Table Of Contents

The CHICKEN User's Manual

This is the manual for Chicken Scheme, version 4.6.0

Getting started

CHICKEN is a compiler that translates Scheme source files into C, which in turn can be fed to a C compiler to generate a standalone executable. An interpreter is also available and can be used as a scripting environment or for testing programs before compilation.

This chapter is designed to get you started with CHICKEN programming, describing what it is and what it will do for you, and covering basic use of the system. With almost everything discussed here, there is more to the story, which the remainder of the manual reveals. Here, we only cover enough to get you started. Nonetheless, someone who knows Scheme already should be able to use this chapter as the basis for writing and running small CHICKEN programs.

Scheme

Scheme is a member of the Lisp family of languages, of which Common Lisp and Emacs Lisp are the other two widely-known members. As with Lisp dialects, Scheme features

- a wide variety of programming paradigms, including imperative, functional, and object-oriented
- a very simple syntax, based upon nested parenthesization
- the ability to extend the language in meaningful and useful ways

In contrast to Common Lisp, Scheme is very minimal, and tries to include only those features absolutely necessary in programming. In contrast to Emacs Lisp, Scheme is not anchored into any one program (Emacs), and has a somewhat more modern language design.

Scheme is defined in a document called *The Revised^5 Report on the Algorithmic Language Scheme*, or *R5RS* for short. (Yes, it really has been revised five times, so an expanded version of its name would be *The Revised Revised Revised Revised Revised Report*.) A newer report, *R6RS*, was released in 2007, but this report has attracted considerable controversy, and not all Scheme implementations will be made compliant with it. CHICKEN essentially complies with R5RS.

Even though Scheme is consciously minimalist, it is recognized that a language must be more than a minimal core in order to be useful. Accordingly, the Scheme community uses a process known as `Scheme Requests For Implementation' (SRFI, pronounced `SUR-fee') to define new language features. A typical Scheme system therefore complies with one of the Scheme reports plus some or all of the accepted SRFIs.

A good starting point for Scheme knowledge is http://www.schemers.org. There you will find the defining reports, FAQs, lists of useful books and other resources, and the SRFIs.

The CHICKEN community is at present developing tutorials for programmers who are new to Scheme but experienced with Python, Ruby, or other languages. These can be found on the CHICKEN wiki.

CHICKEN

CHICKEN is an implementation of Scheme that has many advantages.

CHICKEN Scheme combines an optimising compiler with a reasonably fast interpreter. It supports almost all of R5RS and the important SRFIs. The compiler generates portable C code that supports tail recursion, first-class continuations, and lightweight threads, and the interface to and from C libraries is flexible, efficient, and easy to use. There are hundreds of contributed CHICKEN libraries that make the programmer's task easier. The interpreter allows interactive use, fast prototyping, debugging, and scripting. The active and helpful CHICKEN community fixes bugs and provides support. Extensive documentation is supplied.

CHICKEN was developed by Felix L. Winkelmann over the period from 2000 through 2007. In early 2008, Felix asked the community to take over the responsibility of developing and maintaining the system, though he still takes a strong interest in it, and participates actively.

CHICKEN includes

- a Scheme interpreter that supports almost all of R5RS Scheme, with only a few relatively minor omissions, and with many extensions
- a compatible compiler whose target is C, thus making porting to new machines and architectures relatively straightforward
  - the C support allows Scheme code to include `embedded' C code, thus making it relatively easy to invoke host OS or library functions
- a framework for language extensions, library modules that broaden the functionality of the system

This package is distributed under the **BSD license** and as such is free to use and modify.

Scheme cognoscenti will appreciate the method of compilation and the design of the runtime-system, which follow closely Henry Baker's CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. paper and expose a number of interesting properties.

- Consing (creation of data on the heap) is relatively inexpensive, because a generational garbage collection scheme is used, in which short-lived data structures are reclaimed extremely quickly.
- Moreover, `call-with-current-continuation` is practically for free and CHICKEN does not suffer under any performance penalties if first-class continuations are used in complex ways.

The generated C code is fully tail-recursive.

Some of the features supported by CHICKEN:

- SRFIs 0, 1, 2, 4, 6-19, 23, 25-31, 37-40, 42, 43, 45, 47, 55, 57, 60-63, 66, 69, 72, 78, 85, 95 and 98.
- Lightweight threads based on first-class continuations
- Pattern matching with Andrew Wright's `match` package
- Record structures
- Extended comment- and string-literal syntaxes
- Libraries for regular expressions, string handling
- UNIX system calls and extended data structures
- Create interpreted or compiled shell scripts written in Scheme for UNIX or Windows
- Compiled C files can be easily distributed
- Allows the creation of fully self-contained statically linked executables
- On systems that support it, compiled code can be loaded dynamically
- Built-in support for cross-compilation and deployment

CHICKEN has been used in many environments ranging from embedded systems through desktop machines to large-scale server deployments. The number of language extensions, or **eggs**, is constantly growing.

- extended language features
- development tools, such as documentation generators, debugging, and automated testing libraries
- interfaces to other languages such as Java, Python, and Objective-C
- interfaces to database systems, GUIs, and other large-scale libraries,
- network applications, such as servers and clients for ftp, smtp/pop3, irc, and http
- web servers and related tools, including URL parsing, HTML generation, AJAX, and HTTP session management
- data formats, including XML, JSON, and Unicode support

CHICKEN is supported by SWIG (Simplified Wrapper and Interface Generator), a tool that produces quick-and-dirty interface modules for C libraries (http://www.swig.org).

This chapter provides you with an overview of the entire system, with enough information to get started writing and running small Scheme programs.

CHICKEN repositories, websites, and community

The master CHICKEN website is http://www.call-with-current-continuation.org. Here you can find basic information about CHICKEN, downloads, and pointers to other key resources.

The CHICKEN wiki (http://wiki.call-cc.org) contains the most current version of the User's manual, along with various tutorials and other useful documents. The list of eggs is at http://wiki.call-cc.org/chicken-projects/egg-index-4.html#category-list.

A very useful search facility for questions about CHICKEN is found at http://chickadee.call-cc.org. The CHICKEN issue tracker is at http://bugs.call-cc.org.

The CHICKEN community has two major mailing lists. If you are a CHICKEN user, `chicken-users` (http://lists.nongnu.org/mailman/listinfo/chicken-users) will be of interest. The crew working on the CHICKEN system itself uses the very low-volume `chicken-hackers` list (http://lists.nongnu.org/mailman/listinfo/chicken-hackers) for communication.

Installing CHICKEN

CHICKEN is available in source form (C) which can be built on several platforms. Refer to the `README` file in the distribution for instructions on installing it on your system.

Because it compiles to C, CHICKEN requires that a C compiler be installed on your system. (If you're not writing embedded C code, you can pretty much ignore the C compiler once you have installed it.)

- On a Linux system, the GNU Compiler Collection (`gcc`) should be installed as part of the basic operating system, or should be available through the package management system (e.g., APT, Synaptic, RPM, or Yum, depending upon your Linux distribution).
- On Macintosh OS X, you will need the XCode tools, which are shipped on the OS X DVD with recent versions of the operating system.
- On Windows, you have three choices.
  - Cygwin (http://sources.redhat.com/cygwin) provides a relatively full-featured Unix environment for Windows. CHICKEN works substantially the same in Cygwin and Unix.
  - The GNU Compiler Collection has been ported to Windows, in the MinGW system (http://mingw.sourceforge.net). Unlike Cygwin, executables produced with MinGW do not need the Cygwin DLLs in order to run. MSys is a companion package to MinGW; it provides a minimum Unix-style development/build environment, again ported from free software.
    - You can build CHICKEN either with MinGW alone or with MinGW plus MSYS. Both approaches produce a CHICKEN built against the mingw headers and import libraries. The only difference is the environment where you actually run make. `Makefile.mingw` is can be used in `cmd.exe` with the version of make that comes with mingw. `Makefile.mingw-msys` uses unix commands such as `cp` and `rm`. The end product is the same.

Refer to the `README` file for the version you're installing for more information on the installation process.

Alternatively, third party packages in binary format are available. Se http://wiki.call-cc.org/platforms for information about how to obtain them.

