Chez Scheme Version 5 Release Notes
(Preliminary)

August 1994

Overview

This document outlines the changes made to *Chez Scheme* for Version 5 since Version 4. New items include support for `syntax-case` and revised report high-level lexical macros, support for multiple values, support for guardians and weak pairs, support for generic ports, improved trace output, support for shared incremental heaps, support for passing single floats to and returning single floats from foreign procedures, support for passing structures to and returning structures from foreign procedures on MIPS-based computers, and various performance enhancements, including a dramatic improvement in the time it takes to load Scheme source and object files on DEC Ultrix MIPS-based computers.

Overall performance of generated code has increased by an average of 15–50 percent over Version 4 depending upon optimize level and programming style. Programs that make extensive use of locally-defined procedures will benefit more than programs that rely heavily on globally-defined procedures, and programs compiled at higher optimize levels will improve more than programs compiled at lower optimize levels. Compile time is approximately 25 percent faster.

Refer to the *Chez Scheme System Manual*, Revision 2.5 for detailed descriptions of the new or modified procedures and syntactic forms mentioned in these notes.

Version 5 is available for the following platforms:

- Sun Sparc SunOS 4.X & Solaris 2.X
- DEC Alpha AXP OSF/1 V2.X
- Silicon Graphics IRIX 5.X
- Intel 80x86 NeXT Mach 3.2
- Motorola Delta MC88000 SVR3 & SVR4
- Decstation/Decsystem Ultrix 4.3A
- Intel 80x86 BSDI BSD/386 1.1
- Intel 80x86 Linux

This document contains four sections describing (1) functionality enhancements, (2) performance enhancements, (3) bugs fixed, and (4) compatibility issues.

1. Functionality Enhancements Since Version 4

1.1. Lexical macros

Support for the `syntax-case` macro system, a fully general lexical syntactic extension facility that includes the revised report high-level macro system as a subset, has been incorporated. The system provides automatic pattern-based syntax checking, input destructuring, and output restructuring, like its predecessor `extend-syntax`, while respecting lexical scoping of identifiers in macro definitions and macro calls. Whereas the pattern facilities of `extend-syntax` must be abandoned when writing "low-level" macros, the pattern facilities of `syntax-case` are available to all macros. In fact, the system draws no distinction between high- and low-level macros, so there is never a need to completely rewrite a macro originally written in a high-level style simply because it needs to perform some low-level operation. The system supports a controlled form of identifier capture that allows most common "capturing" macros to be written without violating the spirit of lexical scoping. Local macro definitions are supported, and internal syntax definitions may be freely intermixed with internal variable definitions. The more restrictive revised report `syntax-rules` form is provided in addition to `syntax-case`, but is in fact itself a macro defined with and in terms of `syntax-case`.

Most existing `extend-syntax` macros can be redefined easily using `syntax-case` or `syntax-rules`. Some "expansion-passing style" (EPS) macros, however, are not easily expressed; specifically, those that made use of fluid binding during macro expansion and those that made assumptions about the (undocumented) structure of the macro expander output. A compatibility package supporting both EPS and `extend-syntax` will be provided for a limited time.

1.2. Multiple values

Complete and efficient support for the proposed Scheme multiple values interface has been added. The primitive Scheme syntactic forms and procedures `dynamic-wind`, `call-with-input-file`, `call-with-output-file`, `with-output-to-file`, `with-input-from-file`, `delay`, and `force` have been updated to handle multiple values, as have both `time` and `trace`. *Chez Scheme* "cafés" have been updated to return multiple values, and `exit` and the default `exit-handler` now accept multiple values.

Engines have been updated in a non-upwardly compatible fashion to support multiple values: the `complete` procedure passed to an engine is now passed the remaining ticks as its first argument, and the values returned by the engine computation as its remaining arguments. In prior releases, the single value of the engine computation was passed as the first argument, with the ticks remaining passed as the second argument.

1.3. Support for new platforms

Support has been added for Silicon Graphics computers running IRIX Version 5.X, Sun Sparc-based computers running Solaris 2.X, Decsystem MIPS 4400-based computers running Ultrix 4.3A, Motorola MC88000-based systems running System-V, Release 4, Intel 80x86-based systems running NeXT Mach 3.2, BSDI BSD/386 1.1, and Linux.

Full use has been made of the memory mapping and dynamic linking capabilities of Solaris 2.X, IRIX 5.X, and SVR4. Dynamic shared libraries are used to reduce executable size, the run-time linker is used to load foreign code, and saved heaps are shared by all processes that use them (see the note regarding saved heaps below). Foreign code is loaded, using `load-shared-object`, in the form of "shared objects" rather than simple object files. Programs that previously used `provide-foreign-entries` or `load-foreign`, which are not supported under these operating systems, should be rewritten to use `load-shared-object` instead.

