Chez Scheme Version 6 Release Notes

October 1998

Overview

This document outlines the changes made to *Chez Scheme* for Version 6 since Version 5.

Version 6 is available for the following platforms:

- DEC Alpha AXP Digital Unix 3.X, 4.X
- HP PA-RISC HPUX 10.x
- Intel 80x86 Linux 2.0.X
- Intel 80x86 Windows 95/NT
- Motorola PowerPC AIX 4.X
- Sun Sparc Solaris 2.X
- Silicon Graphics IRIX 6.X

This document contains three sections describing significant (1) major functionality changes, (2) performance enhancements, and (3) bugs fixed. A version number listed in parentheses in the header for a change indicates the first minor release or prerelease to support the change.

1. Major Functionality Changes Since Version 5

1.1. *Petite Chez Scheme*

A freely distributable version of *Chez Scheme*, *Petite Chez Scheme*, is included with *Chez Scheme* Version 6. *Petite Chez Scheme* may be used as a run-time environment for compiled *Chez Scheme* applications or as a stand-alone Scheme system. With the exception that the compiler is not present, *Petite Chez Scheme* is fully compatible with *Chez Scheme*. The interpreter, normally available via the `interpret` procedure, replaces the compiler as the default evaluator.

Lacking the compiler limits *Petite Chez Scheme* in a few specific ways. Most importantly, interpreted code is generally not nearly as fast as compiled code, so source code loaded into *Petite Chez Scheme* is not as fast as in *Chez Scheme*. Compiled code loaded into *Petite Chez Scheme* runs at the same speed as in *Chez Scheme*, however, unless it explicitly invokes the evaluator. `foreign-procedure` expressions must be compiled, so *Petite Chez Scheme* cannot process these expressions. Compiled `foreign-procedure` forms loaded into *Petite Chez Scheme* pose no difficulty, however. Finally, inspector information is not generated for interpreted code, so the inspector is not as useful in *Petite Chez Scheme* as it is in *Chez Scheme*.

Applications developed using *Chez Scheme* may be distributed in source or object code form along with *Petite Chez Scheme*, which serves as the run-time environment for the application. Applications that require the compiler cannot be distributed without special licensing arrangements. It is often useful to distribute an installation script along with an application that produces a heap file on the local machine and to redefine the parameter `scheme-start` to start the application automatically. See the *Chez Scheme User's Guide* for details.

1.2. New *Chez Scheme User's Guide*

Complete documentation for all features specific to *Chez Scheme* Version 6 can be found in the new *Chez Scheme User's Guide*. This book complements *The Scheme Programming Language*, second edition, which documents ANSI/IEEE Scheme and extensions described in the Revised^4 and Revised^5 Reports on Scheme.

1.3. Online Documentation

The *Chez Scheme User's Guide* and *The Scheme Programming Language*, second edition are now distributed electronically with *Chez Scheme* as html documents. The *Chez Scheme User's Guide* includes a unified index and a unified summary of forms that contain entries for both documents, effectively linking the two documents into a single document that covers the entire language supported by *Chez Scheme*. The online index and summary of forms also contain page numbers for the printed versions of the books and can thus be used as searchable indexes for the printed versions. The documentation is also available via a link on the *Chez Scheme* web site, *www.scheme.com*.

1.4. Support for Intel Windows NT and Windows 95 (5.9a, 5.9i, 6.0)

Support for Intel-based systems running Windows NT and Windows 95 has been added. *Chez Scheme* runs under Windows as a console-mode application, *i.e.*, no direct windowing support is provided. Because access to C library routines is supported via the foreign procedure interface, however, it is possible to interact with windowing code written in other languages, including the Windows API. The Scheme Widget Library (SWL) windowing system, distributed by Indiana University and included with *Chez Scheme* as a contributed program, is one such example, linking *Chez Scheme* to the Tcl/Tk graphics and windowing toolkit. Dynamic loading of dynamic link libraries (DLL) is supported via `load-shared-object`.

Under Windows NT, keyboard interrupts work as follows. When used to interrupt a running computation, a single keyboard interrupt (initiated with Control-C or Control-Break) causes the keyboard interrupt handler to be invoked. The default keyboard interrupt handler enters the debugger. A second keyboard interrupt may be used if the system does not respond and will terminate the Scheme process. A keyboard interrupt may be initiated to cancel input from the console, which by default causes a reset to the current café.

