Chez Scheme Version 6.9b Release Notes

August 2003

1. Overview

This document outlines the changes made to *Chez Scheme* for Version 6.9b since Version 6.0.

Version 6.9b is available for the following platforms:

- Intel 80x86 Linux
- Intel 80x86 Windows 95/98/ME/NT/2000/XP
- Apple PowerPC Mac OS X
- Sun Sparc Solaris (32-bit)
- Sun Sparc Solaris (64-bit)

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 prerelease to support the change.

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, 3rd edition* and the *Chez Scheme User's Guide: Version 7.*

2. Functionality Changes

2.1. Expanded `format` functionality (6.9b)

`format` and related procedures, including `printf`, `fprintf`, and the default warning and error handlers, have been enhanced to accept all of the Common Lisp format directives, except for those directly related to pretty printing.

Here are some scintillating examples of this over-engineered utility for your entertainment.

- floating-point printing—3 digits to right of decimal point

  `(format "~,3f" 3.14159)` ⇒ `"3.142"`

- scientific notation—field of width 12 padded with asterisks, 3 digits to right of decimal point, 3 digits for exponent

  `(format "~12,3,3,,,'*e" 1e23)` ⇒ `"**1.000e+023"`

- monetary notation—default two digits to right of decimal point.

  `(format "$~$" (* 39.95 1.06))` ⇒ `"$42.35"`

- plurals

  `(format "~s bear~:p in ~s den~:p" 10 1)` ⇒ `"10 bears in 1 den"`

- printed numbers

```
(format "~r" 2599)  ⇒   "two thousand five hundred ninety-nine"
(format "~:r" 99)  ⇒   "ninety-ninth"
```

- roman numerals

```
(format "~@r" 2599)  ⇒  "MMDXCIX"
```

- case conversion

```
(format "~@(~r~)" 2599)  ⇒   "Two thousand five hundred ninety-nine"
(format "~@:(~a~)" "Ouch!")  ⇒   "ouch!"
```

There's much, much more, including conditionals, iteration, indirection, and justification.

In addition, `format` can now be used instead of `printf` and `fprintf` to print to the current output port or to another port, ala Common Lisp `format`. Here are the different modes supported by `format`.

`(format` *string arg* `...)`	evaluates to formated string (as always)
`(format #f` *string arg* `...)`	evaluates to formated string
`(format #t` *string arg* `...)`	prints to current-output port
`(format` *port string arg* `...)`	prints to *port*

2.2. New environment procedures (6.9b)

Support for creating new environments has been added, in the form of a new `copy-environment` procedure. `copy-environment` takes from one to three arguments.

```
(copy-environment env)
(copy-environment env mutable?)
(copy-environment env mutable? syms)
```

`copy-environment` returns a copy of *env*, i.e., a new environment that contains the same bindings as *env*.

The environment is mutable if *mutable?* is omitted or true; if *mutable?* is false, the environment is immutable. A mutable environment can be extended with new bindings, its existing bindings can be modified, and its variables can be assigned. An immutable environment cannot be modified in any of these ways.

The set of bindings copied from *env* to the new environment is determined by *syms*, which defaults to the value of `(environment-symbols` *env*`)`. The binding, if any, for each element of *syms* is copied to the new environment, and no other bindings are present in the new environment.

```
(define e (copy-environment (scheme-environment)))
(eval '(define cons +) e)
(eval '(cons 3 4) e)                         ⇒  7
(eval '(cons 3 4) (scheme-environment))  ⇒  (3 . 4)
```

The procedure `environment-symbols` takes an environment and returns a list of the symbols representing variables bound in the environment. It is primarily useful in building the list of symbols to be copied from one environment to another.

```
(define listless-environment
  (copy-environment
    (scheme-environment)
    #t
    (remq 'list (environment-symbols (scheme-environment)))))
(eval '(let ((x (cons 3 4))) x) listless-environment)  ⇒  (3 . 4)
(eval '(list 3 4) listless-environment)  ⇒  Error: variable list is not bound.
```

2.3. `interaction-environment` is now a parameter (6.9b)

The procedure `interaction-environment` has been converted into a full parameter which can be altered to change the environment used by default by various evaluation and expansion procedures. This parameter is consulted by `compile`, `interpret`, `compile-file`, `sc-expand`, `top-level-value`, `top-level-bound?`, `define-top-level-value`, and `set-top-level-value!`. Because `eval` and `load` use the current evaluator, i.e., the value of `current-eval`, and the default evaluator for Chez Scheme is `compile` and for Petite Chez Scheme is `interpret`, `eval` and `load` are also affected by the setting of `interaction-environment`. Similarly, `expand` uses the current expander, i.e., the value of `current-expand`, which defaults to `sc-expand`, so `expand` is also affected by `interaction-environment`.

```
> (define e (copy-environment (interaction-environment)))
> (parameterize ((interaction-environment e)) (new-cafe))
>> (define cons +)
>> (cons 3 4)
7
>> (exit)
> (cons 3 4)
(3 . 4)
```

2.4. Extended top-level-value procedures (6.9b)

An optional *environment* argument has been added to the procedures `top-level-bound?`, `top-level-value`, `define-top-level-value`, and `set-top-level-value!`. The environment argument is always last, if present.

```
(define cons +)
(cons 3 4)                                 ⇒  7
(top-level-value 'cons)                    ⇒  #<procedure +>
(top-level-value 'cons (scheme-environment))  ⇒  #<procedure cons>
```

2.5. New `trace-do` syntactic form (6.9b)

Loops written using `do` may now be traced via `trace-do`.

```
> (trace-do ([i 4 (- i 1)]) ((= i 0) 'done) (printf "~s\n" i))
|(do 4)
4
|(do 3)
3
|(do 2)
2
|(do 1)
1
|(do 0)
|done
done
```

2.6. No more undefined variable warnings (6.9a)

The compiler no longer generates a warning whenever an attempt to reference or assign an undefined locally defined or `letrec`-bound might occur, since these warnings are sometimes false warnings and quickly get old. Actual attempts to reference undefined variables are still detected and caught at run time. (These warnings were produced only by Version 6.9.)

