collect-maximum-generation` can now also be used to change the number of generations. When called with one argument, g, which must be an exact integer in the range 1 through 254, `collect-maximum-generation` changes the maximum nonstatic generation to g. 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.

Finally, the `bytes-allocated` procedure now takes an optional generation argument, which must be an exact nonnegative integer no grater than the value of `(collect-maximum-generation)` or the symbol `static`. When a generation argument is supplied, `bytes-allocated` reports the number of bytes allocated in that generation, rather than the total number of bytes allocated in all generations. This information might be

used to help guide an adaptive collection strategy based on the sizes of the generations.

2.15. New system querying procedures (7.9.4/7.5)

The procedure `scheme-version-number` returns three exact nonnegative integer values: the major version number, the minor version number, and the sub-minor version number. For example, in *Chez Scheme* Version 7.9.4, `(scheme-version-number)` returns the three values 7, 9, and 4.

The procedure `scheme-version` returns a string that identifies the system (*Petite Chez Scheme* or *Chez Scheme*) and version number.

The procedure `petite?` returns `#t` when called in *Petite Chez Scheme* and `#f` when called in *Chez Scheme*

The procedure `threaded?` returns `#t` when called in a threaded version of *Chez Scheme* and `#f` otherwise.

The procedure `get-process-id` returns the process id of the process. It is the same for all threads running in the same process in threaded versions of *Chez Scheme*.

The procedure `get-thread-id` returns the *Chez Scheme* thread id for the current thread. It is different for each thread and is not related to the process id or any other O/S process or thread id.

2.16. New date/time conversion procedures (7.9.4)

The new procedures `date->time-utc` and `time-utc->date` convert between time objects with type `time-utc` and date objects.

2.17. New `sleep` procedure (7.9.4)

The `sleep` procedure takes a time object with type `time-duration`, which can be created using `make-time`, and causes the invoking thread to suspend operation for at least 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.

2.18. Profile palette change (7.9.4)

Profile palettes must now contain three or more (rather than two or more) entries. The first entry is used for unprofiled code, the second for unexecuted profiled code, and the remainder for executed profiled code. By default, a black background is used for unprofiled code, and a gray background is used for unexecuted profiled code, while 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.

2.19. New gensym printing option (7.9.4)

When the parameter `print-gensym` is 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.

```
(define g1 (gensym))
(define g2 '#{g0 qlw4di6s4woz0sy-0})
g1  ⇒  #{g0 ql8sto9wbe3de7u-0}
g2  ⇒  #{g0 qlw4di6s4woz0sy-0}
(print-gensym 'pretty/suffix)
```

```
g1  ⇒  g0.0
g2  ⇒  g0.qlw4di6s4woz0sy-0
```

This new option is particularly useful as alternative to the `pretty` option for programs that generate many gensyms with the same pretty name, by calling `gensym` with a string argument, e.g., `(gensym "x")`, as illustrated below.

```
(print-gensym 'pretty/suffix)
(map gensym (make-list 5 "x"))  ⇒  (x.0 x.1 x.2 x.3 x.4)
```

2.20. Compile-time Values and Properties (7.9.4)

Two mechanisms are now available for attaching information to identifiers in the compile-time environment: compile-time values and compile-time properties. These mechanisms are useful for communicating information between macros. For example, a record-definition macro might use one of these mechanisms to store information about the record type in the compile-time environment for possible use by record definitions that define subtypes of the record type.

A compile-time value is a kind of transformer, created with the procedure `compile-time-value`, 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. Compile-time properties are defined with `define-property`, which is a definition and can appear wherever any definition can appear.

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.

2.21. New `define-values` syntax (7.9.4)

The new `define-values` form can be used to define multiple variables to the multiple values produced by an expression. `(define-values` *formals expr*`)` is similar to `(define` *var expr*`)` except that the left-hand side is structured as a `lambda` formals list. It evaluates *expr* and binds the variables specified by *formals* to the resulting values.

2.22. `create-exception-state` procedure (7.9.4)

The new procedure `create-exception-state` creates an exception state whose stack of exception handlers is empty except for, in effect, an infinite number of occurrences of a given *handler* at its base. If no *handler* is supplied, a handler that invokes the value of the `base-exception-handler` parameter is used.