Under Windows 95, keyboard interrupts work similarly, except that when using a keyboard interrupt to cancel input, it is necessary to press the enter key after the interrupt key. A second keyboard interrupt will terminate the Scheme process.

No other signal support is provided under Windows; in particular, `register-signal-handler` is unsupported.

As of Version 6.0, the `process` implementation is more robust. As of Version 5.9i, the virtual memory limit has been increased from the original miserly 32MB limit to a maximum of 512MB depending upon operating system constraints such as available physical memory and swap space. Also as of Version 5.9i, the `cpu-time` primitive is now more accurate under Windows NT; under Windows 95 `cpu-time` returns elapsed real (clock) time, i.e., the same value as `real-time`. The Windows version presently lacks support for `char-ready?`, which always returns `#f`.

1.5. Datum Comments

The reader now allows an entire datum to be commented out with a single `#;` prefix, without regard to line boundaries. For example, the expression

```
(let ()
  (define foo
    (lambda (x #;y z) ; no more y
      #;(pretty-print y)
      (+ x z)))
  #;(define bar
      (lambda (x)
        #;(pretty-print x) x)) ; look ma, nested #;
  (let ([a (foo 3 #;4 5)]
        #;[b (foo 6 7 8)]) ; no more b
    (+ (* a a) #;(* b b))))
```

is equivalent to

```
(let ()
  (define foo
    (lambda (x z)
      (+ x z)))
  (let ([a (foo 3 5)])
    (+ (* a a))))
```

and evaluates to 64. This mechanism is probably most useful for commenting out local definitions, as with the definition of `bar` in the example above.

1.6. R^5RS `eval`

The procedure `eval` now accepts an optional *env-spec* argument. This argument may be one of four values: the value returned by `interaction-environment`, which provides access to the identifier bindings in the normal top-level environment, the value returned by `scheme-report-environment` (when applied to `5`), which provides read-only access to the required and optional identifier bindings described in the Revised[5] Report, the value returned by `null-environment` (when applied to `5`), which provides read-only access to the required and optional keyword bindings described in the Revised[5] Report, and the value returned by `ieee-environment`, which provides read-only access to the identifier bindings described in ANSI/IEEE Standard for Scheme. *env-spec* defaults to the value returned by `interaction-environment`.

Two-argument `eval` is required by the Revised[5] Report on Scheme, as are `interaction-environment`, `scheme-report-environment`, and `null-environment`. One-argument `eval` and `ieee-environment` are *Chez Scheme* extensions. `compile`, `interpret`, `expand`, and `sc-expand` now accept the optional *env-spec* argument and support the same values for *env-spec* as `eval`. `eps-expand` also accepts an optional *env-spec* argument, although it supports only the value returned by `interaction-environment`.

With the addition of the *env-spec* argument to `eval`, *Chez Scheme* now supports all required and optional features of the Revised[5] Report on Scheme.

With this change to `eval`, the old `ieee` and `r4rs` subset modes are no longer necessary and have been removed. The `subset-mode` parameter may now be set only to `system`, which is used primarily in patch files, and `#f`, which is the default (user) mode.

1.7. Source-file information

Source-file information is now associated with compiled code loaded either from a source file or from a compiled file. This allows the compiler to produce better error messages for syntax errors detected by the reader, expander, or compiler, and it allows the inspector to open for listing the file that contains a particular procedure or procedure call found in an inspected continuation. The `source-directories` parameter may be set to define a search path for source files. Because the system records the length and a checksum with each source file descriptor, the system bypasses source files that have the same name as the original file but do not have the same contents. (This includes files that have been modified since they were loaded or compiled.)

1.8. Modules

Chez Scheme Version 6 supports three new definition forms used to define modules and import bindings from them: `module`, `import`, and `import-only`. These forms may appear wherever any other definition may appear, including at top level, within a `lambda`, `let`, `let*`, or `letrec` body, or within another `module` form. Modules may be named or anonymous. A named module encapsulates a set of bindings, some of which may be explicitly exported from the module. None of the bindings are not automatically visible where the module form appears, but may be made visible anywhere within the scope of the module name via `import` or `import-only`. An anonymous modules behaves like a named module, except that the exports are made visible immediately where the module form appears, as with an implicit `import` form.