2.7. New string-port operations (6.9a)

Two new operations on string-ports have been added:

`(with-input-from-string` *string thunk*`)` binds the current-input port to a string-input port created from *string* during the invocation of `thunk` and returns the value or values returned by *thunk*.

`(with-output-to-string` *thunk*`)` binds the current-output port to a new string-output port during the invocation of `thunk` and returns the string extracted from the string-output port when *thunk* returns.

2.8. `mkdir` and `chmod` (6.9a)

`mkdir` and `chmod` primitives have been added. `(mkdir` *pathname mode*`)` creates a directory with the given pathname. *mode* is optional and is used to set the permissions on the directory. `(chmod` *pathname mode*`)` changes the permissions on the file or directory at the given pathname according to *mode*. Under Windows, the *mode* argument to `mkdir` is ignored, and `chmod` does not do anything.

2.9. Top-level `import-only` (6.9a)

`import-only` now works at top level.

2.10. Better warnings and error messages (6.9a)

The compiler and run-time system now generate better warning and error messages for argument-count errors.

2.11. New `compile-script` procedure (6.9a)

A new primitive procedure, `compile-script`, has been added. It is like `compile-file` but differs in two important ways: (1) it copies the leading `#!` line from the source-file script into the object file, and (2) it does not compress the resulting file. This permits compiled script files to be created from source script files for improved efficiency.

2.12. Environments (6.9a)

The values returned by `scheme-report-environment`, `null-environment`, and `interaction-environment` are now of type environment, a unique type distinct from all others. The predicate `environment?` may be used to identify environments.

2.13. Source annotations in compiled files (6.9a)

Source annotations are no longer kept within syntax objects written to compiled files when generation of inspector information is disabled. This makes the compiled files smaller and eliminates the possibility of confusing source file references in error messages produced by production code.

2.14. Bignum shift counts (6.9a)

`ash` now permits the use of very large (bignum) shift counts, although such large shift counts are of dubious value.

2.15. Incompatible records and compiled files (6.9a)

An error is now signaled when an incompatible record is loaded from a compiled file, just as it is when an incompatible record is created by source code.

2.16. Mac OS X Version (6.9)

Full support for *Chez Scheme* on PowerPC based Apple Computers running Mac OS X has been added.

2.17. Better feedback from the reader (6.9)

The reader now provides better feedback when certain lexical errors are detected during the loading or compiling of source files. A major improvement is that end-of-file errors now report what kind of datum was being read and the file position at which the datum appeared, rather than the (useless) position of the actual end of file.

2.18. Records and generativity (6.9)

Each record type is associated with a unique generated name, or gensym. This gensym is used to identify the record type in the printed and compiled representations of each record of the type. When the name argument to `make-record-type` is a gensym, this gensym is used; otherwise a new gensym is created. If `make-record-type` is invoked more than once with the same gensym and the record fields match, the same record type descriptor is returned. If the record fields do not match, an error is signaled. This permits the programmer to control the generativity of record types; full nongenerativity is achieved by passing a gensym, and various levels of generativity are achieved by passing in a non-gensym at expansion time or run time.

`define-record` is ordinarily expansion-time generative, meaning that each time an expression is expanded, e.g., when it is contained in a file loaded for source or compiled by `compile-file`, a new record type is constructed. If the name of the record is specified as a gensym, however, the record type is fully nongenerative. This allows the programmer to place two identical definitions of a record type in two files to be separately compiled or loaded. Only the pretty name of the gensym is used in forming the variables naming the record constructor, predicate, accessors, and setters. For example:

```
(define-record foo ((immutable a)))
```

and

```
(define-record #{foo |*5CNSgDOG8+wd\\%|} ((immutable a)))
```

each implicitly define `make-foo`, `foo?`, and `foo-a`, but the latter is nongenerative while the former is expansion-time generative.

2.19. `letrec` changes (6.9)

A new form of `letrec`, called `letrec*`, has been added. `letrec*` is like `letrec` except that the right-hand sides are evaluated in order from left to right. The scoping rules are the same: `letrec*` and `letrec` both support full mutual recursion among the bound variables.

A new parameter `internal-defines-as-letrec*`, which defaults to `#f` in this release, causes the internal definitions of a `lambda` or similar body to expand into a `letrec*` form rather than `letrec`, effectively sequentializing their evaluation from left to right. This parameter must be set at expansion time (usually evaluation, loading, or compile time) to have any effect.

The system now signals an error at run time for any attempt to use a `letrec` or `letrec*`-bound variable (including variables defined by internal definitions) before it is valid to do so, i.e., any attempt to evaluate a reference or assignment before entering the body of the `letrec` or `letrec*`. The compiler generates a warning whenever such an error might occur. (This is no longer true; see Section 2.6.)

2.20. New `let-values` syntactic form (6.9)

Support for a new `let-values` form has been added. `let-values` is a convenient way to receive multiple values and bind them to variables. It is structured like `let` but permits an arbitrary formals list (like `lambda`) on each right-hand side. For example:

```
> (let-values ([(a b) (values 1 2)] [c (values 1 2 3)])
    (list a b c))
(1 2 (1 2 3))
```

2.21. `open-input-output-file` (6.9)

The new procedure `open-input-output-file` opens the file named by its first argument for reading and writing. It accepts an optional additional argument specifying the set of open options. Currently supported options include `exclusive`, `nonexclusive`, `buffered`, `unbuffered`, and `mode`. Input/output files are usually closed using `close-port` but may also be closed with either `close-input-port` or `close-output-port`.

2.22. New file open options (6.9)

The various file opening operations now accept an `exclusive` option, which establishes a lock on the opened file for as long as it remains open. On some systems the lock is advisory, i.e., it inhibits access by other processes only if they also attempt to open exclusively. A complementary `nonexclusive` option is supported as well, but is never needed since files are opened nonexclusively by default.

Operations that open output files (or input/output files) also support a new `mode` option, which determines the permission bits on Unix systems when the file is created by the operation. The following element in the options list must be an exact integer specifying the permissions in the manner of the Unix `open` function.