Development environments

The simplest development environment is a text editor and terminal window (Windows: Command Prompt, OSX: Terminal, Linux/Unix: xterm) for using the interpreter and/or calling the compiler. If you install the `readline` egg, you have all the benefits of command history in the interpreter, Emacs or vi-compatible line editing, and customization.

You will need a text editor that knows Scheme; it's just too painful with editors that don't do parenthesis matching and proper indentation. Some editors allow you to execute Scheme code directly in the editor. This makes programming very interactive: you can type in a function and then try it right away. This feature is very highly recommended.

As programmers have very specific tastes about editors, the editors listed here are shown in alphabetic order. We aren't about to tell you which editor to use, and there may be editors not shown here that might satisfy your needs. We would be very interested in reports of other editors that have been used with CHICKEN, especially those that support interactive evaluation of forms during editing. Pointers to these (and to any editor customization files appropriate) should be put on the CHICKEN wiki, and will likely be added to future editions of this manual. (We have had a request for editors that support proportional fonts, in particular.)

- Emacs (http://www.gnu.org/software/emacs) is an extensible, customizable, self-documenting editor

available for Linux/Unix, Macintosh, and Windows systems; See /emacs for more information about the available options.
- Epsilon (http://www.lugaru.com) is a commercial (proprietary) text editor whose design was inspired by Emacs. Although Scheme support isn't provided, a Lisp mode is available on Lugaru's FTP site, and could with some work be made to duplicate the Emacs support.
- SciTE (http://scintilla.sourceforge.net/SciTE.html), unlike Emacs or Vim, follows typical graphical UI design conventions and control-key mappings, and for simple tasks is as familiar and easy to use as Notepad, KEdit, TeachText etc. However it has many programming features such as multiple open files, syntax highlighting for a large number of languages (including Lisps), matching of brackets, ability to fold sections of code based on the matched brackets, column selections, comment/uncomment, and the ability to run commands in the same directory as the current file (such as make, grep, etc.) SciTE is written with the GTK toolkit and is portable to any GTK platform, including Windows, Linux and MacOS. It uses the Scintilla text-editing component, which lends itself well to embedding within other IDEs and graphical toolkits. It does not have any other Scheme-specific features, but being open-source and modular, features like auto-formatting of S-expressions could be added. The syntax highlighting can be configured to use different fonts for different types of syntax, including proportional fonts.
- Vim (http://www.vim.org) is a highly configurable text editor built to enable efficient and fast text editing. It is an improved version of the vi editor distributed with most UNIX systems. Vim comes with generic Lisp (and therefore Scheme) editing capabilities out of the box. A few tips on using Vim with CHICKEN can be found at http://cybertiggyr.com/gene/15-vim/.

In the rest of this chapter, we'll assume that you are using an editor of your choice and a regular terminal window for executing your CHICKEN code.

The Read-Eval-Print loop

To invoke the CHICKEN interpreter, you use the `csi` command.

```
$ csi

CHICKEN
(c)2008-2010 The Chicken Team
(c)2000-2007 Felix L. Winkelmann
Version 4.6.0
macosx-unix-gnu-x86 [ manyargs dload ptables ]

#;1>
```

This brings up a brief banner, and then the prompt. You can use this pretty much like any other Scheme system, e.g.,

```
#;1> (define (twice f) (lambda (x) (f (f x))))
#;2> ((twice (lambda (n) (* n 10))) 3)
300
```

Suppose we have already created a file `fact.scm` containing a function definition.

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

We can now load this file and try out the function.

```
#;3> (load "fact.scm")
; loading fact.scm ...
#;4> (fact 3)
6
```

The **read-eval-print loop** (**REPL**) is the component of the Scheme system that *reads* a Scheme expression, *eval*uates it, and *prints* out the result. The REPL's prompt can be customized (see the Using the interpreter) but the default prompt, showing the number of the form, is quite convenient.

The REPL also supports debugging commands: input lines beginning with a `,` (comma) are treated as special commands. (See the full list.)

Scripts

You can use the interpreter to run a Scheme program from the command line. For the following example we create a program that does a quick search-and-replace on an input file; the arguments are a regular expression and a replacement string. First create a file to hold the "data" called *quickrep.dat* with your favorite editor holding these lines:

```
xyzabcghi
abxawxcgh
foonly
```

Next create the scheme code in a file called *quickrep.scm* with the following little program:

```
(use irregex) ; irregex, the regular expression library, is one of the
              ; libraries included with CHICKEN.

(define (process-line line re rplc)
  (string-substitute re rplc line 'all))

(define (quickrep re rplc)
  (let ((line (read-line)))
    (if (not (eof-object? line))
        (begin
          (display (process-line line re rplc))
          (newline)
          (quickrep re rplc)))))

;;; Does a lousy job of error checking!
(define (main args)
  (quickrep (regexp (car args)) (cadr args)))
```

To run it enter this in your shell:

```
$ csi -ss quickrep.scm <quickrep.dat 'a.*c' A
xyzAghi
Agh
foonly
```

The `-ss` option sets several options that work smoothly together to execute a script. You can make the command directly executable from the shell by inserting a `shebang line' at the beginning of the program.

The `-ss` option arranges to call a procedure named `main`, with the command line arguments, packed in a list, as its arguments. (There are a number of ways this program could be made more idiomatic CHICKEN Scheme, see the rest of the manual for details.)

The compiler

There are several reasons you might want to compile your code.

- Compiled code executes substantially faster than interpreted code.
- You might want to deploy an application onto machines where the users aren't expected to have CHICKEN installed: compiled applications can be self-contained.

The CHICKEN compiler is provided as the command `chicken`, but in almost all cases, you will want to use the `csc` command instead. `csc` is a convenient driver that automates compiling Scheme programs into C, compiling C code into object code, and linking the results into an executable file. (Note: in a Windows environment with Visual Studio, you may find that `csc` refers to Microsoft's C# compiler. There are a number of ways of sorting this out, of which the simplest is to rename one of the two tools, and/or to organize your `PATH` according to the task at hand.)

Compiled code can be intermixed with interpreted code on systems that support dynamic loading, which includes modern versions of *BSD, Linux, Mac OS X, Solaris, and Windows.

We can compile our factorial function, producing a file named `fact.so` (`shared object' in Linux-ese, the same file type is used in OS X and Windows, rather than `dylib` or `dll`, respectively).

```
chicken$ csc -dynamic fact.scm
chicken$ csi -quiet
#;1> (load "fact.so")
; loading fact.so ...
#;2> (fact 6)
720
```

On any system, we can just compile a program directly into an executable. Here's a program that tells you whether its argument is a palindrome.

```
(define (palindrome? x)
  (define (check left right)
    (if (>= left right)
        #t
        (and (char=? (string-ref x left) (string-ref x right))
             (check (add1 left) (sub1 right)))))
  (check 0 (sub1 (string-length x))))
(let ((arg (car (command-line-arguments))))
  (display
   (string-append arg
                  (if (palindrome? arg)
                      " is a palindrome\n"
                      " isn't a palindrome\n"))))
```

We can compile this program using `csc`, creating an executable named `palindrome`.

```
$ csc -o palindrome palindrome.scm
$ ./palindrome level
level is a palindrome
$ ./palindrome liver
liver isn't a palindrome
```

CHICKEN supports separate compilation, using some extensions to Scheme. Let's divide our palindrome program into a library module (`pal-proc.scm`) and a client module (`pal-user.scm`).

Here's the external library. We `declare` that `pal-proc` is a `unit', which is the basis of separately-compiled modules in CHICKEN. (Units deal with separate compilation, but don't involve separated namespaces; namespaced module systems are available as eggs.)

```
;;; Library pal-proc.scm
(declare (unit pal-proc))
(define (palindrome? x)
  (define (check left right)
    (if (>= left right)
        #t
        (and (char=? (string-ref x left) (string-ref x right))
             (check (add1 left) (sub1 right)))))
  (check 0 (sub1 (string-length x))))
```

Next we have some client code that `uses' this separately-compiled module.

```
;;; Client pal-user.scm
```

```
(declare (uses pal-proc))
(let ((arg (car (command-line-arguments))))
  (display
   (string-append arg
                  (if (palindrome? arg)
                      " is a palindrome\n"
                      " isn't a palindrome\n"))))
```

Now we can compile and link everything together. (We show the compile and link operations separately, but they can of course be combined into one command.)

```
$ csc -c pal-proc.scm
$ csc -c pal-user.scm
$ csc -o pal-separate pal-proc.o pal-user.o
$ ./pal-separate level
level is a palindrome
```

Installing an egg

Installing eggs is quite straightforward on systems that support dynamic loading (again, that would include *BSD, Linux, Mac OS X, Solaris, and Windows). The command `chicken-install` will fetch an egg from the master CHICKEN repository, and install it on your local system.