Modules may be separately compiled; a compiled file containing a module form may be loaded or visited via `load` or `visit`. `load` loads the module and executes its initialization code; `visit` simply makes the module's interface visible to allow compilation of code that imports from the module.

Four modules are predefined by the system: `\#system`, `scheme`, `ieee`, `r5rs`, and `r5rs-syntax`.

A new `literal-identifier=?` predicate has been added and should be used to compare literal identifiers such as `else` in `cond` in place of `free-identifier=?`. `literal-identifier=?` treats two identifiers the same even if they are bound in different top-level modules. `free-identifier=?` must distinguish such identifiers, which is not usually desirable when the identifiers are being treated as literals.

See the *Chez Scheme User's Guide* for details on module and import forms, visiting files, built-in modules, and `literal-identifier=?`.

1.9. Introduced bindings are now hidden at top level

Top-level definitions of identifiers introduced into the output of a macro are now visible only to other code produced by the macro. This brings the top-level treatment of introduced bindings in line with the treatment of local bindings. For example, if a use of the `counter` form defined below appears either at top level or within a local scope, `local` will not be visible except within the code for the introduced definition of `x`.

```
(define-syntax counter
  (syntax-rules ()
    [(_ x)
     (begin
       (define local 0)
       (define x
         (lambda ()
           (set! local (+ local 1))
           local)))]))
```

1.10. Support for edge-count profiling

Support for edge-count profiling has been added to the compiler. When the parameter `compile-profile` is set to true, the compiler instruments the code it generates to perform low-level counting of basic blocks, thereby computing accurate measures of how often each part of the compiled program is executed. This information may be displayed in terms of the original source code by means of a graphical user interface provided with the Scheme Widget Library (SWL) or via a stand-alone profile-viewing program.

1.11. Uninterned symbols may now have property lists

1.12. Changes to `file-position`

`file-position` now returns the most negative fixnum when the position cannot be determined. Existing generic ports should be updated to follow this protocol. `file-position` now fully supports strings ports.

1.13. Identifier macros (5.0b, 6.0)

The `syntax-case` macro system has been extended to permit the definition of *identifier macros.* If a syntactic keyword is found during expansion in a form context other than one in which it is the first element of a structured form, the associated transformer receives just the keyword itself as input. It can detect this situation via the `identifier?` predicate and expand the identifier itself into an arbitrary expression. For example, the definition for `v0` below expands references to `v0` into references to the first element of vector v, whether v0 appears as the first element of a structured form or not:

```
> (let ([v (vector (lambda (x) (+ x 1)))])
    (define-syntax v0
      (lambda (x)
        (syntax-case x ()
          [_ (identifier? x) (syntax (vector-ref v 0))]
          [(_ e ...) (syntax ((vector-ref v 0) e ...))])))
    (cons (v0 1) (map v0 '(2 3 4))))
```

```
(2 3 4 5)
```

Whenever an identifier should expand the same whether it is the first element of a structured form or not, as above, the shorthand syntactic form `identifier-syntax` may be used. For example, the expression above can be rewritten as:

```
> (let ([v (vector (lambda (x) (+ x 1)))])
    (define-syntax v0
      (identifier-syntax (vector-ref v 0)))
    (cons (v0 1) (map v0 '(2 3 4))))
(2 3 4 5)
```

Identifier macros are useful for such purposes as expanding instance variable references into structure accesses in the implementation of object-oriented programming systems and for supporting noncall-position references to macros serving as integrable procedures.

As of Version 6.0, identifier macros may be used to redefine the semantics of assignments as well as references to identifiers. This change required an extension of the syntax of the `identifier-syntax` expression, although the older syntax is still supported. The new syntax separates the treatment of references and assignments into two separate clauses. For example,

```
(let ([x (list 0)])
  (define-syntax a
    (identifier-syntax
      [id (car x)]
      [(set! id e) (set-car! x e)]))
  (let ([before a])
    (set! a 1)
    (list before a x)))
```

evaluates to `(0 1 (1))`. In this example, the `identifier-syntax` expression creates a transformer that converts references to `a` into calls to `car` on `x` and assignments to `a` into calls to `set-car!` on `x`.