The following example demonstrates the use of both new options, along with the existing `truncate` option, to open a file exclusively for writing, truncating the file if it exists, and setting the permissions if the file must be created to allow read and write access by the owner, read access by the group, and no access by others.

```
(define p
  (open-output-file "/tmp/shared"
    '(exclusive truncate mode #o640)))
```

2.23. `truncate-file` (6.9)

The new procedure `truncate-file` truncates to zero length the file associated with its port argument and repositions the port to the beginning of the truncated file.

2.24. Error handling change (6.9)

It is no longer an error to return from the error handler. Instead, this is now the appropriate way to reset to the current cafe after an error is handled, if a reset is desired. Returning rather than explicitly reseting allows the system to clean up any aborted operations, e.g., to close any files opened by `load` or compile-file that are aborted as a result of the reset. The default error handler used to take care of this explicitly; this change allows new error handlers to be written independently of the default error handler that perform this same cleanup.

2.25. Logical operations on exact integers (6.9)

Version 6.9b implements the binary logical operations `logand`, `logor`, and `logxor`, along with unary `lognot`, for exact integers, including arbitrary-precision integers (bignums). Support for the fixnum versions of these operators is still present.

2.26. Generalized `syntax-case` patterns (6.9)

`syntax-case` patterns have been generalized to allow a fixed number of items after an ellipsis, allowing patterns like `(a b ... c d)` or `(a b ... c . d)`. In essence, the restriction that ellipses appear only at the end of a list or vector pattern has been replaced by a lesser restriction that only one ellipsis appear within a given list or vector pattern.

A pattern variable following an ellipsis and a dot, e.g., `d` in `(a b ... c . d)` matches only non-list items. For example, `(a ... . b)` matched against `(1 2 3)` binds `a` to `(1 2 3)` and `b` to `()`.

Because `syntax-rules` is defined in terms of `syntax-case`, `syntax-rules` patterns have been generalized similarly.

2.27. "Defun" syntax for `define-syntax` (6.9)

`define-syntax` now supports the so-called "defun" syntax of `define`, e.g.,

```
(define-syntax (when x)
  (syntax-case x ()
    [(_ e0 e1 e2 ...) #'(if e0 (begin e1 e2 ...))]))
```

This syntax is not useful for when `syntax-rules` is used, since the argument to a `syntax-rules` transformer is implicit.

2.28. Heap search change (6.9)

Search for a heap does not terminate when a heap of the expected name is found unless the heap is compatible with the executable. This allows the user to have heaps for multiple builds of the system in the heap search path, without interference.

2.29. Machine-type check (6.8)

The `load` procedure now verifies that compiled object files loaded with `load` were produced for the current machine type and signals an error if there is a machine-type mismatch.

2.30. Support for scripting (6.8)

Several changes have been made to support the use of *Chez Scheme* and Petite *Chez Scheme* for shell scripts:

- A new "`--script` *scriptfile*" command-line argument tells Chez Scheme to run in script mode, with *scriptfile* as the script-file name. In script mode, the interactive Chez Scheme greeting and prompts are suppressed, i.e., `--quiet` is implied, and the procedure value of the parameter `scheme-script` is applied to the script-file name and the other command-line arguments not otherwise interpreted by Chez Scheme. These include earlier filename arguments and all remaining command-line arguments, as if `--` had been supplied after *scriptfile*.
- The default value of the parameter `scheme-script` sets the parameter *command-line-arguments* to the additional command-line arguments, then loads the script file.

- `load` ignores the first line of any file that begins with `#!` followed by a space or slash.

Taken together, these allow Scheme-coded shell scripts to be written, such as the following implementation of the traditional Unix `echo` command, and run in shells that support such shell scripts.

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

The location of the `scheme` varies depending upon how it was installed, so the first line may have to be adjusted. `petite` will work as well as long as the compiler is not required to run the shell script.

2.31. More command-line option changes (6.8)

The set of command-line arguments has been expanded and refined. Here is the present set as presented by `scheme --help` or `petite --help`.

`-b` *path*, `--boot` *path*	load boot file
`-c`, `--compact`	toggle compaction flag
`-h` *path*, `--heap` *path*	load heap file
`-q`, `--quiet`	suppress greeting and prompt
`-s[`*n*`]` *path*, `--saveheap[`*n*`]` *path*	save heap file
`--script` *path*	run as shell script
`--version`	print version and exit
`--help`	print help and exit
`--`	pass through remaining args

2.32. Saved heap changes (6.7)

Heaps explicitly named by the `-h` (`--heap`) option are now resolved using the SCHEMEHEAPDIRS path unless the full (absolute or relative) path is specified.

Each heap is now associated with a date stamp created when the corresponding executable image is created, and this date stamp is checked to insure heap compatibility.

2.33. String escapes (6.7)

New reader syntax for including special characters in strings has been added, along with character syntax for audible bell and vertical tabs. The new and existing character names and string escapes are summarized in the table below.

string escape	*character equivalent*	*description*
`\a`	`#\bel`	audible bell
`\b`	`#\backspace`	back space
`\f`	`#\page`	form feed
`\n`	`#\newline`	newline
`\r`	`#\return`	carriage return
`\t`	`#\tab`	tab
`\v`	`#\vt`	vertical tab
`\\`	`#\\`	back slash
`\"`	`#\"`	double quote
`\'`	`#\'`	single quote
`\nnn`	`#\nnn`	ascii code nnn (octal)
`\xnn`		ascii code nn (hexadecimal)

The reader signals an error for any escape sequence not listed above, e.g., for `"\c"`.

The string escape for single quote is not useful in Scheme; it is included to simplify interoperability with ANSI C.

2.34. New `revisit` procedure (6.7)

The new `revisit` procedure loads and executes run-time code only from a compiled file, i.e., that portion of the code not loaded by `visit`.

Some of the code in a compiled object file can be classified as compile-time or run-time code. Compile-time code establishes compile-time identifier bindings or performs other tasks that are required for subsequent compilation. This includes keyword bindings for syntactic abstractions and module information, and may also include any code explicitly specified as “visit-time” code using `eval-when` (see below).

Run-time code is code that is not required for subsequent compilation; this includes the code to establish the run-time values of variables and other computations that are not specified as “visit-time” only.