2.23. New `environment-mutable?` predicate (7.9.4)

The new procedure `environment-mutable?` accepts one argument, an environment. It returns `#t` if the environment is mutable, otherwise `#f`.

2.24. `with-interrupts-disabled` (7.9.4)

The new syntactic form `with-interrupts-disabled` is identical in syntax and behavior to the existing `critical-section` form. While the `critical-section` is still supported, it may be removed in a future release, because it does not actually provide a critical section when multiple native threads are used in threaded versions of the *Chez Scheme*. Rather, as the new name suggests, the form disables interrupts, which are per-thread, but when used in one thread has no affect on what other threads can do.

2.25. `custom-port-buffer-size` parameter (7.9.4)

A new parameter, `file-buffer-size`, has been added. This parameter controls the sizes of the buffers associated with custom ports. The initial value is currently 128.

2.26. `make-boot-file` (7.9.4)

The new procedure `make-boot-file` is used to combine a set of source and object files into a boot file. The procedure takes an output filename, a list of strings naming base boot files, and zero or more input filenames. It writes to the output file a boot header naming the base boot files (as with `make-boot-header`) followed by the object code for each input file. If an input file is not already compiled, `make-boot-file` compiles the file as it proceeds.

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. If no input files are specified, `make-boot-file` creates a boot header; thus, `make-boot-file` subsumes the functionality of `make-boot-header`.

2.27. Expression editor enabled for scripts (7.9.4)

For scripts that call `new-cafe`, the expression editor is now used unless it has been disabled via the `--eedisable` command-line option. The history file is not read until the first call to the prompt-and-read procedure installed by the expression editor.

2.28. Unicode path names in Windows (7.9.4)

Support for Unicode path names under Windows has been added. This affects file open operations such as `open-file-input-port` or `load`, as well as filesystem operations such as `delete-file`, `file-access-time`, `file-exists?`, and `mkdir`.

2.29. Windows heap search path (7.9.4)

Within a heap-search path under Windows, the two-character sequence `%x` now expands into the executable's directory, and the default heap-search path now consists only of `%x`. The system thus looks for boot files only, by default, in the same directory as the executable. 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 environment variable `SCHEMEHEAPDIRS` can be set to override both the default and the registry setting, if any.

2.30. New `(chezscheme)` and `(chezscheme csv7)` libraries (7.9.4/7.9.3)

The new `(chezscheme)` library is identical to the existing `(scheme)` library, and the new `(chezscheme csv7)` library is identical to the existing `(scheme csv7)` library. The new names can be used to make clear that *Chez Scheme* extensions are involved.

2.31. Procedures for manipulating top-level keyword bindings (7.9.4/7.9.3)

The new procedure `define-top-level-syntax`, which accepts a symbol representing a keyword, a transformer, and an optional environment defaulting to the current interaction environment, can be used to establish a top-level keyword binding in an interactive environment, just as `define-top-level-value` can be used to establish a top-level variable binding. The new procedure `top-level-syntax`, which accepts a symbol and an optional environment, can be used to retrieve the transformer associated with a keyword in an environment. The new procedure `top-level-syntax?`, which accepts a symbol and an optional environment, can be used to ask whether an identifier represented by a symbol is bound as a keyword in an environment.

2.32. Compiling and loading programs and libraries (7.9.4/7.9.3/7.5)

New procedures for compiling and loading programs and libraries have been added. `compile-program` is similar to `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`. `load-program` is similar to `load` but also differs in that it implements the semantics of RNRS top-level programs.

`compile-library` and `load-library` are identical to `compile-file` and `load` except they treat the input file as if prefixed by an implicit `#!r6rs`, disabling any *Chez Scheme* lexical extensions, such as explicit vector lengths. More precisely, *Chez Scheme* lexical extensions can be used within a library compiled or loaded with these procedures only if the `#!chezscheme` marker appears before the first use of those extensions.

