NAME

Chez Scheme
Petite Chez Scheme

SYNOPSIS

scheme [*options*] *file* ...
petite [*options*] *file* ...

DESCRIPTION

Chez Scheme is a high-performance implementation of R6RS Scheme with numerous extensions. *Chez Scheme* compiles source expressions *incrementally*, providing the speed of compiled code in an interactive system.

Petite Chez Scheme is a freely distributable interpreted version of *Chez Scheme* that may be used as a run-time environment for *Chez Scheme* applications or as a stand-alone Scheme system. With the exception that the compiler is not present, *Petite Chez Scheme* is 100% compatible with *Chez Scheme*. Interpreted code is fast in *Petite Chez Scheme*, but generally not nearly as fast as compiled code.

Scheme is normally used interactively. The system prompts the user with a right angle bracket (“>”) at the beginning of each input line. Any Scheme expression may be entered. The system evaluates the expression and prints the result. After printing the result, the system prompts again for more input. The user can exit the system by typing Control-D or by using the procedure *exit*.

COMMAND-LINE OPTIONS

Chez Scheme recognizes the following command line options:

-q, --quiet Suppress greeting and prompts.

--script *file* Run *file* as a shell script.

--program *file* Run rnrs program in *file* as a shell script.

--libdirs *dir***:...** Set library directories to *dir*:....

--libexts *ext***:...** Set library extensions to *ext*:....

--compile-imported-libraries
Compile libraries before loading them.

--import-notify Enable import search messages.

--optimize-level 0 | 1 | 2 | 3
Set optimize level to 0, 1, 2, or 3.

--debug-on-exception
On uncaught exception, call debug.

--eedisable Disables the expression editor.

--eehistory off | *file*
Set expression-editor history file or disable restore and save of history.

--revert-interaction-semantics
Use old interaction semantics.

-b *file***, --boot** *file*
Load boot code from *file*.

--verbose Trace boot search process.

--version Print version and exit.

--help Print brief command-line help and exit.

-- Pass all remaining command-line arguments through to Scheme.

The following options are recognized but cause the system to print an error message and exit because saved heaps are not presently supported.

-h *file*, **--heap** *file*

-s[*level***]** *file*, **--saveheap[***level***]** *file*

-c, --compact

WAITERS and CAFES

Interaction of the system with the user is performed by a Scheme program called a *waiter*, running in a program state called a *café*. The waiter merely prompts, reads, evaluates, prints and loops back for more. It is possible to open up a chain of *Chez Scheme* cafés by invoking the *new-cafe* procedure with no arguments. New-cafe is also one of the options when an interrupt occurs. Each café has its own reset and exit procedures. Exiting from one café in the chain returns you to the next one back, and so on, until the entire chain closes and you leave the system altogether. Sometimes it is useful to interrupt a long computation by typing the interrupt character, enter a new café to execute something (perhaps to check a status variable set by computation), and exit the café back to the old computation.

You can tell what level you are at by the number of angle brackets in the prompt, one for level one, two for level two, and so on. Three angle brackets in the prompt means you would have to exit from three cafés to get out of *Chez Scheme*. If you wish to abort from *Chez Scheme* and you are several cafés deep, the procedure *abort* leaves the system directly.

You can exit the system by typing the end-of-file character (normally Control-D) or by using the procedure *exit*. Typing Control-D is equivalent to (exit), (exit (void)), or (exit 0), each of which is considered a "normal exit".

DEBUGGER

Ordinarily, if an exception occurs during interactive use of the system, the default exception handler displays the condition with which the exception was raised, saves it for possibly later use by the debugger, and prints the message ''type (debug) to enter the debugger.'' Once in the debugger, the user has the option of inspecting the raise continuation, i.e., the stack frames of the pending calls. When an exception occurs in a script or top level program, or when the standard input and/or output ports are redirected, the default exception handler does not save the continuation of the exception and does not print the ''type (debug)'' message.

If the parameter debug-on-exception is set to #t, however, the default exception handler directly invokes debug, whether running interactively or not, and even when running a script or top-level program. The "--debug-on-exception" option may be used to set debug-on-exception to #t from the command line, which is particularly useful when debugging scripts or top-level programs run via the "--script" or "--program" options.

None of this applies to exceptions raised with a non-serious (warning) condition, for which the default exception handler simply displays the condition and returns.

KEYBOARD INTERRUPTS

Running programs may be interrupted by typing the interrupt character (normally Control-C). In response, the system enters a break handler, which prompts for input with a "break>" prompt. Several commands may be issued to the break handler, including "e" to exit from the handler and continue, "r" to reset to the current café, "a" to abort *Chez Scheme*, "n" to enter a new café, "i" to inspect the current continuation, and "s" to display statistics about the interrupted program. While typing an expression to the waiter, the interrupt character simply resets to the current café.