The standard procedure `load` loads (and runs) all of the code in an object file, i.e., both compile-time code and run-time code. The *Chez Scheme* procedure `visit`, introduced in Version 6.0, loads only the compile-time portion of the code. The new *Chez Scheme* procedure `revisit`, loads only the run-time portion of the code.

Loading a file by visiting then revisiting the file is almost, but not quite, equivalent to loading it with `load`. The difference is the order in which the items in the file are processed. In the latter case, the loading of compile-time and run-time code is interleaved as it appeared in the original source file. This difference is typically unimportant.

2.35. New `eval-when` situations (6.7)

Two new `eval-when` situations, `visit` and `revisit`, are now supported. Like the existing situations `compile` and `load`, these new situations are relevant only when compiling a file (using `compile-file`). As implied by their names, these situations direct the compiler to generate code to be evaluated when the resulting object file is visited or revisited using the `visit` or `revisit` procedures.

`eval-when` is often used to establish variable bindings, such as help procedures, that are needed by syntax transformers. Since syntax transformers are evaluated, by default, at compile, visit, and eval times, variable bindings needed by syntax transformers should be wrapped in an `eval-when` form with situations `compile`, `visit`, and `eval`.

The behavior of `eval-when` is usually intuitive but can be understood precisely as described here. The `syntax-case` expander, which handles `eval-when` forms, maintains two state sets, one for compile-time forms and one for run-time forms. The set of possible states in each set are “L” for `load`, “C” for `compile`, “V” for `visit`, “R” for `revisit`, and “E” for `eval`.

When compiling a file, the compile-time set initially contains "L" and "C" and the run-time set initially contains only "L." When not compiling a file (as when a form is evaluated by the read-eval-print loop or loaded from a source file), both sets initially contain only "E." The subforms of an `eval-when` form at top level are expanded with new compile- and run-time sets determined by the current sets and the situations listed in the `eval-when` form. Each element of the current set contributes zero or more elements to the new set depending upon the given situations according to the following table.

	`load`	`compile`	`visit`	`revisit`	`eval`
L	L	C	V	R	—
C	—	—	—	—	C
V	V	C	V	—	—
R	R	C	—	R	—
E	—	—	—	—	E

For example, if the current compile-time state set is {L} and the situations are `load` and `compile`, the new compile-time state set is {L, C}, since L/`load` contributes "L" and L/`compile` contributes "C."

The state sets determine how forms are treated by the expander. Compile-time forms such as syntax definitions are evaluated at a time or times determined by the compile-time state set, and run-time forms are evaluated at a time or times determined by the run-time state set. A form is evaluated immediately if "C" is in the state set. Code is generated to evaluate the form at visit or revisit time if 'V" or "R" is present. If "L" is present in the compile-time set, it is treated as "V;" likewise, if "L" is present in the run-time set, it is treated as "R." If more than one of states is present in the state set, the form is evaluated at each specified time.

"E" can appear in the state set only when not compiling a file, i.e., when the expander is invoked from an evaluator such as `compile` or `interpret`. When it does appear, the expanded form is returned from the expander to be processed by the evaluator, e.g., `compile` or `interpret`, that invoked the expander.

The initial compile-time state set can be controlled by the parameter `eval-syntax-expanders-when`. This parameter is bound to a list of situations, which defaults to `(compile load eval)`. When compiling a file, `compile` contributes "C" to the state set, `load` contributes "L," `visit` contributes "V," `revisit` contributes "R," and `eval` contributes nothing. When not compiling a file, `eval` contributes "E" to the state set, and the other situations contribute nothing. There is no corresponding parameter for controlling the initial value of the run-time state set.

2.36. `eval-when` and definitions (6.7)

`eval-when` may now appear anywhere that definitions may appear and is considered to be a definition syntactically in those contexts.

An `eval-when` in an internal definition context is treated as a splicing construct, like `begin`, if the list of situations includes `eval`. That is, the sequence of forms within the `eval-when` form are treated as if they appeared in place of the `eval-when` form. If the situation list does not include `eval`, the `eval-when` form is ignored, i.e., the forms in the body are dropped.

2.37. `record-writer` and `type-descriptor` (6.6)

A print procedure may be established for record type via the `record-writer` procedure, which accepts a record-type descriptor and a procedure. The procedure is passed an object (an instance of the record type), an output port, and a write procedure. The write procedure must be used in situations where a call to `write` would ordinarily be used by the record writer. The print procedure may be called multiple times to print a single object to support cycle detection and graph printing. Thus, it should have no side effects other than writing to the given port argument.

When called with a record-type descriptor only, `record-writer` returns the record writer currently registered for the record type, which is the default record writer if no record writer has been registered for the record type.

This facility replaces the `print-method` option in `define-record`, which was eliminated when record types produced by `define-record` were made nongenerative. (See “Record inheritance and other changes” below.)

A new syntactic form, `type-descriptor`, has been introduced. When given the name of a record defined by `define-record`, `type-descriptor` returns the record-type descriptor for that record.

```
(define-record point (x y))
(type-descriptor point)  ⇒  #<record type point>
```

In earlier versions, it was necessary to create an instance of the record type and extract the descriptor with `record-type-descriptor`. Using `type-descriptor` is both cleaner and more efficient.

2.38. File position on string ports (6.6)

`file-position` now works for string ports.

2.39. Record accessor changes (6.6)

`define-record` no longer defines accessors/mutators for inherited fields. It signals an error if duplicate field names are defined. Child field names can, however, overlap with parent field names, and duplicate field names can arise via the low-level `make-record-type` interface.

`record-field-accessor` and `record-field-mutator` now accept field ordinals as well as field names. A field's ordinal is its zero-based position in the list of fields returned by `record-type-field-names`. Field ordinals are used by the inspector and printer to avoid problems with duplicate field names.

2.40. Changes to the debug and break handlers (6.6)

The debug handler, invoked as a result of an explicit call to `debug`, supports several new or altered options: `s` to show the current error message, `e` to exit and retain the error continuation, and `q` to quit and discard the error continuation, Discarding the error continuation can reduce storage retention, although this is often not a problem while debugging. When running multithreaded, the debug handler supports inspection of error continuations from any thread.