2.33. `#!fold-case` and `#!no-fold-case` (7.9.3)

The reader now recognizes two new comment directives, analogous to the existing `#!r6rs` and `#!chezscheme` directives. If `#!fold-case` has been seen (more recently than `#!no-fold-case`) in an input stream, subsequent reads from port are case-insensitive, i.e., symbols and character names are case-folded, as if by `string-foldcase`. If `#!no-fold-case` has been seen (more recently than `#!fold-case`) in an input stream, subsequent reads from the port are case-sensitive. If neither has been seen in an input stream, case-sensitivity is determined as before by the `case-sensitive` parameter, whose value defaults to `#t`. The case sensitive parameter is not consulted if either `#!fold-case` or `#!no-fold-case` has been seen. `#!fold-case`, `#!no-fold-case`, and `case-sensitive` are all ignored when vertical bars or slashes (excluding Unicode hex escapes) are seen. Like `#!r6rs` and `#!chezscheme`, `#!fold-case` and `#!no-fold-case` do not require delimiting, so `#!fold-caseaBc` reads as the symbol `abc`.

2.34. New `--optimize-level` command-line option (7.9.3)

The new `--optimize-level` option sets the initial value of the optimize-level parameter to 0, 1, 2, or 3. The value is 0 by default.

2.35. `file-buffer-size` parameter (7.9.3)

A new parameter, `file-buffer-size`, has been added. This parameter controls the size of a buffer used when a file is opened with anything by buffer mode `none`. The default value of this parameter is currently set at the minimum of 4096 and the value of the underlying operating system's stdio `BUFSIZ` constant.

2.36. `current-exception-state` parameter (7.9.3)

The new `current-exception-state` may be used to save and later restore the state of the exception system.

2.37. `fork-thread` and multiple values (7.9.3)

`fork-thread` now permits its thunk argument to return zero values or more than one value.

2.38. `system` return value (7.9.3)

The `system` procedure, which runs a command in a newly forked process and waits for it to terminate, now raises an exception if the subprocess creation fails, if the exit status of the forked process cannot be determined, or if the forked process is terminated by a signal. Otherwise, it returns the exit status of the child process.

2.39. `abort` optional argument (7.9.3)

The `abort` procedure now takes an optional argument, like `exit`, that may be used to specify the exit code for the Scheme process.

2.40. New inspector commands (7.9.3)

The interactive inspector now supports `reset` (`r`) and `abort` (`a`) commands, which may be used to reset to the current REPL or abort from the system.

For characters and strings, the interactive inspector also supports a new `unicode` command that displays the Unicode scalar values of the character or string elements.

2.41. `inspect/object` extensions (7.9.3)

The continuation and code inspector-object source-path message now returns a position as well as a file name if the file name is known but the (unmodified) file cannot be found. Previously, it returned a single value in this case, the file name.

2.42. New I/O procedures and support for nonblocking I/O (7.9.3)

New input procedures `get-string-some`, `get-string-some!`, and `get-bytevector-some!` have been added. `get-string-some` is like the R6RS `get-bytevector-some` but operates on textual input ports and returns a string rather than a bytevector. `get-string-some!` and `get-bytevector-some!` are like the R6RS `get-string-n!` and `get-bytevector-n!` except they may return less than the requested count.

The new procedures `put-string-some` and `put-bytevector-some` are like `put-string` and `put-bytevector` except they do not guarantee to write the entire string or bytevector, and they return a count of the actual number of elements written.

The procedure `set-port-nonblocking!` may be used to put a port into nonblocking mode. When a port is in nonblocking mode, `get-string-some` may return an empty string, `get-bytevector-some` may return an empty bytevector, and `get-string-some!` and `get-bytevector-some!` may return 0, in each case indicating that no input is available, i.e., an attempt to read would block. Similarly, `put-string-some` and `put-bytevector-some` may return 0, indicating that an attempt write to the port would block.

The procedure `port-nonblocking?` may be used to determine if a port is in nonblocking mode. The predicates `port-has-port-nonblocking??` and `port-has-set-port-nonblocking!?` return `#t` if the nonblocking status of a port may be queried or set and `#f` otherwise. They should be called before attempting to use the `set-port-nonblocking!` or `port-nonblocking?` procedures.

2.43. Expression editor delimiter handling (7.9.3)

The expression editor no longer changes the corresponding close delimiter when an open delimiter is inserted into existing code, since this often led to surprising results.