See the *Chez Scheme User's Guide* for details on both forms of `identifier-syntax`.

1.14. Improved inspector support for pairs and symbols

The inspector prints pairs with non-list cdrs more intuitively and no longer suppresses the the last cdr of an improper list. It also supports `value` (`v`), `property-list` (`pl`), and `name` (`n`) messages as equivalent to the `r1`, `r2`, and `r3` messages previously supported for inspecting these fields.

1.15. `compile-port`

A new primitive, `compile-port`, has been added. `compile-port` is like `compile-file`, except that it compiles code read from a specified input port to a specified output port, rather than from a specified file into another file. Both port arguments are required, with the input port first and the output port second. Like `compile-file`, `compile-port` takes an optional machine-type argument as its third parameter.

1.16. Support for record datatypes (5.9i)

Support for record datatypes has been added. Records datatypes should most often be created via the new syntactic form `define-record`. A "point" record with two fields, `x` and `y`, may be defined as follows:

```
> (define-record point (x y))
```

This definition results in the definition of several procedures: a constructor named `make-point`, a predicate named `point?`, two accessors named `point-x` and `point-y`, and two setters named `set-point-x!` and and `set-point-y!`.

```
> (define p (make-point 3 2))
> (point? p)
#t
> (vector? p)
#f
> (point-x p)
3
> (set-point-x! p 4)
> (point-x p)
4
```

The resulting datatype is distinct from all other datatypes. This is in contrast to structures defined with `define-structure`, which are represented as vectors.

The record datatype facility is described in the *Chez Scheme User's Guide*, including a procedural interface, facilities for reading, printing, and inspecting records, and mechanisms for declaring field types and mutability of fields.

1.17. Process now "dups" stdout to stderr (5.9i)

The `process` primitive now redirects `stderr` to the `stdout` for child processes it creates. Stderr had been undefined.

1.18. Creating ports from open file descriptors (5.9i)

When the handler argument to `make-input-port`, `make-output-port`, and `make-input/output-port` is a nonnegative fixnum, the resulting port is given the default handler, and the fixnum, which is assumed to be a valid open file descriptor, is recorded within the port for use by subsequent I/O operations. In the case of `make-input/output-port`, the same file descriptor must be usable for both reading and writing; there is no provision for separate file descriptors.

1.19. Example socket code (5.9i)

A new example program, `socket.ss`, has been included with the *Chez Scheme* distribution. This program demonstrates the use of the foreign interface and generic ports to create a convenient interface for communicating with other processes via sockets.

1.20. `trace-output-port` parameter (5.9i)

A new parameter, `trace-output-port`, has been added. Output produced by traced procedures (established by `trace`, `trace-lambda`, `trace-let`, etc.) is sent to the value of this parameter, which must be an output port. (An undocumented parameter, `trace-output`, that performed the same function has been eliminated.)

1.21. Eval argument to `transcript-cafe` (5.9i)

The procedure `transcript-cafe` now accepts an optional *eval* argument analogous to the optional *eval* argument to `new-cafe`.

1.22. Persistent cafe reset and exit handlers (5.9i)

`new-cafe` now sets up reset and exit handlers in such a way that changes to the parameters `reset-handler` and `exit-handler` persist across resets within the cafe.

1.23. Support for the MIPS n32 ABI under IRIX (5.9g)

Support for the n32 ABI under IRIX as a new *Chez Scheme* machine type ("n32sgi") has been added. Support for the older (o32) ABI is still available as "xgi", but may be phased out in later releases. Since the operating system does not permit code from the two ABIs to be intermixed, Scheme object code created for one machine type cannot be used in the other, and any C object code linked with *Chez Scheme* or loaded dynamically via `load-shared-object` must be compiled with the -o32 (default) or -n32 option, as appropriate.

1.24. Support for C manipulation of Scheme objects (5.9d)

A variety of C preprocessor macros and external entry points have been added to allow C programs to examine, allocate, and alter Scheme objects. They also permit C programs to call Scheme procedures (see the next item). C code that uses these features must include the `"scheme.h"` header file distributed with *Chez Scheme* and be linked with the *Chez Scheme* executable. These features are documented in the *Chez Scheme User's Guide*.