The break handler, invoked as the result of a keyboard interrupt or explicit call to `break`, now displays the prompt `break>` rather than `debug>`, which is reserved for the now separate debug handler. The break handler is otherwise unchanged.

2.41. Expander change (6.6)

An incompatible change to the `syntax-case` expander has been made to reduce the loss of source information when one macro generates another macro definition. List and vector structure in the subexpression of a syntax form is no longer guaranteed to be list and vector structure in the output form except where pattern variables are contained within that structure. For example, `#'(a ...)`, where `a` is a pattern variable, is guaranteed to evaluate to a list, but the constant structure `#'(a b c d)`, where none of `a`, `b`, `c`, and `d` are pattern variables, may not. The practical consequence of this change is that constant structures must be deconstructed using `syntax-case` or `syntax-rules` rather than `car`, `cdr`, and other list-processing operations.

2.42. More generated symbol changes (6.6)

(See also "Generated symbol changes" below.) The "pretty" name of a generated symbol (gensyms) is now determined when the name is printed or obtained via `symbol->string`, rather than when the gensym is created. Changes to the parameters `gensym-prefix` and `gensym-count` are effective only when the gensym's name is determined. To get the old behavior, it is necessary to force the name to be determined when the gensym is created, using `symbol->string`.

2.43. Multithreading (6.5)

Chez Scheme now supports multithreading, including threads running on multiple processors. The thread support is implemented in terms of Posix Threads (pthreads). Information on the thread system may be found in the *Chez Scheme User's Guide: Version 7*.

2.44. Unbufferred Ports (6.5)

Chez Scheme buffers file I/O operations for efficiency, but buffered I/O is not thread safe. Two threads that write to or read from the same port can corrupt the port, resulting in buffer overruns and, ultimately, invalid memory references. When running the threaded version of *Chez Scheme*, the console output port is unbuffered by default. This allows multiple threads to print error and/or debugging messages to the console. The output may be interleaved, even within the same line, but the port will not become corrupted.

Files created by `open-input-file`, `open-output-file`, `call-with-input-file`, `call-with-output-file`, `with-input-from-file`, and `with-output-to-file` are buffered by default: Threads that wish to allow multiple threads to write to or read from the same file via a single port should use the `unbuffered` flag when opening the file, e.g.:

```
(define p (open-output-file "prog.out" '(unbuffered)))
```

Refer to the *Chez Scheme User's Guide: Version 7* for details.

2.45. 64-bit Sparc V9 architecture port of *Chez Scheme* (6.4)

Support for the Sparc 64-bit (V9) architecture has been added. Programs running in the 64-bit version (machine type "sparcv9") can create heaps larger than the four-gigabyte limit imposed by the 32-bit architecture. Because data structures are larger and code sequences needed to perform various operations are longer in the 64-bit version, the 32-bit version (machine type "sps2") is generally faster and should be used when heaps smaller than four gigabytes suffice. Because the data structures are incompatible, it is not possible to run both 32- and 64-bit code in the same process.

Foreign C code intended to be linked with the 64-bit version should be compiled using the C compiler's `-xarch=v9` option.

Under the 64-bit version, two new foreign/record data types are available: `integer-64` and `unsigned-64`. The `iptr` and `uptr` data types (see below) map to `integer-64` and `unsigned-64` respectively.

2.46. New foreign-interface datatypes (6.4)

Because Scheme objects may no longer be the size of an `int` in C, but in fact generally map to the size of a C pointer, the definition of "`ptr`" in the header file `scheme.h` has changed from `int` to `void *`. Two new datatypes, `iptr` and `uptr`, are defined by the header file as well; these are defined to be signed and unsigned integers of the same size as a `ptr`.

The foreign interface (`foreign-procedure` and `foreign-callable`) also recognizes `iptr` and `uptr` as argument and result type specifiers, mapping them to the appropriate size of signed or unsigned integer.

To further facilitate consistency between Scheme and C code, the foreign interface supports an additional new type, `(void *)`, that maps to an unsigned integer of the appropriate size.

The argument-type specifiers supported by `foreign-procedure` and `foreign-callable` are summarized below.

`integer-32`	32-bit signed integer
`integer-64`	64-bit signed integer (64-bit platforms only)
`unsigned-32`	32-bit unsigned integer
`unsigned-64`	64-bit unsigned integer (64-bit platforms only)
`iptr`	32- or 64-bit signed integer on 32- or 64-bit platform
`uptr`	32- or 64-bit unsigned integer on 32- or 64-bit platform
`fixnum`	integer in fixnum range (maps to `iptr`)
`single-float`	single-precision (32-bit) floating-point number
`double-float`	double-precision (64-bit) floating-point number
`boolean`	boolean value (maps to C `int`)
`char`	character value (maps to C `char`)
`string`	string (maps to C `char *`)
`scheme-object`	any Scheme object
`(void *)`	32- or 64-bit unsigned integer on 32- or 64-bit platform

The result-type specifiers supported by `foreign-procedure` include all of the above plus `void`. The result-type specifiers supported by `foreign-callable` include all of the above except `string`, plus `void`.

2.47. Foreign-interface datatype extensions (6.4)

The range of Scheme integers accepted as `foreign-procedure` arguments specified to be `integer-32` or `unsigned-32` has been extended. For both types, the accepted range is now -2^{31} through 2^{32-1}. In either case, all inputs are converted into the corresponding 32-bit two's complement representation, so that for `integer-32`, values of 2^{31} and above map to negative values and for `unsigned-32`, values below zero map to values of 2^{31} or greater.

The range values accepted by `integer-64` and `unsigned-64` is -2^{63} through 2^{64-1}, with a similar treatment of signed and unsigned values.

2.48. New record field types (6.3)

The record system (`define-record` and `make-record-type`) now supports signed and unsigned integers of size 8, 16, and, on 64-bit platforms, 64 bits, in addition to the previously supported 32-bit integers. It also supports single (32-bit) as well double (64-bit) floating-point numbers. The field type specifiers supported by the record system are summarized below.