EXPRESSION EDITOR

When *Chez Scheme* is used interactively in a shell window, the waiter's "prompt and read" procedure employs an expression editor that permits entry and editing of single- and multiple-line expressions, automatically indents expressions as they are entered, and supports name-completion based on the identifiers defined in the interactive environment. The expression editor also maintains a history of expressions typed during and across sessions and supports tcsh(1)-like history movement and search commands. Other editing commands include simple cursor movement via arrow keys, deletion of characters via backspace and delete, and movement, deletion, and other commands using mostly emacs key bindings.

The expression editor does not run if the TERM environment variable is not set, if the standard input or output files have been redirected, or if the --eedisable command-line option has been used. The history is

saved across sessions, by default, in the file "$HOME/.chezscheme_history". The --eehistory command-line option can be used to specify a different location for the history file or to disable the saving and restoring of the history file.

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

Key bindings for other keys and key sequences initially recognized by the expression editor are given below, organized into groups by function. Some keys or key sequences serve more than one purpose depending upon context. For example, tab is used both for identifier completion and for indentation. Such bindings are shown in each applicable functional group.

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

Newlines, acceptance, exiting, and redisplay:

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

Basic movement and deletion:

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

Line movement and deletion:

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

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

Expression movement and deletion:

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

Entry movement and deletion:

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

Indentation:

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

Identifier completion:

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

If at end of existing identifier, i.e., not one just typed, the first tab re-indents, the second tab inserts identifier completion, and the third shows possible completions.

History movement:

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

To search, enter a prefix or string followed by one of the search key sequences. Follow with additional search key sequences to search further backward or forward in the history. For example, enter “(define” followed by one or more esc-p key sequences to search backward for entries that are definitions, or “(define” followed by one or more esc-P key sequences for entries that contain definitions.

Word and page movement:

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

Inserting saved text:

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

Mark operations:

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

Command repetition:

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

TOP-LEVEL ENVIRONMENT SEMANTICS

Upon startup, the "interaction environment" used to hold the top-level bindings for user-defined variables and other identifiers contains an initial set of bindings, some standard and some specific to *Chez Scheme*. Any initial identifier binding may be replaced by redefining the identifier with a normal top-level definition. For example, the initial binding for *cons* can be replaced with one that performs a ''reverse cons'' as follows.

```
(define cons (lambda (x y) (import scheme) (cons y x)))
```

Code entered into the REPL or loaded from a file prior to this point will still use the original binding for *cons*. If you want it to use the new binding, you must reenter or reload the code. Furthermore, the initial bindings for variables like *cons* are immutable, so you cannot assign one (e.g., via set! or trace) without first defining it. This is a change from how earlier versions of Chez Scheme (Version 7 and before) treated variables (but not keywords). If you prefer the Version 7 semantics, use the "--revert-interaction-semantics" command-line option or type (revert-interaction-semantics) before doing anything else. The new semantics has some advantages over the old semantics, however. Because the initial set of bindings are immutable and have known values, the compiler can generate better code and sometimes produce better compiler warnings about incorrect argument counts. With the new semantics, the system can assume, for example, say, *cons* really is *cons*, check to make sure it receives the expected two arguments at compile time, and generate inline code to allocate the pair. This is not the case with the old semantics, where the compiler had to assume that the value of *cons* could change at any time during a program run.

COMMAND-LINE FILE ARGUMENTS

In the normal mode of operation, the file names on the command line (except for the arguments to the various command-line options) are loaded before *Chez Scheme* begins interacting with the user. Each of the expressions in the loaded files is executed just as if it were typed by the user in response to a prompt. If you wish to load a set of definitions each time, consider setting up a shell script to load the file ".schemerc" from your home directory:

```
scheme ${HOME}/.schemerc $*
```

If you have a substantial number of definitions to load each time, it might be worthwhile to compile the .schemerc file (that is, compile the definitions and name the resulting object file .schemerc).

Typically, a Scheme programmer creates a source file of definitions and other Scheme forms using an editor such as *vi*(1), *emacs*(1), or the SWL (Scheme Widget Library) user interface and loads the file into Scheme to test them. The conventional filename extension for *Chez Scheme* source files is *.ss*. Such a file may be loaded during a session by typing (load "*filename*"), or by specifying the filename on the command line as mentioned above. Any expression that may be typed interactively may be placed in a file to be loaded.