1.25. Support for C calls into Scheme (5.9d)

Support for calling Scheme procedures from C has been added. C code may now call Scheme procedures of zero through three arguments via the `Scall0`, `Scall1`, `Scall2`, and `Scall3` routines in the `"scheme.h"` header file mentioned above. A more general interface for longer argument lists is also provided. Continuations may be used to perform nonlocal exits beyond pending C calls as well as pending Scheme calls. These features are documented in the *Chez Scheme User's Guide*.

The example file `"foreign.ss"` distributed with Version 6 contains prototype code, with examples, for converting "foreign-callable" declarations into C interface routines to support C calls to Scheme procedures with automatic datatype conversion analogous to that provided for Scheme calls to C procedures via `foreign-procedure`.

1.26. Support for locking objects (5.9d)

Objects may now be locked to prevent the storage manager from reclaiming or relocating the objects. Objects are locked via the Scheme procedure `lock-object`, which is also available to C code using the C interface described above as `Slock_object`. `unlock-object` or `Sunlock_object` (from C) may be used to unlock the object, allowing it to be reclaimed or relocated.

1.27. Range checks for foreign return values (5.9d)

Range checks for foreign procedures returning `fixnum` or `char` values have been eliminated. These checks added both code and time overhead and were of questionable utility. The upper 24 bits of `char` return values are now ignored. Returning a number outside of the fixnum range from C with a `fixnum` result type produces an unspecified value.

1.28. New primitive for obtaining environment settings (5.9c)

The new primitive `getenv` accepts a single string argument and returns the operating system shell's environment value associated with the argument, or `#f` if no environment value is associated with the argument.

1.29. Support for box syntax in `syntax-case` expander (5.9c)

The `syntax-case` expander `sc-expand` now supports the box syntax in `syntax-case` and `syntax-rules` patterns and templates and within `quasiquote` forms. For example, `` `#&(,(+ 3 4)) `` now evaluates to `#&(7)`.

1.30. Support for PowerPC AIX 4.X (5.9a)

Support for PowerPC-based systems running AIX 4.X has been added. Dynamic loading of foreign (C) object code is supported via `load-shared-object`. Because `load-shared-object` is implemented via the C library `load()` function, object code loaded dynamically must follow the requirements set forth for programs loaded via `load()`.

Libraries such as the C library cannot be loaded directly via `load-shared-object` under AIX. It is possible to make library entries visible to Scheme, however, by registering them explicitly via `register_symbol`, e.g.:

```
register_symbol("strcmp", (int)strcmp);
```

In order for `register_symbol` to be made visible to a program loaded into *Chez Scheme*, `register_symbol` must be included in the import file given to the linker when the program is linked.

1.31. Compatibility features removed from base system (5.9a)

Various library procedures and syntactic forms supported by Version 5 for compatibility with older versions of *Chez Scheme* and with other Scheme and Lisp systems have been removed from the base system. Most are available as user-level definitions in the the file `"examples/compat.ss"` in the release directory. The procedures `support-sicp` and `support-first-class-environments` have been dropped, since they were included only to support the now obsolete first edition of Abelson and Sussman's *Structure and Interpretation of Computer Programs.*

1.32. `*scheme*` is now a parameter, `scheme-start` (5.0d)

The variable `*scheme*` has been converted into the parameter `scheme-start`. The procedure value of this parameter is applied on system start-up from a saved heap to a list of the file names from the command line. Defining or assigning `*scheme*` no longer has any effect on system start-up.

1.33. New command-line options (5.0d)

Support for two new command-line options, `-c` and `--`, has been added.

Normally, the heap is compacted whenever a heap file is saved. This can consume significant time and additional memory for very large heaps. The `-c` option can be used to disable this compaction. Multiple `-c` options toggle heap compaction; it is thus possible to use a shell script that disables compaction by default while still allowing compaction if desired. The `-c` option has no effect if the `-s` option has not been specified.

The `--` option forces all remaining arguments on the command line to lose any special significance. By default, all remaining arguments are treated as file names. This feature may be used in connection with the `scheme-start` parameter to allow applications to give arbitrary interpretation to command line arguments that follow the initial `--`.