In this example, we install the `uri-common` egg, for parsing Uniform Resource Identifiers. The installation produces a lot of output, which we have edited for space.

```
$ chicken-install uri-common

retrieving ...
resolving alias `kitten-technologies' to: http://chicken.kitten-technologies.co.uk/
connecting to host "chicken.kitten-technologies.co.uk", port 80 ...
requesting "/henrietta.cgi?name=uri-common&mode=default" ...
reading response ...
[...]
/usr/bin/csc -feature compiling-extension -setup-mode    -s -O2 uri-common.scm -j u
/usr/bin/csc -feature compiling-extension -setup-mode    -s -O2 uri-common.import.s
cp -r uri-common.so /usr/lib/chicken/5/uri-common.so
chmod a+r /usr/lib/chicken/5/uri-common.so
cp -r uri-common.import.so /usr/lib/chicken/5/uri-common.import.so
chmod a+r /usr/lib/chicken/5/uri-common.import.so
chmod a+r /usr/lib/chicken/5/uri-common.setup-info
```

`chicken-install` connects to a mirror of the egg repository and retrieves the egg contents. If the egg has any uninstalled dependencies, it recursively installs them. Then it builds the egg code and installs the resulting extension into the local CHICKEN repository.

Now we can use our new egg.

```
#;1> (use uri-common)
; loading /usr/lib/chicken/5/uri-common.import.so ...
; [... other loaded files omitted for clarity ...]

#;2> (uri-host (uri-reference "http://www.foobar.org/blah"))
"www.foobar.org"
```

Accessing C libraries

Because CHICKEN compiles to C, and because a foreign function interface is built into the compiler, interfacing to a C library is quite straightforward. This means that nearly any facility available on the host system is accessible from CHICKEN, with more or less work.

Let's create a simple C library, to demonstrate how this works. Here we have a function that will compute and return the **n**th Fibonacci number. (This isn't a particularly good use of C here, because we could write this function just as easily in Scheme, but a real example would take far too much space here.)

```
/* fib.c */
int fib(int n) {
  int prev = 0, curr = 1;
  int next;
  int i;
  for (i = 0; i < n; i++) {
    next = prev + curr;
    prev = curr;
    curr = next;
  }
  return curr;
}
```

Now we can call this function from CHICKEN.

```
;;; fib-user.scm
#>
  extern int fib(int n);
<#
(define xfib (foreign-lambda int "fib" int))
(do ((i 0 (+ i 1))) ((> i 10))
  (printf "~A " (xfib i)))
(newline)
```

The syntax `#>...<#` allows you to include literal C (typically external declarations) in your CHICKEN code. We access `fib` by defining a `foreign-lambda` for it, in this case saying that the function takes one integer argument (the `int` after the function name), and that it returns an integer result (the `int` before.) Now we can invoke `xfib` as though it were an ordinary Scheme function.

```
$ gcc -c fib.c
$ csc -o fib-user fib.o fib-user.scm
$ ./fib-user
0 1 1 2 3 5 8 13 21 34 55
```

Those who are interfacing to substantial C libraries should consider using the bind egg.

Back to The User's Manual

Next: Basic mode of operation

Basic mode of operation

The compiler translates Scheme source code into fairly portable C that can be compiled and linked with most available C compilers. CHICKEN supports the generation of executables and libraries, linked either statically or dynamically. Compiled Scheme code can be loaded dynamically, or can be embedded in applications written in other languages. Separate compilation of modules is fully supported.

The most portable way of creating separately linkable entities is supported by so-called *unit*s. A unit is a single compiled object module that contains a number of toplevel expressions that are executed either when the unit is the *main* unit or if the unit is *used*. To use a unit, the unit has to be *declare*ed as used, like this:

```
(declare (uses UNITNAME))
```

The toplevel expressions of used units are executed in the order in which the units appear in the `uses` declaration. Units may be used multiple times and `uses` declarations may be circular (the unit is initialized at most once). To compile a file as a unit, add a `unit` declaration:

```
(declare (unit UNITNAME))
```

When compiling different object modules, make sure to have one main unit. This unit is called initially and initializes all used units before executing its toplevel expressions. The main-unit has no `unit` declaration.

Another method of using definitions in separate source files is to *include* them. This simply inserts the code in a given file into the current file:

```
(include "FILENAME")
```

Macro definitions are only available when processed by `include` or `import`. Macro definitions in separate units are not available, since they are defined at compile time, i.e the time when that other unit was compiled (macros can optionally be available at runtime, see `define-syntax` in Substitution forms and macros).

On platforms that support dynamic loading of compiled code (Windows, most ELF based systems like Linux or BSD, MacOS X, and others) code can be compiled into a shared object `.dll`, `.so`, `.dylib`) and loaded dynamically into a running application.

Previous: Getting started

Next: Using the compiler

Using the compiler

The `csc` compiler driver provides a convenient interface to the basic Scheme-to-C translator (`chicken`) and takes care for compiling and linking the generated C files into executable code. Enter

```
csc -help
```

on the command line for a list of options.

Compiler command line format

```
csc FILENAME-OR-OPTION
```

`FILENAME` is the pathname of the source file that is to be compiled. A filename argument of `-` specifies that the source text should be read from standard input.

Basic command-line options

-analyze-only
Stop compilation after first analysis pass.

-block
Enable block-compilation. When this option is specified, the compiler assumes that global variables are not modified outside this compilation-unit. Specifically, toplevel bindings are not seen by `eval` and unused toplevel bindings are removed.

-case-insensitive
Enables the reader to read symbols case insensitive. The default is to read case sensitive (in violation of R5RS). This option registers the `case-insensitive` feature identifier.

-check-syntax
Aborts compilation process after macro-expansion and syntax checks.

-consult-inline-file FILENAME
load file with definitions for cross-module inlining generated by a previous compiloer invocation via `-emit-inline-file`. Implies `-inline`.

-debug MODES

Enables one or more compiler debugging modes. `MODES` is a string of characters that select debugging information about the compiler that will be printed to standard output.

```
    t          show time needed for compilation
    b          show breakdown of time needed for each compiler pass
    o          show performed optimizations
    r          show invocation parameters
    s          show program-size information and other statistics
    a          show node-matching during simplification
    p          show execution of compiler sub-passes
    l          show lambda-lifting information
    m          show GC statistics during compilation
    n          print the line-number database
    c          print every expression before macro-expansion
    u          lists all unassigned global variable references
    d          lists all assigned global variables
    x          display information about experimental features
    D          when printing nodes, use node-tree output
    N          show the real-name mapping table
    0          show database before lambda-lifting pass
    S          show applications of compiler syntax
    T          show expressions after converting to node tree
    L          show expressions after lambda-lifting
    U          show expressions after unboxing
    M          show syntax-/runtime-requirements
    1          show source expressions
    2          show canonicalized expressions
    3          show expressions converted into CPS
    4          show database after each analysis pass
    5          show expressions after each optimization pass
    6          show expressions after each inlining pass
    7          show expressions after complete optimization
    8          show database after final analysis
    9          show expressions after closure conversion
```

-debug-level LEVEL
Selects amount of debug-information. `LEVEL` should be an integer.

```
    -debug-level 0             is equivalent to -no-trace -no-lambda-info
    -debug-level 1             is equivalent to -no-trace
    -debug-level 2             is equivalent to -scrutinize
```

-disable-interrupts
Equivalent to the `(disable-interrupts)` declaration. No interrupt-checks are generated for compiled programs.

-disable-stack-overflow-checks
Disables detection of stack overflows. This is equivalent to running the compiled executable with the `-:o` runtime option.

-dynamic
This option should be used when compiling files intended to be loaded dynamically into a running Scheme program.

-epilogue FILENAME
Includes the file named `FILENAME` at the end of the compiled source file. The include-path is not searched. This option may be given multiple times.

-emit-all-import-libraries
emit import libraries for all modules defined in the current compulation unit (see also: `-emit-import-library`).