`integer-8`	8-bit signed integer
`integer-16`	16-bit signed integer
`integer-32`	32-bit signed integer
`integer-64`	64-bit signed integer (64-bit platforms only)
`unsigned-8`	8-bit unsigned integer
`unsigned-16`	16-bit unsigned integer
`unsigned-32`	32-bit unsigned integer
`unsigned-64`	64-bit unsigned integer (64-bit platforms only)
`single-float`	single-precision (32-bit) floating-point number
`double-float`	double-precision (64-bit) floating-point number
`scheme-object`	any Scheme object (the default)

2.49. Fenders in `syntax-rules` (6.3)

The `syntax-rules` form has been extended to support fenders, with the same syntax and semantics of the fenders supported by `syntax-case`. This extension to `syntax-rules` should not be used in Scheme code intended to be portable to other Scheme systems, which do not generally support fenders.

2.50. Support for anonymous foreign procedures (6.3)

Support for anonymous foreign procedures has been added. When *entry-exp* in

`(foreign-procedure` *entry-exp* `(`*arg-spec* `...)` *result-spec*`)`

evaluates to an integer, it is assumed to be the address of the foreign entry point rather than its name.

2.51. Support for calling Scheme from C (6.3)

A new `foreign-callable` form has been added that permits C or other languages that can make C-compatible calls to invoke Scheme procedures. As with `foreign-procedure`, `foreign-callable` permits the specification of input and output types and automatically performs the necessary conversions (marshalling) to and from Scheme datatypes. Information on `foreign-callable` and how to use it may be found in the *Chez Scheme User's Guide: Version 7*. (At present, `foreign-callable` is supported only under Windows 9x/NT and Intel Linux. Support for other machine types will be added later.)

2.52. Support for file compression (6.3)

Support for transparent compression/decompression when writing to/reading from files has been added via inclusion of zlib compression code. Compression/decompression is performed whenever a file is opened using the optional `compressed` argument to one of the file open routines, including `open-input-file` and `open-output-file`. See the *Chez Scheme User's Guide: Version 7* for details.

The compression and decompression is performed with the use of the zlib compression library developed by Jean-loup Gailly and Mark Adler. It is therefore compatible with the `gzip` program, which means that `gzip` may be used to uncompress files produced by *Chez Scheme* and visa versa.

Object files produced by the compiler are compressed by default. Object-file compression may be disabled by setting the parameter `compile-compressed` to false.

2.53. Enhancement to `load` and `include` (6.3)

`load` and `include` are now sensitive to the `source-directories` parameter for non-absolute pathname arguments. `load` consults this parameter for both source and object files. This parameter had previously been used only by the inspector.

2.54. New `print-record` parameter (6.3)

A new parameter, `print-record`, has been added. When set to `#t` (the default), the printer prints records fully. When set to `#f`, the printer uses an abbreviated form that shows a record's name but not its contents.

2.55. Record inheritance and other changes (6.2)

The record system now supports inheritance. If the optional first argument to `make-record-type` is a record-type descriptor, it identifies the parent record type for the new record type. With `define-record`, a parent record type is identified by placing the name of the parent record type (which must have been established by an earlier use of `define-record`) after the new record type name in the `define-record` syntax.

```
(define-record name (field ...))
(define-record name parent-name (field ...))
```

The new record type inherits the fields of the parent. The parent type predicate returns true for instances of the child (and any of its children). The parent accessors and mutators work on child instances as well.

`define-record` is also now nongenerative, meaning that each time a given `define-record` form is evaluated, it produces the same record type. That is, the record type is created when the `define-record` form is compiled, not each time the resulting code is run.

The print method argument to `make-record-type` has been eliminated, as have both the `print-method` and `reader-name` `define-record` options.

2.56. Generated symbol changes (6.2)

Several changes to generated symbols (gensyms) have been made. In previous versions, symbols created by `gensym` were "uninterned," meaning they were not entered into the system's internal hash table. This is still true for gensyms in Version 6.9b until they are printed. When a gensym is printed, it is given a globally unique name and interned under that name. This allows them to be written in such a way that they can be read back and reliably commonized on input. A gensym's unique name can also be obtained by calling the new procedure `gensym->unique-string`, which, like printing, forces the unique name to be created if it has not already been created. Delaying the creation of unique names and the interning of gensyms as long as possible allows the system to reclaim gensyms that are no longer accessible even if they have top-level values or nonempty property lists.

Each gensym is also given a "pretty" name, which is formed as in past systems. By default, the pretty name consists of the letter "g" followed by a sequence number, e.g., `g25`. The pretty name is returned by `symbol->string`.

When `print-gensym` is set to true (the default), gensyms print as shown below.

`#{`*pretty-name unique-name*`}`

When set to false, gensyms print simply as *pretty-name*. The syntax `#:`*name* is no longer recognized by the reader.

The procedure `string->uninterned-symbol` no longer exists. In its place, `gensym` now accepts an optional second argument, a string to use as the pretty name. Similarly, the predicate `uninterned-symbol` has been replaced by `gensym?`. Compatibility definitions for both can be found in the file `examples/compat.ss` in the *Chez Scheme* distribution.

```
> (define g1 (gensym))
> (symbol->string g1)
"g1411"
> (symbol->string (gensym))
"g1412"
> g1
#{g1411 |"V*_da}f)+)Bp\\"|}
> (gensym->unique-string g1)
"\"V*_da}f)+)Bp\\\\\""
```

2.57. Automatic heap loading (6.2)

Support for automatic heap loading has been added. This eliminates the need for shell scripts to start *Chez Scheme*, Petite *Chez Scheme*, and Scheme-coded applications in most cases. If the system is invoked as `progname` *arg* `...` with no heap arguments, the system looks for progname.heap in a set of default locations, which can be overridden by setting the environment variable SCHEMEHEAPDIRS. Similarly, if lower-level heaps are not specified, the system looks for them in the same set of locations. SCHEMEHEAPDIRS is a colon-separated list of directories. Within SCHEMEHEAPDIRS, the current version, e.g., `6.2`, is substituted for `%v`, the machine type, e.g., `sps2`, is substituted for `%m`, and *char* is substituted for `%`*char* for all other characters to allow directory names to include `%v`, `%m`, and colons. If SCHEMEHEAPDIRS ends in a colon, the default directories are searched after the specified directories; otherwise only the specified directories are searched. When creating an application named progname, link `scheme` to `progname` and put `progname.heap` in one of the standard locations. See the *Chez Scheme User's Guide: Version 7* for details.