1.34. New reader syntax for `syntax` (5.0d)

Just as `(quote` *obj*`)` may be abbreviated `'`*obj*, `(syntax` *template*`)` may now be abbreviated `#'`*template*,

1.35. `fluid-let-syntax` (5.0d)

Support for `fluid-let-syntax` has been added. This feature is described in the second edition of *The Scheme Programming Language.*

1.36. `extend-syntax` with the `syntax-case` expander (5.0d)

`extend-syntax` is now supported by `sc-expand`. In Version 5, `extend-syntax` was supported only by `eps-expand`. This allows both lexical (`syntax-rules` and `syntax-case`) syntactic extensions to coexist with `extend-syntax`, but the interaction between the two is not entirely reliable. In general, it is best to use only one or the other kind of syntactic extension in a single application. This feature should be used only as an aide in migrating applications from `extend-syntax` to `syntax-case`/`syntax-rules`, and will not necessarily be directly supported in future releases.

1.37. Support for Linux ELF (5.0d)

Support for Intel-based Linux ELF systems has been added. Dynamic loading of foreign (C) object code is supported on Linux ELF systems via `load-shared-object`.

1.38. One-shot continuations (5.0d)

Support for *one-shot* continuations has been added. A one-shot continuation is like a normal (multi-shot) continuation, except that it can be invoked at most once, implicitly or explicitly. One-shot continuations may be more efficient than multi-shot continuations when one-shot continuations suffice. One-shot continuations are obtained via `call/1cc`, just as as multi-shot continuations are obtained via `call/cc`.

1.39. Signal handling (5.0c)

A mechanism for handling low-level signals has been added. The new primitive `register-signal-handler` is used to establish a signal handler for a given signal. It expects two arguments: an integer signal number and a procedure of one argument. See your host system's `<signal.h>` or documentation for a list of signals and their numbers. After a signal handler for a given signal has been registered, the handler is called whenever the signal is delivered to the process. The handler should accept one argument: it will be passed the signal number. This allows the same handler to be used for different signals and to be able to differentiate among them.

It is generally not a good idea 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 signal is delivered to the Scheme handler.

1.40. Support for HP PA-RISC 700 Series under HPUX A.09.05 (5.0c)

Support for HP PA-RISC 700 Series systems under HPUX A.09.05 has been added. Saved heaps are not currently memory mapped on this platform due to difficulties in making this work under HPUX. Memory mapping improves startup time and memory utilization. Functionality is not affected. Dynamic loading of foreign (C) object code is supported via `load-shared-object`.

1.41. New broken-weak-pointer object (5.0b)

A new "broken-weak-pointer object", `#!bwp`, has been added. In Version 5, when an object to which the car of a weak pair points is collected, the car of the weak pair is set to `#f`. In Version 6, the car is set to `#!bwp`. The new predicate `bwp-object?` should be used to determine whether an object is the broken-weak-pointer object; it returns true only for `#!bwp`.

```
> (define x (weak-cons (string #\h #\i) '()))
> x
("hi")
> (collect)
> x
(#!bwp)
> (bwp-object? (car x))
#t
```

Code that uses weak pairs must be altered to reflect this change.

1.42. Change to `let-syntax` and `letrec-syntax` (5.0b)

In Version 5, `let-syntax` and `letrec-syntax` behave similarly to `let` in that the forms in the body of a `let-syntax` or `letrec-syntax` form are treated as a `lambda` body, forming a new lexical contour. `let-syntax` and `letrec-syntax` forms now behave more like `begin` forms, so that they can expand into one or more definitions when they appear in contexts where definitions are allowed.

1.43. Inspector eval message (5.0b)

Procedure and continuation objects now accept an `eval` message. If `x` is bound to a procedure or continuation object, `(x 'eval 'expr)` evaluates `expr` with bindings for the free or frame variables bound as for the eval command to the interactive inspector.

1.44. `trace-print` parameter (5.0b)

A new parameter, `trace-print`, has been added. The value of `(trace-print)` is used by the trace package to print both inputs to and outputs from procedures. Its default value is `pretty-print`. The value of this parameter should be a procedure that accepts two arguments, the object to print and a port to which the object should be printed.