-emit-external-prototypes-first
Emit prototypes for callbacks defined with `define-external` before any other foreign declarations. This is sometimes useful, when C/C++ code embedded into the a Scheme program has to access the callbacks. By default the prototypes are emitted after foreign declarations.

-emit-import-library MODULE
Specifies that an import library named `MODULE.import.scm` for the named module should be generated (equivalent to using the `emit-import-library` declaration).

-emit-inline-file FILENAME
Write procedures that can be globally inlined in internal form to `FILENAME`, if global inlining is enabled. Implies `-inline -local`. If the inline-file would be empty (because no procedure would be inlinable) no file is generated and any existing inline-file with that name is deleted.

-explicit-use
Disables automatic use of the units `library, eval` and `extras`. Use this option if compiling a library unit instead of an application unit.

-extend FILENAME
Loads a Scheme source file or compiled Scheme program (on systems that support it) before compilation commences. This feature can be used to extend the compiler. This option may be given multiple times. The file is also searched in the current include path and in the extension-repository.

-feature SYMBOL
Registers `SYMBOL` to be a valid feature identifier for `cond-expand`. Multiple symbols may be given, if comma-separated.

-fixnum-arithmetic
Equivalent to `(fixnum-arithmetic)` declaration. Assume all mathematical operations use small integer arguments.

-heap-size NUMBER
Sets a fixed heap size of the generated executable to `NUMBER` bytes. The parameter may be followed by a `M` (`m`) or `K` (`k`) suffix which stand for mega- and kilobytes, respectively. The default heap size is 5 kilobytes. Note that only half of it is in use at every given time.

-heap-initial-size NUMBER
Sets the size that the heap of the compiled application should have at startup time.

-heap-growth PERCENTAGE
Sets the heap-growth rate for the compiled program at compile time (see: `-:hg`).

-heap-shrinkage PERCENTAGE
Sets the heap-shrinkage rate for the compiled program at compile time (see: `-:hs`).

-help
Print a summary of available options and the format of the command line parameters and exit the compiler.

-ignore-repository
Do not load any extensions from the repository (treat repository as empty). Also do not consult compiled (only interpreted) import libraries in `import` forms.

-include-path PATHNAME
Specifies an additional search path for files included via the `include` special form. This option may be given multiple times. If the environment variable `CHICKEN_INCLUDE_PATH` is set, it should contain a list of alternative include pathnames separated by `;`.

-inline
Enable procedure inlining for known procedures of a size below the threshold (which can be set through the `-inline-limit` option).

-inline-global
Enable cross-module inlining (in addition to local inlining). Implies `-inline`. For more information, see also Declarations.

-inline-limit THRESHOLD
Sets the maximum size of a potentially inlinable procedure. The default threshold is `20`.

-keyword-style STYLE
Enables alternative keyword syntax, where `STYLE` may be either `prefix` (as in Common Lisp, e.g. :keyword), `suffix` (as in DSSSL, e.g. keyword:) or `none`. Any other value is ignored. The default is `suffix`.

-keep-shadowed-macros
Do not remove macro definitions with the same name as assigned toplevel variables (the default is to remove the macro definition).

-lambda-lift
Enable the optimization known as lambda-lifting.

-local
Assume toplevel variables defined in the current compilation unit are not externally modified.

-no-argc-checks
disable argument count checks

-no-bound-checks
disable bound variable checks

-no-feature SYMBOL
Disables the predefined feature-identifier `SYMBOL`. Multiple symbols may be given, if comma-separated.

-no-lambda-info
Don't emit additional information for each `lambda` expression (currently the argument-list, after alpha-conversion/renaming).

-no-module-registration
Do not generate module-registration code in the compiled code. This is only needed if you want to use an import library that is generated by other means (manually, for example).

-no-parentheses-synonyms STYLE
Disables list delimiter synonyms, [..] and {...} for (...).

-no-procedure-checks
disable procedure call checks

-no-procedure-checks-for-usual-bindings
disable procedure call checks only for usual bindings

-no-procedure-checks-for-toplevel-bindings
disable bound and procedure call checks for calls to procedures referenced through a toplevel variable.

-no-symbol-escape
Disables support for escaped symbols, the |...| form.

-no-trace
Disable generation of tracing information. If a compiled executable should halt due to a runtime error, then a list of the name and the line-number (if available) of the last procedure calls is printed, unless `-no-trace` is specified. With this option the generated code is slightly faster.

-no-warnings
Disable generation of compiler warnings.

-nursery NUMBER

-stack-size NUMBER
Sets the size of the first heap-generation of the generated executable to `NUMBER` bytes. The parameter may be followed by a `M` (`m`) or `K` (`k`) suffix. The default stack-size depends on the target platform.

-optimize-leaf-routines
Enable leaf routine optimization.

-optimize-level LEVEL
Enables certain sets of optimization options. `LEVEL` should be an integer.

```
    -optimize-level 0          is equivalent to -no-usual-integrations -no-compiler
    -optimize-level 1          is equivalent to -optimize-leaf-routines
    -optimize-level 2          is equivalent to -optimize-leaf-routines -inline
    -optimize-level 3          is equivalent to -optimize-leaf-routines -local -inl
    -optimize-level 4          is equivalent to -optimize-leaf-routines -local -inl
    -optimize-level 5          is equivalent to -optimize-leaf-routines -block -inl
```

-output-file FILENAME
Specifies the pathname of the generated C file. Default is `FILENAME.c`.

-postlude EXPRESSIONS
Add `EXPRESSIONS` after all other toplevel expressions in the compiled file. This option may be given multiple times. Processing of this option takes place after processing of `-epilogue`.

-prelude EXPRESSIONS
Add `EXPRESSIONS` before all other toplevel expressions in the compiled file. This option may be given multiple times. Processing of this option takes place before processing of `-prologue`.

-profile

-accumulate-profile
Instruments the source code to count procedure calls and execution times. After the program terminates (either via an explicit `exit` or implicitly), profiling statistics are written to a file named `PROFILE.<randomnumber>`. Each line of the generated file contains a list with the procedure name, the number of calls and the time spent executing it. Use the `chicken-profile` program to display the profiling information in a more user-friendly form. Enter `chicken-profile` with no arguments at the command line to get a list of available options. The `-accumulate-profile` option is similar to `-profile`, but the resulting profile information will be appended to any existing `PROFILE` file. `chicken-profile` will merge and sum up the accumulated timing information, if several entries for the same procedure calls exist. Only profiling information for global procedures will be collected.

-profile-name FILENAME
Specifies name of the generated profile information (which defaults to `PROFILE.<randomnumber>`. Implies `-profile`.

-prologue FILENAME
Includes the file named `FILENAME` at the start of the compiled source file. The include-path is not searched. This option may be given multiple times.

-r5rs-syntax
Disables the Chicken extensions to R5RS syntax. Does not disable non-standard read syntax.

-raw
Disables the generation of any implicit code that uses the Scheme libraries (that is all runtime system files besides `runtime.c` and `chicken.h`).

-require-extension NAME
Loads the extension `NAME` before the compilation process commences. This is identical to adding `(require-extension NAME)` at the start of the compiled program. If `-uses NAME` is also given on the command line, then any occurrences of `-require-extension NAME` are replaced with `(declare (uses NAME))`. Multiple names may be given and should be separated by `,`.

-setup-mode
When locating extension, search the current directory first. By default, extensions are located first in the *extension repository*, where `chicken-install` stores compiled extensions and their associated metadata.

-scrutinize
Enable simple flow-analysis to catch common type errors and argument/result mismatches. You can also use the `scrutinize` declaration to enable scrutiny.

-static-extension NAME
similar to `-require-extension NAME`, but links extension statically (also applies for an explicit `(require-extension NAME)`).

-types FILENAME
load additional type database from `FILENAME`. Type-definitions in `FILENAME` will override previous type-definitions.

-compile-syntax
Makes macros also available at run-time. By default macros are not available at run-time.

-to-stdout
Write compiled code to standard output instead of creating a `.c` file.

-unboxing
try to use unboxed temporaries for numerical operations. This optimization is only effective in unsafe mode.

-unit NAME
Compile this file as a library unit. Equivalent to `-prelude "(declare (unit NAME))"`

-unsafe
Disable runtime safety checks.