2.58. Command-line option changes (6.2)

The obsolete "-o" option and an undocumented "-t" (test) option are no longer recognized by *Chez Scheme*. The "–", "-b", "-c", "-h", and "-s*n*" options must now be followed by a null character; that is, they are no longer recognized as options if there are additional characters. So, for example, "-help" and "-s99" are passed along the scheme startup routine. Multiple "-b" (boot file) options are now accepted. (The "-b" option is used for base heap creation and is not of interest to most users.)

2.59. Compile-time checking of format strings (6.1)

Compile-time control-string and argument-count checking is now performed at optimize levels 2 and 3 for `format`, `printf`, `fprintf`, `error`, and `warning`.

2.60. Quasiquote extension (6.1)

`unquote` and `unquote-splicing` have been extended in an upwardly compatible way to allow multiple subforms in splicing (list or vector) contexts. `(unquote-splicing x ...)` now splices into the surrounding context the result of appending the values of `x ...`, and `(unquote x ...)` now inserts into the surrounding context the values of `x ...`. These changes support certain useful nested `quasiquote` idioms, such as `,@,@`, which has the effect of a doubly indirect splicing when used within a doubly nested and doubly evaluated `quasiquote` expression. For example,

```
(define x '(m n))
(define m '(b c))
(define n '(d e))
(eval ``(a ,@,@x f) (interaction-environment))
```

evaluates to `(a b c d e f)`.

2.61. Tracing syntax transformers (6.1)

Support for `trace-define-syntax` has been added. This form is like `define-syntax`, except that the transformer bound to the defined keyword is wrapped in tracing code that displays the input and output to the transformer in the same manner as `trace-lambda`. See the *Chez Scheme User's Guide: Version 7* for details.

2.62. Making ports from file descriptors (6.1)

`make-output-port`, `make-input-port`, and `make-input/output-port` now allow the first, *handler*, argument to be a nonnegative fixnum representing a low-level file descriptor, obtained via some operating-system dependent means. When *handler* is a file descriptor, the newly created port is given a handler that reads from and/or writes to the file descriptor. See the *Chez Scheme User's Guide: Version 7* for details.

2.63. Inspection of record-type descriptors (6.1)

The interactive and object inspectors now allow a record's record-type descriptor to be accessed and inspected. The command `rtd` is used while inspecting a record in the interactive inspector to inspect the record's record-type descriptor. The message `rtd` is used similarly in the object inspector.

2.64. Block Comments (6.0a)

The reader now allows a block of characters to be commented out within properly nested `#|`, `|#` pairs, without regard to line boundaries. For example, the expression

```
(+ #| This is a comment |# 3 4)
```

evaluates to 7, as does

```
(+ #| This is a comment
      with another #| nested
      inside, crossing several
      line |# boundaries |# 3 4)
```

2.65. Support for Windows "stdcall" Convention (6.0a)

Chez Scheme now supports the Windows `__stdcall` procedure calling convention on Intel 80x86 systems for foreign procedures created with `foreign-procedure`. The default is still the `__cdecl` convention, as with Microsoft's C compiler. `__stdcall` is specified by placing the keyword `__stdcall` before the foreign procedure name, e.g.:

```
(foreign-procedure __stdcall "GetVolumeInformationA"
  (string string unsigned-32 string string string string unsigned-32)
  boolean)
```

`__stdcall` is useful for calling Windows API functions, except for those with "varargs" interfaces, which use the default `__cdecl` convention.

3. Bug Fixes

3.1. Undefined variable detection bug (6.9b)

A bug in the compiler's handling of `letrec` and `letrec*` that allowed some undefined-variable references to go undetected has been fixed. [This bug dated back to Version 6.9.]

3.2. Sparc 64-bit Dynamic linker bug (6.9a)

A bug in the dynamic linker that could cause an invalid memory reference or illegal instruction trap when very large heaps are used has been fixed. [This bug dated back to Version 6.4.]

3.3. Sparc 64-bit `file-position` bug (6.9a)

`file-position` no longer complains when handed positions outside of the 32-bit range when running the 64-bit Sparc version of the system. [This bug dated back to Version 6.4.]

3.4. `expt` bug (6.9a)

A bug which caused excessive round-off error in `expt` has been fixed. [This bug dated back to Version 6.0.]

3.5. `compile-profile` bug (6.9a)

`compile-file` now determines the value of `compile-profile` once for each expression so that portions of a file can be instrumented for profiling, e.g., via the insertion of

`\scheme{(eval-when (compile) (compile-profile #t))}`

into the file. [This bug dated back to Version 6.0.]

3.6. `foreign-callable` memory fault (6.9a)

A bug in `foreign-callable` that resulted in occasional memory faults has been fixed. [This bug dated back to Version 6.3.]

3.7. Laziness in local transformer evaluation (6.9)

To avoid unnecessary work at expansion time, the expander defers the evaluation of local transformers, i.e., those bound by `let-syntax`, `letrec-syntax` or internal `define-syntax` forms, until they are actually used. To avoid this laziness from being detectable, the expander now defers the evaluation only for transformer expressions that expand into `lambda` expressions. [This bug dated back to Version 6.0.]

3.8. Inspector output bug (6.9)

The inspector now properly prints the output of the `=>` and `eval` messages to the console output port rather than to the current output port, if different. [This bug dated back to Version 6.0.]

3.9. Reader bug (6.9)

A bug in the reading of records, which sometimes made the annotations created by the reader visible within record constants appearing in files loaded from source or compiled, has been fixed. The bug occurred when a record field was constructed using a graph reference to a marked portion of the structure outside of the record, e.g., `'#1=(#[foo #1#])` [This bug dated back to Version 6.0.]

3.10. Compiler bug (6.9)

A bug in the compiler that caused inappropriate "incorrect argument count" warnings and errors when invoking a locally bound procedure created with `case-lambda` has been fixed. [This bug dated back to Version 6.0.]