2.44. `file-directory?` and `file-exists?` change under Windows (7.9.3)

The `file-directory?` and `file-exists?` predicates now return `#t` under Windows for existing drive names like `c:`, mounted volumes like `//server/mount`, and directories specified with (or without) a trailing slash (and regardless of whether forward or backward slashes are used).

2.45. R6RS (7.9.2)

(Initial support was included in Version 7.9.1.)

Version 7.9.4 implements the entire language of the Revised[6] Report on Scheme (R6RS) and associated libraries, including all of the syntactic requirements and each export of the standard libraries, namely:

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

Top-level programs are supported via the `--program` command-line option, i.e., the shell command:

`scheme --program` *pathname*

runs the top-level program contained in the file named by *pathname*. To create an executable R6RS top-level program on Unix-based system, insert:

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

at the front of the top-level program (adjusting the path for `scheme` or replacing `scheme` with `petite` as appropriate) and give the file execute permissions. To make use of existing executable top-level programs containing the "shebang" line

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

recommended by R6RS nonnormative appendix D.1, create a copy or symbolic link of the Scheme executable to `scheme-script` and copies or symbolic links of the "scheme" or "petite" boot file to `scheme-script.boot`. Place the former somewhere in the path searched by `/usr/bin/env` and the latter in a standard directory for *Chez Scheme* boot files, e.g., the same place where `scheme.boot` and/or `petite.boot` already reside. (This may already be done as part of the installation process.)

2.46. Interaction environment and R6RS (7.9.2)

The default interaction environment used for any code that occurs outside of an R6RS top-level program or library (including such code typed at the REPL or loaded from a file) contains all of the bindings of the `(scheme)` library (or `scheme` module, which exports the same set of bindings). This set contains a number of bindings that are not in the R6RS libraries. It also contains a number of bindings that extend the R6RS counterparts in some way and are thus not strictly compatible with the R6RS bindings for the same identifiers. To replace these with bindings strictly compatible with R6RS, 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* R6RS bindings, use the following.

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

The read-eval-print loop (REPL) and files loaded using `load` or included using `include` also support various Chez Scheme lexical extensions, by default. To disable these extensions, `#!r6rs` can be inserted at the front of a file to be loaded or included or before the start of an expression typed into the REPL. The `#!r6rs` mode is implicit for libraries loaded as a result of an `import` form and for R6RS top-level programs. To enable *Chez Scheme* extensions in libraries and R6RS top-level programs, the prefix `#!chezscheme` can be used.

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

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

It might also be useful to include `debug` in the set of identifiers imported from `(scheme)` 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 R6RS 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 R6RS syntax in some way. This should not be a problem except for R6RS 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.

The procedures that are not strictly compatible include the following, which return `#t` for one argument (while the R6RS versions require two or more):

`<`, `<=`, `=`, `>`, `>=`, `char<=?`, `char<?`, `char=?`, `char>=?`, `char>?`, `string<=?`, `string<?`, `string=?`, `string>=?`, `string>?`, `char-ci<=?`, `char-ci<?`, `char-ci=?`, `char-ci>=?`, `char-ci>?`, `string-ci<=?`, `string-ci<?`, `string-ci=?`, `string-ci>=?`, `string-ci>?`;

the following, which are extended to accept zero or more arguments:

`fx*`, `fx+`, `exit`;

the following, which is extended to accept one or more arguments:

`fx-`;

the following, which allow radixes 3, 5–7, 9–15, and 17–36:

`string->number`, `number->string`;

the following, which accepts either one or two arguments and allows the first argument to represent a definition:

`eval`;

the following, for which the argument is optional and defaults to the current output port:

`flush-output-port`;

the following, which are parameters and thus may be used to alter the encapsulated value:

`command-line`, `current-error-port`, `current-input-port`, `current-output-port`, `standard-error-port`, `standard-input-port`, `standard-output-port`;

the following, which accept an "options" symbol or list:

`call-with-input-file`, `call-with-output-file`, `open-input-file`, `open-output-file`, `with-input-from-file`, `with-output-to-file`;

and the following, which accept optional arguments:

`delete-file`, `dynamic-wind`, `file-exists?`, `record?`.