-uses NAME
Use definitions from the library unit `NAME`. This is equivalent to `-prelude "(declare (uses NAME))"`. Multiple arguments may be given, separated by `,`.

-no-usual-integrations
Specifies that standard procedures and certain internal procedures may be redefined, and can not be inlined. This is equivalent to declaring `(not usual-integrations)`.

-version
Prints the version and some copyright information and exit the compiler.

-verbose
Prints progress information to standard output during compilation.

The environment variable `CHICKEN_OPTIONS` can be set to a string with default command-line options for the compiler.

Further options

Enter

```
csc -help
```

to see a list of all supported options and short aliases to basic options.

Runtime options

After successful compilation a C source file is generated and can be compiled with a C compiler. Executables generated with CHICKEN (and the compiler itself) accept a small set of runtime options:

-:?
Shows a list of the available runtime options and exits the program.

-:aNUMBER
Specifies the length of the buffer for recording a trace of the last invoked procedures. Defaults to 16.

-:b
Enter a read-eval-print-loop when an error is encountered.

-:B
Sounds a bell (ASCII 7) on every major garbage collection.

-:c
Forces console mode. Currently this is only used in the interpreter (`csi`) to force output of the `#;N>` prompt even if stdin is not a terminal (for example if running in an `emacs` buffer under Windows).

-:d
Prints some debug-information at runtime.

-:D
Prints some more debug-information at runtime.

-:g
Prints information about garbage-collection.

-:G
Force GUI mode (show error messages in dialog box, suitable for platform).

-:H
Before terminating, dump heap usage to stderr.

-:fNUMBER
Specifies the maximal number of currently pending finalizers before finalization is forced.

-:hNUMBER
Specifies fixed heap size

-:hgPERCENTAGE
Sets the growth rate of the heap in percent. If the heap is exhausted, then it will grow by `PERCENTAGE`. The default is 200.

-:hiNUMBER
Specifies the initial heap size

-:hmNUMBER
Specifies a maximal heap size. The default is (2GB - 15).

-:hsPERCENTAGE
Sets the shrink rate of the heap in percent. If no more than a quarter of `PERCENTAGE` of the heap is used, then it will shrink to `PERCENTAGE`. The default is 50. Note: If you want to make sure that the heap never shrinks, specify a value of `0`. (this can be useful in situations where an optimal heap-size is known in advance).

-:o
Disables detection of stack overflows at run-time.

-:r
Writes trace output to stderr. This option has no effect with in files compiled with the `-no-trace` options.

-:sNUMBER
Specifies stack size.

-:tNUMBER
Specifies symbol table size.

-:w
Enables garbage collection of unused symbols. By default unused and unbound symbols are not garbage collected.

-:x
Raises uncaught exceptions of separately spawned threads in primordial thread. By default uncaught exceptions in separate threads are not handled, unless the primordial one explicitly joins them. When warnings are enabled (the default) and `-:x` is not given, a warning will be shown, though.

The argument values may be given in bytes, in kilobytes (suffixed with `K` or `k`), in megabytes (suffixed with `M` or `m`), or in gigabytes (suffixed with `G` or `g`). Runtime options may be combined, like `-:dc`, but everything following a `NUMBER` argument is ignored. So `-:wh64m` is OK, but `-:h64mw` will not enable GC of unused symbols.

Examples

A simple example (with one source file)

To compile a Scheme program (assuming a UNIX-like environment) consisting of a single source file, perform the following steps.

Writing your source file

In this example we will assume your source file is called `foo.scm`:

```
;;; foo.scm

(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1))) ) )

(write (fac 10))
(newline)
```

Compiling your program

Compile the file `foo.scm`:

```
% csc foo.scm
```

This will produce the `foo` executable:

```
% ls
foo  foo.scm
```

Running your program

To run your newly compiled executable use:

```
% foo
3628800
```

If you get a `foo: command not found` error, you might want to try with `./foo` instead (or, in Unix machines, modify your `PATH` environment variable to include your current directory).

An example with multiple files

If multiple bodies of Scheme code are to be combined into a single executable, then we have to compile each file and link the resulting object files together with the runtime system.

Let's consider an example where your program consists of multiple source files.

Writing your source files

The declarations in these files specify which of the compiled files is the main module, and which is the library module. An executable can only have one main module, since a program has only a single entry-point. In this case `foo.scm` is the main module, because it doesn't have a `unit` declaration:

```
;;; foo.scm

; The declaration marks this source file as dependant on the symbols provided
; by the bar unit:
(declare (uses bar))

(write (fac 10)) (newline)
```

`bar.scm` will be our library:

```
;;; bar.scm

; The declaration marks this source file as the bar unit.  The names of the
; units and your files don't need to match.
(declare (unit bar))

(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1))) ) )
```

Compiling and running your program

You should compile your two files with the following commands:

```
% csc -c bar.scm
% csc -c foo.scm
```

That should produce two files, `bar.o` and `foo.o`. They contain the code from your source files in compiled form.

To link your compiled files use the following command:

```
% csc foo.o bar.o -o foo
```

This should produce the `foo` executable, which you can run just as in the previous example. At this point you can also erase the `*.o` files.

You could avoid one step and link the two files just as `foo.scm` is compiled:

```
% csc -c bar.scm
% csc foo.scm bar.o -o foo
```

Note that if you want to distribute your program, you might want it to follow the GNU Coding Standards. One relatively easy way to achieve this is to use Autoconf and Automake, two tools made for this specific purpose.

Extending the compiler

The compiler supplies a couple of hooks to add user-level passes to the compilation process. Before compilation commences any Scheme source files or compiled code specified using the `-extend` option are loaded and evaluated. The parameters `user-options-pass, user-read-pass, user-preprocessor-pass, user-pass` and `user-post-analysis-pass` can be set to procedures that are called to perform certain compilation passes instead of the usual processing (for more information about parameters see: Supported language.

[parameter] `user-options-pass`

Holds a procedure that will be called with a list of command-line arguments and should return two values: the source filename and the actual list of options, where compiler switches have their leading `-` (hyphen) removed and are converted to symbols. Note that this parameter is invoked **before** processing of the `-extend` option, and so can only be changed in compiled user passes.

[parameter] `user-read-pass`

Holds a procedure of three arguments. The first argument is a list of strings with the code passed to the compiler via `-prelude` options. The second argument is a list of source files including any files specified by `-prologue` and `-epilogue`. The third argument is a list of strings specified using `-postlude` options. The procedure should return a list of toplevel Scheme expressions.

[parameter] `user-preprocessor-pass`

Holds a procedure of one argument. This procedure is applied to each toplevel expression in the source file **before** macro-expansion. The result is macro-expanded and compiled in place of the original expression.

[parameter] `user-pass`

Holds a procedure of one argument. This procedure is applied to each toplevel expression **after** macro-expansion. The result of the procedure is then compiled in place of the original expression.

[parameter] `user-post-analysis-pass`

Holds a procedure that will be called after every performed program analysis pass. The procedure (when defined) will be called with seven arguments: a symbol indicating the analysis pass, the program database, the current node graph, a getter and a setter-procedure which can be used to access and manipulate the program database, which holds various information about the compiled program, a pass iteration count, and an analysis continuation flag. The getter procedure should be called with two arguments: a symbol representing the binding for which information should be retrieved, and a symbol that specifies the database-entry. The current value of the database entry will be returned or `#f`, if no such entry is available. The setter procedure is called with three arguments: the symbol and key and the new value. The pass iteration count currently is meaningful only for the 'opt pass. The analysis continuation flag will be `#f` for the last 'opt pass. For information about the contents of the program database contact the author.

Loaded code (via the `-extend` option) has access to the library units `extras, srfi-1, srfi-4, utils, regex` and the pattern matching macros. Multithreading is not available.

Note that the macroexpansion/canonicalization phase of the compiler adds certain forms to the source program. These extra expressions are not seen by `user-preprocessor-pass` but by `user-pass`.