3.11. Expander bug (6.9)

A bug in the expander causing unexpected "out of context" errors for exports of identifiers with the same name as an internal unexported identifier has been fixed. [This bug dated back to Version 6.0.]

3.12. Interpreter evaluation-order bug (6.8)

An interpreter evaluation-order bug has been fixed, preventing the interpreter from signaling an error prematurely for attempts to invoke nonprocedures when the call is never actually made. For example, the interpreter used to signal an error for the following program, but now properly returns `1`.

```
(call/cc (lambda (k) (0 (k 1))))
```

This fix affects only programs running under Petite Chez Scheme or programs running under Chez Scheme that explicitly use the interpreter. [This bug dated back to Version 6.0.]

3.13. Inlining bug (6.8)

A bug in the compiler's inlining of certain indirect calls to procedures that are provided an incorrect number of arguments has been fixed. The bug previously resulted in an invalid memory reference during compilation; it now results in an appropriate run-time "incorrect number of arguments" error. [This bug dated back to Version 6.0.]

3.14. Generated record constructor fixes (6.8)

A bug in `record-constructor`'s algorithm for determining record sizes that resulted in more allocation than necessary has been fixed. A separate bug that could result in "invalid parameter list" errors while creating the record constructor has also been fixed.

$$These bugs dated back to Version 6.0.$$

3.15. Locked object bugs (6.5/6.6)

Bugs in the handling of objects locked via `lock-object` have been fixed. These bugs sometimes resulted in invalid memory references or failure to properly lock objects.

$$These bugs dated back to Version 6.0.$$

3.16. Block comment bug (6.4)

A bug in the reader that prevented certain block comments from being treated properly has been fixed. [This bug dated back to Version 6.0.]

3.17. Application of unbound variables (6.3)

A bug that resulted in an invalid memory reference rather than a sensible error message when applying the value of certain unbound variables has been fixed. [This bug dated back to Version 6.2.]

3.18. Assignments to unreferenced `letrec`-bound variables (6.3)

A bug that sometimes resulted in an invalid memory reference for programs that contain assignments to unreferenced `letrec`-bound variables has been fixed. [This bug dated back to Version 6.0.]

3.19. Setting primitive names to nonprocedures (6.2)

Setting a primitive name, e.g., `car` or `map`, to a nonprocedure values, then invoking it, resulted in an invalid memory reference. This bug has been fixed. [This bug dated back to Version 6.0.]

3.20. Trace error (6.2)

A bug in the trace package, which prevented further uses of `trace` and `untrace` if a traced procedure was redefined as a syntactic abstraction, has been fixed. [This bug dated back to Version 6.0.]

3.21. Module names introduced by macros (6.1)

The expander now properly renames module names introduced into the expansion of a macro at top level, as it already did for other defined identifiers. [This bug dated back to Version 6.0.]

3.22. Interpreter `case-lambda` bug (6.1)

A bug in the interpreter's treatment of `case-lambda`, which caused argument-count errors to be produced for certain correct applications of procedures created by `case-lambda`, has been fixed. [This bug dated back to Version 6.0.]

3.23. IRIX memory limitation (6.1)

A limitation on the maximum size of a heap under IRIX caused by inappropriate linker assumptions about the operating environment has been lifted. In theory, the size of a heap is now limited only by available (32-bit) virtual memory, and in practice we have been able to test heaps of up to about two gigabytes in size. [This bug dated back to Version 6.0.]

3.24. Storage-Management and Records (6.0a)

A bug in the garbage collector's handling of records containing both mutable pointer (`scheme-object`) and nonpointer (`integer-32`, `unsigned-32`, or `double-float`) fields has been fixed. The bug sometimes caused unrecoverable invalid memory references. [This bug dated back to Version 6.0.]

3.25. Source-Tracking Problems (6.0a)

A couple of problems with source tracking in the compiler have been fixed. These problems sometimes resulted in memory faults within the inspector or in misleading information being printed as part of syntax error message. An unrelated bug with finding source files with pathnames that start with a drive letter under Windows has also been fixed. [This bug dated back to Version 6.0.]

3.26. Overeager Argument-Count Checking (6.0a)

The compiler now signals a warning rather than an error when it detects a call to a locally bound procedure with an incorrect number of arguments. This prevents the compiler from rejecting programs such as the following, which might otherwise execute without errors:

```
(let ((f (lambda (x) x)))
  (if (read)
      (f 3)
      (f 3 4)))
```

It also prevents the compiler from rejecting programs with less obvious "errors," uncovered during procedure inlining, such as the following:

```
(let ((f (lambda (g n)
           (if (= n 1)
               (g (lambda (x) x))
               (g (lambda (x y) x))))))
  (f (lambda (h) (h 3)) 1))
```

[This bug dated back to Version 6.0.]

4. Performance Enhancements

4.1. Faster fixnum operations (6.9a)

Fixnum addition and subtraction operators, which had been open-coded only at optimize-level 3, are open-coded at optimize-level 2 as well.

4.2. Faster `append` and `apply` (6.9a)

The compiler now generates better code for `append` and `apply` at optimize-level 3.

4.3. Faster `syntax-case` expander (6.9a)

Improvements have been made in the implementation of `syntax-case` to make it significantly faster for some extreme expansion problems involving many levels of macro expansion.

4.4. Inexact numbers with large exponents (6.9a)

Reading of inexact numbers with very large positive of negative exponents is now much faster; the system checks for such large exponents and immediately returns the appropriate zero or infinity value.

4.5. Giving memory back to the O/S (6.6)

Version 6.9b returns memory to the operating system when possible following the collection of the oldest generation.

4.6. Faster bignum arithmetic (6.4)

Internal changes to the bignum representation and bignum arithmetic package have resulted in a modest speedup of bignum operations, typically by around 10-15%.

4.7. Vector handling in `quasiquote` (6.1)

Handling of certain vectors in `quasiquote` has been improved. For example, `` `#(a 1 ,(+ 3 4) x y) `` now expands into the following code.

```
(#2%vector 'a 1 (+ 3 4) 'x 'y)
```

Previously, the same expression expanded into code that generated a list and converted this list into a vector via `list->vector`.