The only keyword that is not strictly compatible is `syntax-rules`, which allows fenders.

For details on the `(scheme)` library versions of these procedures and syntactic forms, see the *Chez Scheme Version 7 User's Guide*.

2.47. Incompatible Changes (7.9.2)

Although we have strived to maintain backward compatibility with earlier versions of Scheme wherever possible, the following incompatible changes have been made to support R6RS.

- The reader now recognizes R6RS hex Unicode escapes in characters, strings, and symbols. In some cases, characters, strings, and symbols containing next Unicode escapes would have parsed as some different character, string, or symbol.

- The setting of `(case-sensitive)` is now `#t` by default, so the case is distinguished in symbols and character names.

- The C function `Sstring_value` has been eliminated because strings are no longer represented as byte strings but rather using an internal representation that supports Unicode. Routines that accepted Scheme strings as arguments and treated them as nul-terminated byte strings should instead be passed Scheme bytevectors instead. Bytevectors are nul-terminated (with the nul byte being stored in the first byte beyond the last element of the bytevector and not counted in the length), and the address of the start of a bytevector's data can be obtained with `Sbytevector_data`, which is analogous to the old `Sstring_value`.

- For any foreign-procedure argument declared as a `string`, a bytevector or a string may be passed. If a bytevector is passed, the foreign procedure receives the address of the first byte of the bytevector. Bytevectors are nul-terminated as described above to facilitate the use of bytevectors as arguments to foreign procedures that expect nul-terminated strings.

  If a string is passed, it is copied to a freshly allocated bytevector (since strings are no longer represented internally as byte arrays) using the equivalent of `string->utf8`. Modifications of the object by the foreign procedure affect the freshly allocated bytevector only and are not reflected back to the original string.

  In most cases, it is preferable to pass bytevectors rather than strings to avoid the copying overhead and so that modifications made by the foreign procedure are visible to the caller. Even when the argument is represented by a string in Scheme and no modifications are made by the foreign procedure, it may still be appropriate for the caller to convert the string to a bytevector explicitly when the implicit UTF-8 conversion is not appropriate.

- R6RS does not treat complex numbers with inexact zero (`+0.0` or `-0.0`) imaginary parts as real numbers, nor does it treat infinities and nans as rational numbers. Thus, `integer?`, `rational?`, and `real?` now return `#f` for complex numbers with inexact zero imaginary parts, while `integer?` and `rational?` now return `#f` for `+inf.0` and `-inf.0`, `rational?` now returns `#f` for `+nan.0`. Correspondingly, procedures that accept only integer, only rational, and only real arguments now reject those that no longer qualify as integer, rational, or real.

- Binary data (bytes) cannot be read from or written to textual ports, and textual data (characters) cannot be read from or written to binary ports. Source (textual) and compiled (binary) code can thus no longer be loaded from the same file. Similarly, (binary) fasl data can no longer be combined with textual data, and fasl data must be read with `fasl-read`, not `read`. The port argument of `fasl-write` must now be a binary output port and is no longer optional, since there is no current binary output port.

  Applications requiring the mixing of textual and binary data should use a binary port and convert the portions of the file representing textual data to and from characters or strings using an appropriate conversion procedure. It is is also possible, when the textual portions are represented using characters in the Latin-1 character set, to open the port as a textual port using a transcoder constructed from the `latin-1` codec with eol-style `none`, then convert the binary portions to and from bytes using `char->integer` and `integer->char`.

- The ordering of *old* and *new* ids in the module `import rename` syntax is now *old* first, *new* second to match the `import` syntax for libraries.

- The `letrec` and `letrec*` forms assume the continuations of right-hand-side expressions are invoked at most once.

- The `(scheme)` library and default interaction-environment bindings for `record-type-descriptor`, `record-type-field-names`, `record-type-name`, and `record-type-mutable?` have their R6RS semantics, which conflicts with the old *Chez Scheme* semantics. Bindings for these identifiers that

are compatible with the old *Chez Scheme* semantics, along with bindings for the related procedures `record-field-accessible?`, `record-field-accessor` and `record-field-mutator`, as well as `record-type-symbol` and `record-type-field-decls`, which have all been superseded by R6RS versions, are available only in the `(scheme csv7)` compatibility library.