1.4. Guardians and weak pairs

Version 5 includes support for two new features supported by the storage management system: *guardians* and *weak pairs*. Guardians allow programs to protect objects from deallocation by the garbage collector and to determine when the objects would otherwise have been deallocated. Weak pairs allow programs to maintain "weak" pointers to objects, pointers that do not save the object from deallocation but which remain valid as long as the object is otherwise accessible in the system.

1.5. Generic ports

A new *generic port* facility is supported in Version 5. This facility allows the programmer to add new types of ports with arbitrary input/output semantics. This facility may be used, for example, to define any of the built-in Common Lisp stream types, *i.e.*, synonym streams, broadcast streams, concatenated streams, two-way streams, echo streams, and string streams. It may also be used to define more exotic ports, such as ports that represent windows on a bit-mapped display or ports that represent processes connected to the current process via pipes or sockets.

Along with support for generic ports, Version 5 includes support for bidirectional ports, *i.e.*, ports that are both input and output ports.

1.6. Saved heaps

Version 5 supports two major enhancements to the saved heap mechanism. First, support for *incremental heaps* has been added. Incremental heaps contain only those portions of the heap that have changed from a previously saved heap. This usually results in considerable space savings, since all of the system code and much of the rest of the base heap remains the same. Second, on systems that support memory mapped files, saved heaps (including incremental heaps) are simply mapped into memory, reducing start-up time to a minimum and allowing processes to share portions of the heap that have not been modified.

1.7. Trace output

Trace output is no longer as "noisy" as in Version 4, and output values are aligned properly with respect to the corresponding call. For nesting levels 10 or greater, a number in brackets is used in place of indentation to signify nesting level. For example:

```
> (trace-let f ((x 15)) (if (= x 0) 1 (* x (f (- x 1)))))
|(f 15)
| (f 14)
| |(f 13)
| | (f 12)
| | |(f 11)
| | | (f 10)
| | | |(f 9)
| | | | (f 8)
| | | | |(f 7)
| | | | | (f 6)
| | | |[10](f 5)
| | | |[11](f 4)
| | | |[12](f 3)
| | | |[13](f 2)
| | | |[14](f 1)
| | | |[15](f 0)
| | | |[15]1
| | | |[14]1
| | | |[13]2
| | | |[12]6
| | | |[11]24
| | | |[10]120
| | | | | 720
| | | | |5040
| | | | 40320
| | | |362880
| | | 3628800
| | |39916800
| | 479001600
| |6227020800
| 87178291200
|1307674368000
1307674368000
>
```

The trace facility now also uses `pretty-print` to print the argument and return values, resulting in more readable output for lines that would have wrapped in previous versions.

A new syntactic form, `trace-define`, is provided. `trace-define` is sometimes more convenient than using `define` with `trace-lambda`, especially when modifying existing code.

1.8. New inspector commands

The interactive inspector now provides an "eval" command that permits the evaluation of arbitrary Scheme expressions within the context of a frame or procedure environment. The values of frame and procedure locations are bound within the evaluated expression to identifiers of the form $\%n$, where n is the location number displayed by the inspector. The values of named locations are also bound to the name.

The interactive inspector also now permits frame and procedure variables considered "assignable" by the compiler to be modified via a `set!` command. The lower-level object inspector similarly permits assignable variables to be modified via a new `set!` message. Assignable variables are generally limited to those for which the user program contains an assignment. Nonassignable variables cannot be modified since permitting such modifications would violate assumptions made by the compiler and run-time system with potentially unsafe or misleading results.

1.9. Foreign procedure interface enhancements

Two new foreign procedure argument/return types, "fixnum" and "single-float," have been added. The "fixnum" type is used to communicate numbers in the fixnum range and is slightly more efficient than the "integer-32" type. The "single-float" type is used to communicate floating point values to or from C routines expecting or returning single floats. Scheme "flonums" (inexact real numbers), represented internally as double floats, are converted to single floats, and single-float return values are converted to Scheme flonums automatically when the single-float specifier is used.

The "string" argument type now accepts `#f` as an input, converting it to the null pointer (0). This makes it consistent with the "string" return type.

Version 5 also includes support for two additional new foreign procedure parameter and return types on the MIPS-based Decsystem/Decstation and Silicon Graphics systems: `foreign-object` and `foreign-pointer`. These parameter types allow arbitrary foreign structures to be passed into and returned from foreign procedures. Foreign objects are represented in Scheme as Scheme strings, and foreign pointers are represented as integers. Support for this feature on non-MIPS platforms is not planned at this time.