Distributing compiled C files

It is relatively easy to create distributions of Scheme projects that have been compiled to C. The runtime system of CHICKEN consists of only two handcoded C files (`runtime.c` and `chicken.h`), plus the file `chicken-config.h`, which is generated by the build process. All other modules of the runtime system and the extension libraries are just compiled Scheme code. The following example shows a minimal application, which should run without changes on the most frequent operating systems, like Windows, Linux or FreeBSD:

Let's take a simple example.

```
; hello.scm

(print "Hello, world!")
```

```
 % csc -t hello.scm -optimize-level 3 -output-file hello.c
```

Compiled to C, we get `hello.c`. We need the files `chicken.h` and `runtime.c`, which contain the basic runtime system, plus the three basic library files `library.c`, `eval.c` and `extras.c` which contain the same functionality as the library linked into a plain CHICKEN-compiled application, or which is available by default in the interpreter, `csi`:

```
 % cd /tmp
 %echo '(print "Hello World.")' > hello.scm
 % cp $CHICKEN_BUILD/runtime.c .
 % cp $CHICKEN_BUILD/library.c .
 % cp $CHICKEN_BUILD/eval.c    .
 % cp $CHICKEN_BUILD/extras.c  .
 % gcc -static -Os -fomit-frame-pointer runtime.c library.c eval.c \
   extras.c hello.c -o hello -lm
```

Now we have all files together, and can create an tarball containing all the files:

```
% tar cf hello.tar Makefile hello.c runtime.c library.c eval.c extras.c chicken.h
% gzip hello.tar
```

This is naturally rather simplistic. Things like enabling dynamic loading, estimating the optimal stack-size and selecting supported features of the host system would need more configuration- and build-time support. All this can be addressed using more elaborate build-scripts, makefiles or by using autoconf/automake.

Note also that the size of the application can still be reduced by removing `extras` and `eval` and compiling `hello.scm` with the `-explicit-use` option.

For more information, study the CHICKEN source code and/or get in contact with the author.

Previous: Basic mode of operation

Next: Using the interpreter

Supported language

Previous: Using the interpreter

Next: Interface to external functions and variables

Interface to external functions and variables

The macros in this section, such as `define-foreign-type` and `define-external`, are available in the `foreign` import library. To access them:

```
(import foreign)
```


Previous: Supported language

Next: Extensions

Extensions

Extension libraries

Extension libraries (*eggs*) are extensions to the core functionality provided by the basic CHICKEN system, to be built and installed separately. The mechanism for loading compiled extensions is based on dynamically loadable code and as such is only available on systems on which loading compiled code at runtime is supported. Currently these are most UNIX-compatible platforms that provide the `libdl` functionality like Linux, Solaris, BSD, Mac OS X and Windows using Cygwin.

Note: Extension may also be normal applications or shell scripts, but are usually libraries.

Extensions are technically nothing but dynamically loadable compiled files with added meta-data that describes dependencies to other extensions, version information and things like the author/maintainer of the extension. Three tools provide an easy to use interface for installing extensions, removing them and querying the current status of installed extensions.

Installing extensions

To install an extension library, run the `chicken-install` program with the extension name as argument. The extension archive is downloaded, its contents extracted and the contained *setup* script is executed. This setup script is a normal Scheme source file, which will be interpreted by `chicken-install`. The complete language supported by `csi` is available, and the library units `srfi-1 regex utils posix tcp` are loaded. Additional libraries can be loaded at run-time.

The setup script should perform all necessary steps to build the new library (or application). After a successful build, the extension can be installed by invoking one of the procedures `install-extension`, `install-program` or `install-script`. These procedures will copy a number of given files into the local extension repository or in the path where the CHICKEN executables are located (in the case of executable programs or scripts). Additionally the list of installed files, and user-defined metadata is stored in the repository.

If no extension name is given on the command-line, then all `.setup` scripts in the current directory are processed, in the order given on the command line.

Installing extensions that use libraries

Sometimes an extension requires a C library to compile. Compilation can fail when your system has this library in a nonstandard location. Normally the C compiler searches in the default locations `/usr` and `/usr/local`, and in the prefix where Chicken itself was installed. Sometimes this is not enough, so you'll need to supply `chicken-install` with some extra hints to the C compiler/linker. Here's an example:

```
CSC_OPTIONS='-I/usr/pkg/include/mysql -L/usr/pkg/lib/mysql -L -R/usr/pkg/lib/mysql
```

This installs the mysql egg with the extra compiler options -I and -L to set the include path and the library search path. The second -L switch passes the -R option directly to the linker, which causes the library path to get hardcoded into the resulting extension file (for systems that do not use `ld.so.conf`).

Creating extensions

Extensions can be created by creating an (optionally gzipped) `tar` archive named `EXTENSION.egg` containing all needed files plus a `.setup` script in the root directory. After `chicken-install` has extracted the files, the setup script will be invoked. There are no additional constraints on the structure of the archive, but the setup script has to be in the root path of the archive.

For more details on creating extensions, see the eggs tutorial.

Procedures and macros available in setup scripts

install-extension

[procedure] `(install-extension ID FILELIST [INFOLIST])`

Installs the extension library with the name `ID`. All files given in the list of strings `FILELIST` will be copied to the extension repository. It should be noted here that the extension id has to be identical to the name of the file implementing the extension. The extension may load or include other files, or may load other extensions at runtime specified by the `require-at-runtime` property.

`FILELIST` may be a filename, a list of filenames, or a list of pairs of the form `(SOURCE DEST)` (if you want to copy into a particular sub-directory - the destination directory will be created as needed). If `DEST` is a relative pathname, it will be copied into the extension repository.

The optional argument `INFOLIST` should be an association list that maps symbols to values, this list will be stored as `ID.setup-info` at the same location as the extension code. Currently the following properties are used:

syntax

```
[extension property] (syntax)
```

Marks the extension as syntax-only. No code is compiled, the extension is intended as a file containing macros to be loaded at compile/macro-expansion time.

require-at-runtime

```
[extension property] (require-at-runtime ID ...)
```

Specifies extensions that should be loaded (via `require`) at runtime. This is mostly useful for syntax extensions that need additional support code at runtime.

import-only

```
 [extension property] (import-only)
```

Specifies that this extension only provides a expansion-time code in an import library and does not require code to be loaded at runtime.

version

```
[extension property] (version STRING)
```

Specifies version string.

static

```
[extension property] (static STRING)
```

If the extension also provides a static library, then STRING should contain the name of that library. Used by `csc` when compiling with the `-static-extension` option.

static-options

```
[extension property] (static-options STRING)
```

Additional options that should be passed to the linker when linking with the static version of an extension (see `static` above). Used by `csc` when compiling with the `-static-extension` option.

All other properties are currently ignored. The `FILELIST` argument may also be a single string.

install-program

[procedure] `(install-program ID FILELIST [INFOLIST])`

Similar to `install-extension`, but installs an executable program in the executable path (usually `/usr/local/bin`).

install-script

[procedure] `(install-script ID FILELIST [INFOLIST])`

Similar to `install-program`, but additionally changes the file permissions of all files in `FILELIST` to executable (for installing shell-scripts).

standard-extension

[procedure] `(standard-extension ID VERSION #!key static info)`

A convenience procedure that combines the compilation and installation of a simple single-file extension. This is roughly equivalent to:

```
(compile -s -O2 -d1 ID.scm -j ID)
(compile -c -O2 -d1 ID.scm -j ID -unit ID)  ; if STATIC is not given or true
(compile -s -O2 -d0 ID.import.scm)

(install-extension
 'ID
 '("ID.o" "ID.so" "ID.import.so")
 '((version 1.0)
   ... `INFO' ...
   (static "ID.o")))   ; if `static' is given and true
```

`VERSION` may be `#f`, in that case the version obtained from where the extension has been retrieved wil be taken. If installed directly from a local directory, the version will default to `"unknown"`.

run

[syntax] `(run FORM ...)`

Runs the shell command `FORM`, which is wrapped in an implicit `quasiquote`. `(run (csc ...))` is treated specially and passes `-v` (if `-verbose` has been given to `chicken-install`) and `-feature compiling-extension` options to the compiler.

compile

[syntax] `(compile FORM ...)`

Equivalent to `(run (csc FORM ...))`.

make

[syntax] `(make ((TARGET (DEPENDENT ...) COMMAND ...) ...) ARGUMENTS)`

A *make* macro that executes the expressions `COMMAND ...`, when any of the dependents `DEPENDENT ...` have changed, to build `TARGET`. This is the same as the `make` extension, which is available separately. For more information, see make.