- The `error-handler` and `warning-handler` parameters have been eliminated because their semantics are in direct conflict with the R6RS exception handling mechanism.
- The `(scheme)` library and thus interaction-environment binding for `error` is compatible with the R6RS error, which means the second argument is not treated as a format string. When the argument is a format string, use `errorf` instead, which is like `error` except the second argument *is* treated as a format string (and the additional arguments as arguments to be consumed while formatting). For consistency, the same change has been made for `warning`, and the corresponding procedure `warningf` has also been added. Similarly, the new procedure `assertion-violationf` is like the R6RS `assertion-violation` but treats its second argument as a format string.
- The thread-system `condition?` procedure has been renamed `thread-condition?`, since the R6RS `condition?` predicate is used to test for the condition objects that are passed to exceptions.
- As required by R6RS, of the standard data types, only booleans, numbers, characters, strings, and bytevectors count as expressions if they are not quoted. In particular, vectors and the empty list must be quoted. Of the nonstandard data types, fxvectors, `#!eof`, and `#!bwp` need not be quoted. Previously, *Chez Scheme* treated any unquoted value other than a pair or symbol as a literal.
- Also, as required by R6RS, the hash character (`#`) is now a delimiter, so, for example, `abc#def` is no longer a valid symbol. Although some non-r6rs symbols, like `1+`, are still accepted by default on input, i.e., except after `#!r6rs`, they are printed using r6rs syntax, e.g., `\x31;+`. In particular, while `1+` and `1-` are still defined as variables bound to increment and decrement procedures, their names print in a funny way.

2.48. C library routines for 32- and 64-bit integers (7.9.2)

Four new C library operators have been added for converting 32- and 64-bit integers from their Scheme representations:

- `Sinteger32_value(`x`)` returns the 32-bit integer value of the Scheme exact integer x;
- `Sunsigned32_value(`x`)` returns the 32-bit unsigned integer value of the Scheme exact integer x;
- `Sinteger64_value(`x`)` returns the 64-bit integer value of the Scheme exact integer x; and
- `Sunsigned64_value(`x`)` returns the 64-bit unsigned integer value of the Scheme exact integer x.

An exception is raised if a 32-bit value is not in the range -2^{31} through $2^{32}-1$ or a 64-bit values is not in the range -2^{63} through $2^{64}-1$.

Similarly, four new C library operators have been added for converting 32- and 64-bit integers to their Scheme representations:

- `Sinteger32(`x`)`,
- `Sunsigned32(`x`)`,
- `Sinteger64(`x`)`, and
- `Sunsigned64(`x`)`.

Each returns a Scheme exact integer whose value is x. The arguments are signed or unsigned 32- or 64-bit C integers, as appropriate.

2.49. New nonstandard character names (7.9.2)

The characters `#\nel` and `#\ls`, representing the Unicode next-line and line-separator characters, are recognized by the reader (except after `#!r6rs`).

2.50. Fewer "invalid context for definition" errors (7.9.2)

The expander now expands library body forms left-to-right even after discovering an apparent expression so that a spelling error in a definition keyword (e.g., `defnie` for `define`) results in an "unbound identifier" error rather than an "invalid context for definition" error for some subsequent definition.

2.51. Windows library change (7.9.1/7.9.2)

Windows builds of *Chez Scheme* now link against `msvcr90.dll`. Static libraries built using libcmt are also available.

2.52. Saved heaps not currently supported (7.9.1)

None of the operating systems upon which *Chez Scheme* runs provide the guarantees about storage allocation needed to restore a saved heap with absolute addresses, and while restoration used to work on some operating systems even without the guarantee, this is generally no longer the case as the operating systems are randomizing the addresses of data and even code for security purposes. We are looking into ways to address this problem, including relocatable saved heaps, but for the present, support for saved heaps has been removed from the system.

2.53. Thread-safe console ports (7.9.1)

Unbuffered file output ports are no longer thread-safe. The default console output port remains thread-safe, however, and the default console input port is now thread-safe as well. There is a substantial performance cost in both cases, so programs that need faster I/O should create their own ports and either avoid using them from multiple threads or arrange for appropriate synchronization. (This is relevant only for the threaded versions of Chez Scheme.)