SCHEME SCRIPTS

When the "--script" option is used, the named file is treated as a Scheme shell script, and the script name and remaining command-line arguments are made available via the parameter "command-line". To support executable shell scripts, the system ignores the first line of a loaded script if it begins with #! followed by a space or forward slash. For example, the following script prints its command-line arguments.

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

RNRS TOP-LEVEL PROGRAMS

The "--program" option is like the "--script" option except that the script file is treated as an RNRS top-level program. The following RNRS top-level program prints its command-line arguments, as with the script above.

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

```
(import (rnrs))
(for-each
  (lambda (x) (display x) (newline))
  (cdr (command-line)))
```

"scheme-script" may be used in place of "scheme --program", possibly prefixed by "/usr/bin/env" as suggested in the nonnormative R6RS appendix on running top-level programs as scripts, i.e., the first line of the top-level program may be replaced with the following.

```
#! /usr/bin/env {InstallSchemeScriptName}
```

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

These options set the corresponding *Chez Scheme* parameters library-directories and library-extensions. The values of both parameters are lists of pairs of strings. The first string in each library-directories pair identifies a source-file root directory, and the second identifies the corresponding object-file root directory. Similarly, the first string in each library-extensions pair identifies a source-file extension, and the second identifies the corresponding object-file extension. The full path of a library source or object file consists of the source or object root followed by the components of the library name prefixed by slashes, with the library extension added on the end. For example, for root /usr/lib/scheme, library name (app lib1), and extension .sls, the full path is /usr/lib/scheme/app/lib1.sls.

The format of the arguments to "--libdirs" and "--libexts" is the same: a sequence of substrings separated by a single separator character. The separator character is a colon (:), except under Windows where it is a semi-colon (;). Between single separators, the source and object strings, if both are specified, are separated by two separator characters. If a single separator character appears at the end of the string, the specified pairs are added to the existing list; otherwise, the specified pairs replace the existing list. The parameters are set after all boot files have been loaded.

If multiple "--libdirs" options appear, all but the final one are ignored, and if If multiple "--libexts" options appear, all but the final are ignored. If no "--libdirs" option appears and the CHEZSCHEMELIBDIRS environment variable is set, the string value of CHEZSCHEMELIBDIRS is treated as if it were specified by a "--libdirs" option. Similarly, if no "--libexts" option appears and the CHEZSCHEMELIBEXTS environment variable is set, the string value of CHEZSCHEMELIBEXTS is treated as if it were specified by a "--libexts" option.

The library-directories and library-extensions parameters set by these options are consulted by the expander when it encounters an import for a library that has not previously been defined or loaded. The expander first constructs a partial name from the list of components in the library name, e.g., "a/b" for library (a b). It then searches for the partial name in each pair of root directories, in order, trying each of the source extensions then each of the object extensions in turn before moving onto the next pair of root directories. If the partial name is an absolute pathname, e.g., "~/.myappinit" for a library named (~/.myappinit), only the specified absolute path is searched, first with each source extension, then with each object extension. If the expander finds a source file before it finds an object file, it loads the corresponding object file if the object file exists and is not older than the source file. If this is not the case, and the parameter compile-imported-libraries is set to #t, the expander compiles the library via compile-library. Otherwise, the expander loads the source file. An exception is raised during this process if a source or object file exists but is not readable or if an object file cannot be created.

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

OPTIMIZE LEVELS

The "--optimize-level" option sets the initial value of the *Chez Scheme* optimize-level parameter to 0, 1, 2, or 3. The value is 0 by default.

At optimize-levels 0, 1, and 2, code generated by the compiler is *safe*, i.e., generates full type and bounds checks. At optimize-level 3, code generated by the compiler is *unsafe*, i.e., may omit these checks. Unsafe code is usually faster, but optimize-level 3 should be used only for well-tested code since the absense of type and bounds checks may result in invalid memory references, corruption of the Scheme heap (which may cause seemingly unrelated problems later), system crashes, or other undesirable behaviors.

At optimize levels 2 and 3, the system also assumes that the names of built-in procedures have their original values, even if assigned, in an interaction environment whose semantics have been reverted by the revert-interaction-semantics procedure or because the "--revert-interaction-semantics" command-line option has been supplied. This aspect of the optimize level is irrelevant for code appearing within a library or RNRS top-level program or bindings imported from a module or library.

COMPILING FILES

Chez Scheme compiles source expressions as it sees them. In order to speed loading of a large file, the file may be compiled with the output placed in an object file. (compile-file "foo") compiles the expressions in the file "foo.ss" and places the resulting object code on the file "foo.so". Loading a pre-compiled file is no different from loading the source file, except that loading is faster since compilation is already done.