1.45. Character name syntax (5.0b)

A new mechanism has been established for extending or changing the set of character names recognized by the reader and printer. The procedure `char-name` is used to associate symbolic names with characters and to look up names associated with characters or characters associated with names.

`char-name` accepts either one or two arguments. When passed one argument, either a symbol or a character, `char-name` returns the associated item. For example, with the default set of character names, `(char-name #\space)` returns `space` and `(char-name 'space)` returns `#\space`. If no association as been made for a symbol or character, `char-name` returns `#f`, so for example, `(char-name #\a)` and `(char-name 'foo)` initially return `#f`.

When passed two arguments, the first must be a symbol *s* whose name consists of two or more alphabetic characters. The second must be a character or `#f`. If the second argument is a character *c*, the name *s* is associated with the character *c*. Any other association for *s* is dropped, while other associations for *c* are retained. Thus, a name can map to only one character, but more than one name can map to the same character. If the second argument to `char-name` is `#f`, any association for *s* is dropped, and no new association is established.

The reader (`read`) and printer (`write` and `pretty-print`) use `char-name` to look up characters associated with names and names associated with characters.

When passed one character argument *c*, `char-name` returns the name associated with *c* for which the association was most recently established.

```
> (char-name 'etx)
#f
> (char-name 'etx #\003)
> (char-name 'etx)
#\etx
> (char-name #\003)
etx
> #\etx
#\etx
> (eq? #\etx #\003)
#t
> (char-name 'etx #\space)
> (char-name #\003)
#f
> (char-name 'etx)
#\etx
> #\space
#\etx
> (char-name 'etx #f)
> #\etx
Error in read: invalid character name #\etx.
> #\space
#\space
```

1.46. Support for new Sparc systems (5.0a)

Support has been added for Sparc systems with separate instruction and data caches, including the Sparc-5. Prior releases of *Chez Scheme* on these systems exhibit sporadic instruction faults and other bugs due to a lack of synchronization between the instruction and data caches.

2. Performance Enhancements Since Version 5

2.1. Improved interpreter performance (5.9i)

By default, code loaded from a file or entered into the read-eval-print loop is compiled directly to machine code by *Chez Scheme*'s incremental compiler. An interpreter has long been available as an alternative to the incremental compiler, although its existence has not been well publicized. It is the default evaluator in *Petite Chez Scheme*, and may be used in *Chez Scheme* by invoking `interpret` on an expression, by setting `current-eval` to `interpret`, or by passing `interpret` as the second argument to `load`. The interpreter is now considerably faster and allocates less intermediate storage than the old interpreter. In addition, the source optimizations described elsewhere in these notes are performed prior to interpretation as well as by the compiler.

2.2. Improved memory compaction for code objects (5.9i)

Code objects are now packed more efficiently whenever a heap is saved, which decreases heap size and increases instruction cache locality. For some programs, this results in a substantial (5% or more) increase in performance.

2.3. Improved letrec optimization (5.9i)

The compiler now handles `letrec` expressions with `let` or `letrec` expressions nested within the "right-hand side" expressions more efficiently. For programs that make heavy use of `letrec` expressions nested in this fashion, this optimization can make a sizable difference in program performance, as it facilitates both procedure inlining and optimization of direct calls between procedures.

2.4. Procedure inlining (5.9a, 5.9i)

The compiler now performs aggressive inlining of procedures along with various other source optimizations, using a strategy that rarely causes much code expansion. The performance increase varies from program to program, but generally seems to be around 8–10% on average, with much more impressive speed-ups for some programs. Inlining may be controlled via a set of new compiler parameters; see the *Chez Scheme User's Guide* for details.

2.5. Faster continuations (5.0d)

Changes have been made in the continuation handling code that make both obtaining and invoking continuations. Performance gains for highly continuation-intensive code can measure as much as 50%, although gains for typical programs are more modest.

3. Bugs Fixed Since Version 5

3.1. Float printing (5.9i)

The floating-point printing algorithm now takes into account the input rounding algorithm to avoid printing more digits than necessary. This change brings the printer up-to-date with the floating-point printing algorithm described in "Printing floating-point numbers quickly and accurately" by Robert G. Burger and R. Kent Dybvig, which was presented at the SIGPLAN PLDI conference in 1996.