2.54. `input-port-ready?` and `char-ready?` (7.9.1)

A new `input-port-ready?` procedure has been that is similar to the old `char-ready?` but works on both binary and textual ports. The `char-ready?` procedure still works for textual ports.

Under Windows, these procedures properly handle files and ports, while in previous releases, `char-ready?` always returned `#t`. In cases where the system cannot determine if input is ready, e.g., for a console port that has no data already buffered, each of these procedures raises an exception so that the application can control what happens in such cases. Thus, these procedures either (a) return `#f` if no input is ready and the port is not at end of file, (b) return `#t` if input is ready or the port is at end of file, or (c) raise an exception if the ready status cannot be determined.

2.55. X86_64 Support (7.9.1)

Support for running *Chez Scheme* with 64-bit pointers on the x86_64 architecture under Linux, MacOS X, and OpenBSD with machine types a6le, a6osx, and a6ob for nonthreaded and ta6le, ta6osx, and ta6ob for threaded, has been added. C code intended to be linked with these versions of the system should be compiled using the Gnu C compiler's `-m64` option, which may or may not be the default.

2.56. Additional byte-vector operations (7.9.1)

In addition to the standard R6RS bytevector operations, *Chez Scheme* also supports the following procedures:

`(bytevector->s8-list` *bytevector*`)` ⇒ *list*
`(s8-list->bytevector` *s8-list*`)` ⇒ *bytevector*
`(bytevector` *fill* `...)` ⇒ *bytevector*

where each element of *s8-list* is an exact integer value ranging from −128 through +128 and each *fill* is an exact integer value ranging from −128 through +255.

As with vectors and bytevectors, the reader allows an optional length to appear in the bytevector prefix, e.g.:

```
(bytevector? #10vu8(0))
```

is a bytevector containing 10 zero bytes. (This cannot be used following `#!r6rs`, which limits the syntax recognized by the reader to R6RS-only features.)

The `read-token` procedure returns token types `vu8paren` and `vu8nparen` for bytevector prefixes, analogous to the `vparen` and `vnparen` token types for vector prefixes.

The C library interface supports four new bytevector operators:

- `Smake_bytevector(`*len*`,`*n*`)`, which returns a new bytevector of length *len* with each element set to *n*;
- `Sbytevector_u8_ref(`*bv*`,`*i*`)`, which returns the *i*th byte of *bv*;
- `Sbytevector_u8_set(`*bv*`,`*i*`,`*n*`)`, which sets the *i*th byte of *bv* to *n*; and
- `Sbytevector_data(`*bv*`)`, which returns a pointer to *bv*'s data.

2.57. `meta-cond` generalization (7.9.1)

The `meta-cond` syntax has been extended to allow it to be used in definition contexts. `meta-cond` still expands into `(void)` if no clause's test evaluates to true and no `else` clause is present, however, which is not suitable in a definition context. To avoid this problem, use an explicit `else` clause and expand into `(begin)`.

2.58. Default heap/boot search path (7.9.1)

The default heap and boot search path on Unix-based systems now starts with `~/lib/csv%v/%m` if the home directory can be determined. The default heap and boot search path is still determined by a registry setting under Windows.

2.59. Tilde-prefixed paths under Windows (7.9.1)

The filename prefix `~/` now expands to the user's home directory under Windows, as on Unix-based systems, if the `HOMEDRIVE` and `HOMEPATH` environment variables are set.

2.60. Months range from 1 to 12 (7.5)

The month field of a date record, e.g., one returned by `current-date`, runs from 1 through 12 rather than 0 through 11, for compatibility with SRFI 19.

2.61. Monotonic time (7.5)

Requesting monotonic no longer fails on Linux systems that do not support monotonic clocks; on such systems, the real-time clock is now used instead. All of the high-precision time procedures now fall back on older time mechanisms when requesting high-precision timers fails.

2.62. New and altered expression-editor key bindings (7.5)

The expression editor now binds the common delete-key sequence (`Esc[3~`) to `ee-delete-char` and escape followed by the same sequence to `ee-delete-sexp`. The effect of this is that the delete key should now delete forward rather than backward. The backspace key should still delete backward.