To compile a program to be run with --program, use compile-program instead of compile-file. compile-program preserves the first line unchanged, if it begins with #! followed by a forward slash or space. Also, while compile-file compresses the resulting object file, compile-program does not do so if the #! line is present, so it can be recognized by the shell's script executor. Any libraries upon which the top-level program depends, other than built-in libraries, must be compiled first via compile-file or compile-library. This can be done manually or by setting the parameter compile-imported-libraries to #t before compiling the program.

To compile a script to be run with --script, use compile-script instead of compile-file. compile-script is like compile-program, but, like compile-file, implements the interactive top-level semantics rather than the RNRS top-level program semantics.

BOOT and HEAP FILES

When *Chez Scheme* is run, it looks for one or more boot files to load. Boot files contain the compiled Scheme code that implements most of the Scheme system, including the interpreter, compiler, and most libraries. Boot files may be specified explicitly on the command line via "-b" options or implicitly. In the simplest case, no "-b" options are given and the necessary boot files are loaded automatically based on the name of the executable. For example, if the executable name is "myapp", the system looks for "myapp.boot" in a set of standard directories. It also looks for and loads any subordinate boot files required by "myapp.boot". Subordinate boot files are also loaded automatically for the first boot file explicitly specified via the command line. When multiple boot files are specified via the command line and boot each file must be listed before those that depend upon it.

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

Ordinarily, the search for boot files is limited to a set of default installation directories, but this may be overridden by setting the environment variable SCHEMEHEAPDIRS. SCHEMEHEAPDIRS should be a colon-separated list of directories, listed in the order in which they should be searched. Within each directory, the two-character escape sequence "%v" is replaced by the current version, and the two-character escape sequence "%m" is replaced by the machine type. A percent followed by any other character is replaced by the second character; in particular, "%%" is replaced by "%", and "%:" is replaced by ":". If SCHEMEHEAPDIRS ends in a non-escaped colon, the default directories are searched after those in SCHEMEHEAPDIRS; otherwise, only those listed in SCHEMEHEAPDIRS are searched. Under Windows, semi-colons are used in place of colons.

Boot files consist of a header followed by ordinary compiled code and may be created with make-boot-file. For example,

```
(make-boot-file "myapp.boot" '("petite")
  "myapp1.so" "myapp2.so")
```

creates a boot file containing the code from myapp1.so and myapp2.so with a header identifying petite.boot as a boot file upon which the new boot file depends. Source files can be provided as well and are compiled on-the-fly by make-boot-header.

Multiple alternatives for the boot file upon which the new boot file depends can be listed, e.g.:

```
(make-boot-file "myapp.boot" '("petite" "scheme")
  "myapp1.so" "myapp2.so")
```

When possible, both “scheme“ and “petite“ should be specified when creating a boot file for an application, as shown above, so that the application can run in either *Petite Chez Scheme* or *Chez Scheme*. If the application requires the use of the compiler, just “scheme“ should be specified.

If the new boot file is to be a base boot file, i.e., one that does not depend on another boot file, petite.boot (or some other boot file created from petite.boot) should be listed first among the input files.

```
(make-boot-file "myapp.boot" '() "petite.boot"
  "myapp1.so" "myapp2.so")
```

DOCUMENTATION

Complete documentation for *Chez Scheme* is available in two parts: *The Scheme Programming Language, 4th Edition*, and *The Chez Scheme Version 8 User's Guide*. Both documents are available electronically at *www.scheme.com* as well as in printed form.

Several example Scheme programs, ranging from a simple factorial procedure to a somewhat complex unification algorithm, are in the examples directory (see FILES below). Looking at and trying out example programs is a good way to start learning Scheme.

ENVIRONMENT

The environment variable **SCHEMEHEAPDIRS** (see above) may be set to a colon-separated (semi-colon under Windows) list of directories in which to search for boot files.

FILES

/usr/bin/scheme	executable file
/usr/bin/petite	executable file
/usr/bin/{InstallSchemeScriptName}	executable file
/usr/lib/csv8.0/lib	example program library
/usr/lib/csv8.0/i3le	boot and include files

SEE ALSO

R. Kent Dybvig, *The Scheme Programming Language, 4th Edition*, MIT Press (2009), http://www.scheme.com/tspl4/.

R. Kent Dybvig, *Chez Scheme Version 8 User's Guide*, Cadence Research Systems (2010), http://www.scheme.com/csug8/.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, eds., “Revised^6 Report on the Algorithmic Language Scheme,” (2007), http://www.r6rs.org/.

Daniel P. Friedman and Matthias Felleisen, *The Little Schemer*, fourth edition, MIT Press (1996).

Harold Abelson and Gerald J. Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs, Second Edition*, MIT press (1996).

AUTHOR

Cadence Research Systems, *http://www.scheme.com*