3.2. Ambiguous printing of symbol names starting with `@` (5.9i)

The printer now escapes symbol names that start with `@` to avoid an ambiguity that can arise if the symbol is preceded in the output by a comma produced as a result of printing an `unquote` form. Prior to this change, `(unquote @x)` and `(unquote-splicing x)` would both print as `,@x`.

3.3. Incorrect ceiling behavior (5.9i)

A bug in `ceiling` when invoked on real numbers represented as inexact complexnums, e.g., `3.2+0.0i`, has been fixed. The bug resulted in `ceiling` computing the `floor` of its input instead.

3.4. Storage management bug (5.9g)

A bug that sometimes resulted in invalid memory references for programs that allocate large quantities of medium- to large-sized objects has been fixed.

3.5. Invalid memory fault (5.9d)

A bug that caused spurious (and rare) invalid memory references on Alpha processors has been fixed.

3.6. Integer division bug (5.9a)

A bug in that caused integer division operators to return 0 instead of -1 when dividing the most-negative fixnum by its additive inverse (the least-positive bignum) has been fixed.

3.7. Signal handling fixes (5.0f and 5.0g)

A bug in signal handling whereby sigchild interrupts during a read from the console were mistaken for keyboard interrupts has been fixed. Also, a bug that caused keyboard interrupt handling to revert to the operating system default behavior on some systems after calls to `system` or `process` has been fixed.

3.8. Multiple return values bug (5.0d)

A bug in the interaction of the `or` syntactic form and multiple return values has been fixed. This bug would result in the erroneous signaling of the error, "incorrect number of values received in multiple value context."

3.9. Memory fault bug under NeXTSTEP (5.0d)

An bug that caused spurious memory faults under NeXTSTEP has been fixed. This bug could also sometimes result in an EMT or illegal instruction trap.

3.10. Nested `quasiquote/unquote-splicing` bug (5.0b)

A bug that caused `unquote-splicing` forms within nested `quasiquotes` to be spliced improperly into the surrounding list structure has been fixed. For example:

```
(let ((x 1) (y 2))
  `(foo (,x ,y) `(bar ,@(baz ,y))))
```

evaluated to:

```
(foo (1 2) `(bar (unquote-splicing baz 2)))
```

but now correctly evaluates to:

```
(foo (1 2) `(bar ,@(baz 2)))
```

3.11. Weak pair cdr mutation bug (5.0b)

A bug that sometimes caused the cdr field of a weak pair to become corrupt after restoring a saved heap has been fixed.

3.12. End-of-file on block read from generic port (5.0b)

The procedure `block-read` incorrectly required that generic port handlers for block read return a nonnegative fixnum, when in fact they should also be permitted to return the end-of-file object (`#!eof`). This has been fixed.

3.13. Bug in `string-fill!` (5.0b)

A bug in `string-fill!` that caused it to fill beyond the end of its argument string has been fixed.

3.14. Memory fault on incorrect argument count (5.0b)

A compiler bug that resulted in a memory fault instead of a meaningful error for certain expressions has been fixed. For example, the expression

```
(letrec ([a (lambda (v) v)]) ((begin 'foo a)))
```

resulted in an invalid memory reference, but now correctly signals an "incorrect argument count" error.

3.15. Register allocation bug (5.0a)

Some applications involving several arguments were evaluated incorrectly, resulting in one argument receiving the value of another. For example:

```
(let ()
  (define a (lambda (x) x))
  (define b (lambda (k) (k 'b)))
  (define c (lambda (k)
              (b (lambda (y)
                   (let ([x 'c])
                     (k x y (a #f) (a #f)))))))
  (c list))
```

should evaluate to `(c b #f #f)` but actually evaluated to `(c #f #f #f)`. This bug has been fixed.

3.16. Error saving heaps on SGI systems under IRIX 5.2 (5.0a)

Attempts to save heap files under IRIX 5.2 failed with the error message "cannot set brk". This was caused by a bug in the interaction between `brk` and memory-mapped files under IRIX 5.2. Version 6 avoids using `brk` when saving heaps so that this no longer causes a problem.