New bindings have also been added for the home and end keys of certain terminal emulators, including the Gnome terminal.

2.63. Top-level value handling (7.5)

The `top-level-value` and `top-level-bound?` procedures now recognize as bound a variable assigned by a file compiled in one session and loaded into another session—even if the variable was not defined.

3. Bug Fixes

3.1. Boot file use of `scheme-version` (7.9.4)

The procedure `scheme-version` now works properly when called during the loading of a boot file.

3.2. Work-around for automatic margins on MacOS X Terminal (7.9.3)

The expression editor disables automatic margins (i.e., automatic line wrapping) when possible so it can use the last column without scrolling the display. Unfortunately, when automatic margins are disabled in the MacOS X Terminal application, they are disabled simultaneously for all windows, not just for the current window, causing problems when the expression editor is running in one window and long lines are printed in another. The expression editor now disables automatic margins only temporarily while writing text to the last column, reducing (though not entirely eliminating) the possibility that other windows are affected.

3.3. Invalid memory reference using hashtables (7.9.3)

A bug that caused corruption of the heap and typically resulted in an "unrecoverable invalid memory reference" while using the R6RS eq hashtable interface has been fixed.

3.4. Bug in `format` tabulation (7.9.3)

A bug resulting in an "undefined for zero" error when the optional column width specifier in a tabulate directive is zero, e.g., `~1,0t`, has been fixed.

3.5. Expansion internal error (7.9.3)

A bug in the expander that sometimes resulted in an internal error ("source-wrap ae/x mismatch") has been fixed.

3.6. Expression editor datum comment handling (7.5)

The expression editor no longer returns end-of-file for an entry containing only a single commented-out datum.

3.7. Expression editor clipboard bug under Windows (7.5)

The expression editor now properly closes the clipboard after a paste (control-v) operation, i.e., no longer prevents new items from being copied to the clipboard.

3.8. Bug in `ash` (7.5)

A bug that caused `ash` to return the wrong result for word-sized or greater negative (right) shifts of large powers of two has been fixed.

3.9. Bug in `inexact` (7.5)

A bug that could cause `inexact` (aka `exact->inexact`) to improperly round an exact value halfway between two floating-point values has been fixed.

3.10. Bug in handling of `compile` (7.5)

A bug in the compiler that caused it to treat the return value of `compile` as true in test context, no matter what it actually returns, has been fixed.

4. Performance Enhancements

4.1. Reduced parameter overhead (7.9.4)

References to locally defined parameters, including thread parameters, are now inlined, thus reducing the cost of such references.

4.2. Reduced predicate overhead for sealed record types (7.9.4)

The code generated by `record-predicate` for sealed record types is now more efficient than the more general code produced for non-sealed record types.

4.3. Optimized procedural record interface (7.9.1/7.9.4)

Record constructors, predicates, accessors, and mutators created with the procedural record interface are often faster than before because the compiler now tracks record-type information through to the sites where constructors, predicates, accessors, and mutators are created in order to generate specific and inlinable code for them. The effects of this optimization can be seen with `expand/optimize`. (This optimization is not performed by *Petite Chez Scheme*.)

Initial support for this optimization was implemented in Version 7.9.1, with additional improvements in Version 7.9.4.

4.4. Reduced `letrec*` undefined-variable checking overhead (7.9.3)

An improvement in the handling of `letrec*`, which also affects internal definitions, including library and top-level program definitions, causes the compiler to produce fewer checks for undefined variables. Improvements in safe code can be substantial (up to 15% or so), but the amount of improvement is highly dependent on the structure of the code. Unsafe code (code compiled at optimize-level 3) is not affected, since these checks are not performed in unsafe code.

4.5. Improved register assignment for x86 (7.9.1)

A better choice of register assignments has resulted in measurable improvement in x86 performance. Actual improvements differ from one program to another, and we've seen anything from -9% decreases in speed to 36% increases, with increases of 5-15% appearing typical.

4.6. Improved recursive bindings (7.9.1)

The compiler now generates better code for some `letrec` expressions, `letrec*` expressions, and bodies containing internal definitions by taking advantage of the R6RS restriction that right-hand-side expressions should not return to their continuations multiple times.