1.10. Block I/O

Version 5 includes support for block reads and writes to any port, through the procedures `block-read` and `block-write`. Both procedures accept three arguments: a port p, a string s, and a count n. For reads, the system reads n bytes from p and places them in s; for writes, the system writes n bytes from s to p. Block I/O can be much more efficient than character-by-character I/O. In combination with `file-position` and `file-length`, the block I/O operations allow arbitrary file manipulations to be written directly and efficiently in Scheme.

1.11. Pretty printer control parameters

Three new pretty printer control parameters have been added. The parameter `pretty-initial-indent` is used to tell the pretty-printer when the output starts in some column other than column zero. The parameter `pretty-standard-indent` determines the default indentation for various expressions, such as `let`. The parameter `pretty-maximum-lines` determines how many lines pretty print emits; no limit is imposed when the default setting of `#f` is used.

1.12. Compile-time argument count checks

Significantly more compile-time argument count checks are performed in Version 5. Calls to most locally-bound procedures called within a given top-level expression are checked, and appropriate errors are signaled.

1.13. Rebinding `reset-handler` during initial load

Rebinding `reset-handler` within a file loaded from the command line now has the affect of trapping error resets, just as it does for files loaded once the initial waiter has been started.

1.14. Conversion of flonums to fixnums

An efficient flonum to fixnum conversion procedure, `flonum->fixnum`, has been added. This procedure returns the truncated fixnum equivalent of any flonum whose truncated value is in the fixnum range. `flonum->fixnum` can be significantly more efficient than the alternative of first truncating the flonum, then converting the result to a fixnum using `inexact->exact`.

1.15. New support for primitive references

The `#%id` syntax supported by Version 4 has been expanded to allow two new forms: `#2%id` and `#3%id`. As before, `#%id` is equivalent to `(#primitive id)`; `#2%id` is equivalent to `(#primitive 2 id)` and `#3%id` is equivalent to `(#primitive 3 id)`. The identifier `id` must name a valid primitive. When used as the procedure expression in an application, the `#2%id` syntax causes the application (but not its subexpressions) to be treated as if it were at optimize-level 2, while the `#3%id` syntax causes the application to be treated as if it were at optimize-level 3.

2. Performance Enhancements Since Version 4

2.1. Faster procedure calls

Calls to locally-bound procedures are typically much more time and space efficient in Version 5 than in previous versions. The compiler analyzes the code to determine which local procedures require access to their closures (where free variables are stored), and to determine the entry points for each supported number of actual parameters. Calls to these procedures are made directly to the appropriate entry point, instead of indirectly through the closure as had been done, thereby avoiding memory indirects and argument count checks. No closure is created (at the point of definition) or loaded (at the point of call) for local procedures that do not require them.

In addition, when a list is created for a local procedure with a "dot" interface, the list is created at the point of call, where the length of the list is known, rather than on entry to the call, where more complex code to handle variable numbers of arguments would be needed.

Other improvements have been made in the treatment of both local and global procedure calls, reducing the number of instructions and memory references used in some cases.

2.2. Improved register allocation

The register allocator now makes significantly better use of hardware registers for procedure parameters, local user variables, and compiler temporaries, resulting in a significant reduction in the amount of memory traffic and a substantial increase in performance.

2.3. Improved code generation

The performance of the majority of *Chez Scheme* primitives has been improved at all optimize levels due to improvements mentioned elsewhere, improved code generation strategies, and careful recoding in many instances. Notably improved primitives include `apply`, `assoc`, `assv`, `for-each`, `fxmax`, `fxmin`, `fxremainder`, `length`, `make-string`, `make-vector`, `map`, `member`, `memv`, `remove`, `remv`, `remq`, `string->uninterned-symbol`, `unread-char`, and `write-char` (to console) at all optimize levels; `caar`, `cadr`, ..., `cddddr`, `char=?`, `char<?`, `char<=?`, `char>?`, `char>=?`, `char->integer`, `fx1+`, `fx1-`, `fx*`, `fx/`, `fxeven?`, `fxlogand`, `fxlognot`, `fxlogor`, `fxlogxor`, `fxnegative?`, `fxnonpositive?`, `fxnonnegative?`, `fxpositive?`, `fxquotient`, `fxodd?`, `fxsll`, `fxsra`, `fxsrl`, `fxzero?`, `integer->char`, `set-car!`, `set-cdr!`, `string-length`, `string-ref`, `string-set!`, `vector-length`, `vector-ref`, and `vector-set!` at optimize level 2; and `vector`, `string`, and `list*` at optimize level 0;

2.4. Quicker `compile` for simple expressions

In previous versions, certain simple expressions typed into *Chez Scheme*, loaded from a source file, or passed to `eval` at optimize level 0 under the default value of `current-eval` were interpreted rather than compiled. This results in a significant savings in terms of compile time for simple expressions, noticeable primarily when repeated calls to `eval` are made. This behavior has been extended to a larger class of expressions (nearly all expressions that do not create procedures) and to all optimize levels. A new parameter, `compile-interpret-simple`, may be set to false to force compilation rather than interpretation of these simple expressions at all optimize levels.