patch

[procedure] `(patch WHICH REGEX SUBST)`

Replaces all occurrences of the regular expression `REGEX` with the string `SUBST`, in the file given in `WHICH`. If `WHICH` is a string, the file will be patched and overwritten. If `WHICH` is a list of the form `OLD NEW`, then a different file named `NEW` will be generated.

copy-file

[procedure] `(copy-file FROM TO)`

Copies the file or directory (recursively) given in the string `FROM` to the destination file or directory `TO`.

move-file

[procedure] `(move-file FROM TO)`

Moves the file or directory (recursively) given in the string `FROM` to the destination file or directory `TO`.

remove-file*

[procedure] `(remove-file* PATH)`

Removes the file or directory given in the string `PATH`, if it exists.

find-library

[procedure] `(find-library NAME PROC)`

Returns `#t` if the library named `libNAME.[a|so]` (unix) or `NAME.lib` (windows) could be found by compiling and linking a test program. `PROC` should be the name of a C function that must be provided by the library. If no such library was found or the function could not be resolved, `#f` is returned.

find-header

[procedure] `(find-header NAME)`

Returns `#t` if a C include-file with the given name is available, or `#f` otherwise.

try-compile

[procedure] `(try-compile CODE #!key cc cflags ldflags compile-only c++)`

Returns `#t` if the C code in `CODE` compiles and links successfully, or `#f` otherwise. The keyword parameters `cc` (compiler name, defaults to the C compiler used to build this system), `cflags` and `ldflags` accept additional compilation and linking options. If `compile-only` is true, then no linking step takes place. If the keyword argument `c++` is given and true, then the code will be compiled in C++ mode.

create-directory/parents

[procedure] `(create-directory/parents PATH)`

Creates the directory given in the string `PATH`, with all parent directories as needed.

extension-name-and-version

[parameter] `extension-name-and-version`

Returns a list containing the name and version of the currently installed extension as strings. If the setup script is not invoked via `chicken-install`, then name and version will be empty.

version>=?

[procedure] `(version>=? V1 V2)`

Compares the version numbers `V1` and `V2` and returns `#t` if `V1` is "less" than `V2` or `#f` otherwise. A version number can be an integer, a floating-point number or a string. `version>=?` handles dot-separated version-indicators of the form `"X.Y. .."`.

If one version number is the prefix of the other, then the shorter version is considered "less" than the longer.

installation-prefix

[procedure] `(installation-prefix)`

An alternative installation prefix that will be prepended to extension installation paths if specified. It is set by the `-prefix` option or environment variable `CHICKEN_INSTALL_PREFIX`.

program-path

[parameter] `(program-path [PATH])`

Holds the path where executables are installed and defaults to either `$CHICKEN_PREFIX/bin`, if the environment variable `CHICKEN_PREFIX` is set or the path where the CHICKEN binaries (`chicken`, `csi`, etc.) are installed.

setup-root-directory

[parameter] `(setup-root-directory [PATH])`

Contains the path of the directory where `chicken-install` was invoked.

setup-install-mode

[parameter] `(setup-install-mode [BOOL])`

Reflects the setting of the `-no-install` option, i.e. is `#f`, if `-no-install` was given to `chicken-install`.

required-chicken-version

[procedure] `(required-chicken-version VERSION)`

Signals an error if the version of CHICKEN that this script runs under is lexicographically less than `VERSION` (the argument will be converted to a string, first).

required-extension-version

[procedure] `(required-extension-version EXTENSION1 VERSION1 ...)`

Checks whether the extensions `EXTENSION1 ...` are installed and at least of version `VERSION1 ...`. The test is made by lexicographically comparing the string-representations of the given version with the version of the installed extension. If one of the listed extensions is not installed, has no associated version information or is of a version older than the one specified.

host-extension

[parameter] `host-extension`

For a cross-compiling CHICKEN, when compiling an extension, then it should be built for the host environment (as opposed to the target environment). This parameter is controlled by the `-host` command-line option. A setup script should perform the proper steps of compiling any code by passing `-host` when invoking `csc` or using the `compile` macro.

Examples for extensions

A simple library

The simplest case is a single file that does not export any syntax. For example

```
;;;; hello.scm

(define (hello name)
  (print "Hello, " name " !") )
```

We need a `.setup` script to build and install our nifty extension:

```
;;;; hello.setup

;; compile the code into a dynamically loadable shared object
;; (will generate hello.so)
(compile -s hello.scm)

;; Install as extension library
(install-extension 'hello "hello.so")
```

Lastly, we need a file `hello.meta` defining a minimal set of properties:

```
;;;; hello.meta

((author "Me")
 (synopsis "A cool hello-world library")
 (license "GPLv3"))
```

(for more information about available properties, see the metafile reference)

After entering

```
$ chicken-install
```

at the shell prompt (and in the same directory where the two files exist), the file `hello.scm` will be compiled into a dynamically loadable library. If the compilation succeeds, `hello.so` will be stored in the repository, together with a file named `hello.setup-info` containing an a-list with metadata (what you stored above in `hello.meta`). If no extension name is given to `chicken-install`, it will simply execute the any files with the `.setup` extension it can find.

Use it like any other CHICKEN extension:

```
$ csi -q
#;1> (require-library hello)
; loading /usr/local/lib/chicken/4/hello.so ...
#;2> (hello "me")
Hello, me!
#;3>
```

An application

Here we create a simple application:

```
;;;; hello2.scm

(print "Hello, ")
```

```
(for-each (lambda (x) (printf "~A " x)) (command-line-arguments))
(print "!")
```

We also need a setup script:

```
;;;; hello2.setup

(compile hello2.scm)  ; compile `hello2'
(install-program 'hello2 "hello2") ; name of the extension and files to be installe
```

```
;;;; hello2.meta

((author "Me")
 (synopsis "A cool hello-world application")
 (license "proprietary"))
```

To use it, just run `chicken-install` in the same directory:

```
$ chicken-install
```

(Here we omit the extension name)

Now the program `hello2` will be installed in the same location as the other CHICKEN tools (like `chicken`, `csi`, etc.), which will normally be `/usr/local/bin`. Note that you need write-permissions for those locations and may have to run `chicken-install` with administrative rights or use the `-sudo` option.

The extension can be used from the command line:

```
$ hello2 one two three
Hello,
one two three !
```

De-installation is just as easy - use the `chicken-uninstall` program to remove one or more extensions from the local repository:

```
$ chicken-uninstall hello2
```

A module exporting syntax

The hello module was just a shared library, and not a module.

To create an extension that exports syntax see the chapter on Modules and macros. We will show a simple example here: a module `my-lib` that exports one macro (`prog1`) and one procedure (`my-sum`):

```
;;; my-lib.scm

(module my-lib
  *
  (import scheme chicken)

(define-syntax prog1
  (syntax-rules ()
    ((_ e1 e2 ...)
     (let ((result e1))
       (begin e2 ...)
       result))))

(define my-sum
```

```
  (lambda (numbers)
    (prog1
      (apply + numbers)
      (display "my-sum used one more time!")
      (newline))))

)
```

The `prog1` macro is similar to Common Lisp's `prog1`: it evaluates a list of forms, but returns the value of the first form.

The meta file:

```
;;; my-lib.meta

((licence "BSD")
 (author "Me again")
 (synopsis "My own cool libraries"))
```

The setup file is:

```
;;; my-lib.setup

(compile -s -O3 -d1 "my-lib.scm" -j my-lib)
(compile -c -O3 -d1 "my-lib.scm" -unit my-lib)
(compile -s -O3 -d0 "my-lib.import.scm")

(install-extension
 'my-lib
 '("my-lib.o" "my-lib.so" "my-lib.import.so")
 '((version 1.0)
   (static "my-lib.o")))
```

The first line tells the compiler to create a shared (`-s`) library and to create an import file (`my-lib.import.scm`, because of the `-j` flag). The second line creates a static library `my-lib.o`. The third line compiles the import file created by the first one.

IMPORTANT: the module name exported by my-lib.scm must be the same module name passed to the compiler using the -j option, otherwise the imports file will not be generated!

Running `chicken-install` on the same directory will install the extension.

