

NAME

chez Scheme
Petite Chez Scheme

SYNOPSIS

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

DESCRIPTION

chez Scheme is a high-performance implementation of ANSI 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 runtime 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.

chez Scheme recognizes the following command line options:

- b file, --boot file** Load boot code from *file*.
- c, --compact** Toggle flag for heap compaction on heap save.
- h file, --heap file** Load heap from *file*.
- s[level] file, --saveheap[level] file** Save heap on normal exit in *file*.
- script file** Run *file* as a shell script.
- q, --quiet** Suppress greeting and prompts.
- help** Print brief command-line help and exit.
- verbose** Trace boot/heap search process
- version** Print version and exit.
- Pass all remaining command-line arguments through to Scheme.

The “-c”, “-h”, “-s”, “--script”, and “--verbose” options are described in detail below. The others should be self-explanatory.

In the normal mode of operation, the file names on the command line (except for the arguments to the “-b”, “-h”, and “-s” switches) 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) or to save the definitions in a heap file (see below).

Typically, a Scheme programmer creates a source file of definitions and other Scheme forms using an editor such as *vi*(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.

When the "--script" option is used, the named file is treated as a Scheme shell script, and the remaining command-line arguments are made available via the parameter "command-line-arguments". For example, the following script prints its command-line arguments.

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

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.

When *Chez Scheme* is run, it looks for one or more boot files and/or one or more heap files to load. Boot files contain the compiled Scheme code that implements most of the Scheme system, including the interpreter, compiler, and most libraries. Heap files contain prebuilt heap images, i.e., saved heap images into which the boot code has already been loaded. Heap and boot files may be specified explicitly on the command line via "-b" and "-h" options or implicitly. In the simplest case, no "-b" and "-h" options are given and the necessary heap or 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.heap" in a set of standard directories, then for "myapp.boot". It also looks for and loads any subordinate heaps or boot files required by "myapp.heap" or "myapp.boot". Subordinate heap and boot files are also loaded automatically for the first boot file and any heap files explicitly specified via the command line. When boot and heap files are specified via the command line, all heap files must come before all boot files, and each file must be listed before those that depend upon it.

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

Ordinarily, the search for heap and 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 ordinary compiled code and may be created by concatenating a boot header onto the compiled code for one or more source files, e.g.:

```
% cat myapp.header myapp1.so myapp2.so > myapp.boot
```

Boot headers are created with the Scheme procedure "make-boot-header", which takes an output file name and the names of one or more alternatives for the boot file upon which the new boot file immediately depends, e.g.:

```
% echo '(make-boot-header "myapp.header" "scheme.boot" "petite.boot")' | scheme
```

In ordinary usage, one or both of "scheme" and "petite" should be specified, as shown above, when creating the base boot file for an application. If the application requires the use of the compiler, just "scheme" should be specified.

Heap files are created via the “-s” option. If the “-s” option is present and *Chez Scheme* exits normally (see *exit*, below), it will save the heap on the specified file. The heap contains the entire state of the system, so that any procedures or other objects created during a session are retained. Saved heaps are typically used to create fast-loading Scheme-based applications, but they also provide an alternative to the `.schemerc` file for customizing the scheme system and allow suspension of a session to return to at a later time. Due to differences in memory address allocation across installations of an operating system, heaps should always be built on the target system, i.e., the system upon which the application will run. This can often be done transparently to the end user via an installation program. Heaps may need to be rebuilt from time to time as the operating environment changes. For this reason, boot files are preferred if the somewhat slower start-up time is not an issue.

The heap level is determined by the *level* argument. A level zero heap contains the entire heap, a level one heap contains only those portions that differ from the corresponding level zero heap, and so on. A level one heap may be created in a session in which a level zero heap has been loaded, and the level one heap can be loaded with an additional “-h” option:

```
scheme -h ${HOME}/.heap -s1 ${HOME}/.heap.1 $*
scheme -h ${HOME}/.heap -h ${HOME}/.heap.1 $*
```

If no level follows the “-s” switch, the level defaults to the level of the highest level heap loaded or zero if no heaps have been loaded. If the special level “+” follows the “-s” switch, the level is one level higher than the highest loaded heap or zero if no heaps have been loaded. Thus, the following three lines create level zero, one, one, and two heaps.

```
SCHEMEHEAPDIRS="."; export SCHEMEHEAPDIRS
echo | scheme -b petite.boot -s heap.0
echo | scheme -h heap.0 -s+ heap.1
echo | scheme -h heap.1 -s heap.1
echo | scheme -h heap.1 -s+ heap.2
```

It may be useful to use a shell script that always loads from and saves to a heap file in the home directory.

```
scheme -h ${HOME}/.heap -s ${HOME}/.heap $*
```

The file `${HOME}/.heap` must be created prior to the first time this script is used. You should be careful to retain the source for important definitions, since future releases of *Chez Scheme* will not be compatible with existing heaps or compiled object files.

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

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”. If the “-s” option is present (see above), a normal exit causes the system to save the heap in the given file.

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.

Running programs may be interrupted by typing the interrupt character (normally DEL, BREAK or Control-C). In response, the system enters a debug handler, which prompts for input with a “debug>” prompt. Several commands may be issued to the debug handler, including “e” to exit from the handler and continue, “r” to reset to the current café, “a” to abort *Chez Scheme*, “h” 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.

When an error occurs, the system prints an error message and resets. Typing (debug) after an error occurs places you into the debug handler, where you can inspect the current continuation (control stack) to help determine the cause of the problem.

Complete documentation for *Chez Scheme* is available in two parts: *The Scheme Programming Language, Third Edition*, and *The Chez Scheme Version 7 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 and heap files.

FILES

/usr/bin/scheme	executable file
/usr/bin/petite	executable file
/usr/lib/csv7.0/lib	example program library
/usr/lib/csv7.0/i3le	boot, heap, and include files

SEE ALSO

R. Kent Dybvig, *The Scheme Programming Language, Third Edition*, MIT Press (2003).

R. Kent Dybvig, *Chez Scheme Version 7 User’s Guide*, Cadence Research Systems (2005).

IEEE Computer Society, *IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990 (1991).

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).

Richard Kelsey, Will Clinger and Jonathan Rees, eds., “Revised⁵ Report on the Algorithmic Language Scheme,” *Higher Order and Symbolic Computation* 11, 1, 1999.

AUTHOR

Cadence Research Systems, <http://www.scheme.com>