2.5. Reduced `gensym` overhead

`gensym` is now about 25 times faster, primarily because no name is generated for the gensym until and unless it is passed as an argument to `symbol->string`, such as when it is printed. Gensym name generation has been improved as well, so that gensym cost is greatly reduced (by about a factor of 3) even if the name is actually generated.

2.6. Increased initial `collect-trip-bytes`

In recognition of the prevalence of larger memory sizes in modern computer systems, the initial value of `collect-trip-bytes` (the number of bytes of heap allocation permitted between collections) has been raised from 2^{18} bytes (256 kilobytes) to 2^{20} bytes (one megabyte). Some programs on some systems may work better with a smaller number; if the Scheme process terminates due to insufficient heap space availability, or if excessive paging is noticed, the value should be set lower. Even on machines with sufficient virtual and physical memory, cache hit rates and therefore overall performance may decrease with larger values, so experimentation is needed to find the optimal amount.

2.7. Improved `foreign-procedure` code generation

In previous versions, foreign procedures always checked their argument types, regardless of optimize level. In Version 5, this checking is disabled at optimize-level 3, reducing both code-size and time overhead. Argument checking remains enabled at optimize levels below 3, along with range checks for “char” and “fixnum” return types.

The volume of code generated for foreign-procedures has been greatly reduced in many cases through the use of more generic error handling procedures and out-of-line code for the more complex conversions, *i.e.*, bignum to int.

2.8. Reduced load time

Load time for both source and object files has been improved, most notably on DEC MIPS-based systems under Ultrix. See Section 3 under “Cache flushing overhead.”

3. Bugs Fixed Since Version 4

3.1. Bug in `quotient` and `remainder`

A bug in `quotient` and `remainder` that caused certain operations involving negative inexact numbers to return incorrect results has been fixed.

3.2. Bug in continuation handling code

A bug in the continuation handling code that could result in memory faults or large amounts of memory being consumed has been fixed.

3.3. Compiler transformation bug

Some malformed expressions that appear similar to named `let` forms that were incorrectly recognized as named `let` forms are now flagged as errors. For example, `((letrec ((x (lambda () 0))) x) 1)` was treated as `((letrec ((x (lambda () 0))) x))` but now properly results in an incorrect number of arguments error.

3.4. Bug in `trace-define`

The expansion for the "defun" syntax of `trace-define` did not work properly; this has been fixed.

3.5. Expanding foreign procedure names

The name field in a `foreign-procedure` form is now expanded by the macro expander.

3.6. Foreign procedure "integer-32" return type

A bug that caused a return value of -80000000_{16} to be converted improperly when the return type of a foreign procedure is integer-32 has been fixed.

3.7. Cache flushing overhead.

A mechanism for reducing the effects of inexplicably long hardware cache flush times on DEC MIPS-based systems under Ultrix has been implemented. This mechanism dramatically reduces load times for both Scheme source and object files containing many expressions. A file containing 100 individual expressions that might have taken 30 seconds to load under Version 4.1 might now take less than 1/2 second. Wrapping entire source files in single `begin` expressions to reduce load time, as advised in some earlier release notes, is no longer needed.

The same mechanism has resulted in noticeable but less dramatic load-time improvements on other platforms.

4. Compatibility Issues

As noted in Section 1 under "Multiple values," the engine interface has changed in a non-upwardly compatible fashion. Code that uses engines must be updated to account for this change.

As noted in Section 1 under "Support for new platforms," `load-shared-object` must now be used on some platforms in place of `load-foreign` and `provide-foreign-entries`.

As noted in Section 1 under "Lexical macros," macros written in expansion-passing style or using `extend-syntax` must be rewritten to use `syntax-case` or `syntax-rules`, although a compatibility package allowing the use of these older macro system will be provided for a limited time. Since the compatibility package disables the new macro system, syntactic forms defined using one of the old macro systems cannot be intermixed within the same top-level expression as syntactic forms defined using the new macro system.

The default "standard indentation" used by the pretty printer is now 1 rather than 2, allowing nested code to be displayed in fewer columns. Most code will not be affected by this change. Code that depends on the old default should set the parameter `pretty-standard-indent` to 2.