Next, it should be possible to load the library:

```
$ csi -q
#;1> (use my-lib)
; loading /usr/local/lib/chicken/5/my-lib.import.so ...
; loading /usr/local/lib/chicken/5/scheme.import.so ...
; loading /usr/local/lib/chicken/5/chicken.import.so ...
; loading /usr/local/lib/chicken/5/my-lib.so ...
#;2> (my-sum '(10 20 30))
my-sum used one more time!
60
#;3> (my-sum '(-1 1 0))
my-sum used one more time!
0
#;4> (prog1 (+ 2 2) (print "---"))
---
4
```

Notes on chicken-install

When running `chicken-install` with an argument `NAME`, for which no associated `.setup` file exists, then it will try to download the extension via HTTP from the CHICKEN code repository at http://code.call-cc.org/svn/chicken-eggs/. Extensions that are required to compile and/or use the requested extension are downloaded and installed automatically.

To query the list of currently installed extensions, use `chicken-status`. It can list what extensions are installed and what files belong to a particular installed extension.

chicken-install reference

Available options:

`-h -help`
show this message and exit

`-v -version`
show version and exit

`-force`
don't ask, install even if versions don't match

`-k -keep`
keep temporary files

`-l -location LOCATION`
install from given location instead of default

`-t -transport TRANSPORT`
use given transport instead of default

`-proxy HOST[:PORT]`
connect via HTTP proxy

`-s -sudo`
use `sudo(1)` for installing or removing files

`-r -retrieve`
only retrieve egg into current directory, don't install

`-n -no-install`
do not install, just build (implies `-keep`)

`-p -prefix PREFIX`
change installation prefix to `PREFIX`

`-host`
when cross-compiling, compile extension for host only

`-target`
when cross-compiling, compile extension for target only

`-test`
run included test-cases, if available

`-username USER`
set username for transports that require this

`-password PASS`
set password for transports that require this

`-i -init DIRECTORY`
initialize empty alternative repository

`-u -update-db`
update export database

`-repository`
print path to extension repository

`-deploy`
install extension in the application directory for a deployed application (see Deployment for more information)

`-trunk`
build trunk instead of tagged version (only local)

`-D -feature FEATURE`
pass this on to subinvocations of `csi` and `csc` (when done via `compile` or `(run (csc ...))`)

`-debug`
print full call-trace when encountering errors in the setup script

`-keep-going`
continue installation, even if a dependency fails

`chicken-install` recognizes the `http_proxy` environment variable, if set.

chicken-uninstall reference

`-h -help`
show usage information and exit

`-v -version`
show version and exit

`-force`
don't ask, delete whatever matches

`-s -sudo`
use `sudo(1)` for deleting files

`-host`
when cross-compiling, remove extensions for host system only

`-target`
when cross-compiling, remove extensions for target system only

`-exact`
match extension-name exactly (do not match as pattern)

chicken-status reference

`-h -help`
show usage information and exit

`-v -version`
show version and exit

`-f -files`
list installed files

`-host`
when cross-compiling, show extensions for host system only

`-target`
when cross-compiling, show extensions for target system only

`-exact`
match extension-name exactly (do not match as pattern)

Security

When extensions are downloaded and installed one is executing code from potentially compromised systems. This applies also when `chicken-install` executes system tests for required extensions. As the code has been retrieved over the network effectively untrusted code is going to be evaluated. When `chicken-install` is run as *root* the whole system is at the mercy of the build instructions (note that this is also the case every time you install software via `sudo make install`, so this is not specific to the CHICKEN extension mechanism).

Security-conscious users should never run `chicken-install` as root. A simple remedy is to keep the repository inside a user's home directory (see the section "Changing repository location" below). Alternatively obtain write/execute access to the default location of the repository (usually `/usr/local/lib/chicken`) to avoid running as root. `chicken-install` also provides a `-sudo` option to perform the last installation steps as root user, but do building and other .setup script processing as normal. A third solution is to override `VARDIR` when building the system (for example by passing

"VARDIR=/foo/bar" on the make command line, or by modifying `config.make`. Eggs will then be installed in `$(VARDIR)/chicken/5`.

Changing repository location

When Chicken is installed a repository for eggs is created and initialized in a default location (usually something like `/usr/local/lib/chicken/5/`). It is possible to keep an eggs repository in another location. This can be configured at build-time by passing `VARDIR=<directory>` to `make(3)` or by modifying the `config.make` configuration file. If you want to override this location after chicken is installed, you can create an initial repository directory with some default extensions and set the `CHICKEN_REPOSITORY` environment variable:

First, initialize the new repository with

```
chicken-install -init ~/myeggs/lib/chicken/5
```

Then set this environment variable:

```
export CHICKEN_REPOSITORY=~/myeggs/lib/chicken/5
```

`CHICKEN_REPOSITORY` is the place where extensions are to be loaded from for all chicken-based programs (which includes all the tools).

You can install eggs with

```
chicken-install -p ~/myeggs <package>
```

See that the argument to chicken-install is just `~/myeggs`, while everywhere else it's `~/myeggs/lib/chicken/5`.

When you load eggs from the interpreter, you will see messages showing where libraries are being loaded from:

```
#;1> (use numbers)
; loading /home/jdoe/myeggs/lib/chicken/5/numbers.import.so ...
; loading /home/jdoe/myeggs/lib/chicken/5/scheme.import.so ...
; loading /home/jdoe/myeggs/lib/chicken/5/chicken.import.so ...
; loading /home/jdoe/myeggs/lib/chicken/5/foreign.import.so ...
; loading /home/jdoe/myeggs/lib/chicken/5/regex.import.so ...
; loading /home/jdoe/myeggs/lib/chicken/5/numbers.so ...
#;2>
```

Other modes of installation

It is possible to install extensions directly from a Subversion repository or from a local checkout of the repository tree by using the `-transport` and `-location` options when invoking `chicken-install`. Three possible transport mechanisms are currently supported:

`http`
  download extension sources via HTTP from a web-server (this is the default)

`svn`
  perform an `svn export` from the central extension repository; this will require a `svn(1)` client to be installed on the machine

`local`
  use sources from the local filesystem and build directly in the source directory

The `-location` option specifies where to look for the source repository and names a web URL, a subversion repository URL or a filesystem path, respectively. A list of locations to try when retrieving extensions is stored in the file `setup.defaults` (usually installed in `/usr/local/share/chicken`). For `http` transports, `chicken-install` will detect networking timeouts and try alternative locations, as listed in the file.

Dependency information, which is necessary to ensure required extensions are also installed, is processed automatically.

Linking extensions statically

The compiler and `chicken-install` support statically linked eggs. The general approach is to generate an object file or static library (in addition to the usual shared library) in your `.setup` script and install it along with the dynamically loadable extension. The setup properties `static` should contain the name of the object file (or static library) to be linked, when `csc` gets passed the `-static-extension` option:

```
(compile -s -O2 -d1 my-ext.scm)   ; dynamically loadable "normal" version
(compile -c -O2 -d1 my-ext -unit my-ext)  ; statically linkable version
(install-extension
  'my-ext
  '("my-ext.so" "my-ext.o")
  '((static "my-ext.o")) )
```

Note the use of the `-unit` option in the second compilation step: static linking must use static library units. `chicken-install` will perform platform-dependent file-extension translation for the file list, but does currently not do that for the `static` extension property.

To actually link with the static version of `my-ext`, do:

```
% csc -static-extension my-ext my-program.scm
```

The compiler will try to do the right thing, but can not handle all extensions, since the ability to statically link eggs is relatively new. Eggs that support static linking are designated as being able to do so. If you require a statically linkable version of an egg that has not been converted yet, contact the extension author or the CHICKEN mailing list.

Previous: Interface to external functions and variables

Next: Deployment

Deployment

CHICKEN generates fully native binaries that can be distributed like normal C/C++ programs. There are various methods of deployment, depending on platform, linkage, external dependencies and whether the application should be built from sources or precompiled and whether the CHICKEN runtime-libraries are expected on the destination system or if the application should be completely self-contained.

Simple executables

The simplest form of deployment is the single executable. The runtime libraries (`libchicken.so` or `libchicken.dll` is required for these programs to run, unless you link your application statically:

```
% csc myprogram.scm
% ldd myprogram               # on linux
          linux-gate.so.1 =>  (0xb805c000)
          libchicken.so.5 => /home/felix/chicken/core/lib/libchicken.so.5 (0xb7c22
          libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7bec000)
          libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7be7000)
          libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7a84000)
          /lib/ld-linux.so.2 (0xb805d000)
% ls -l myprogram
-rwxr-xr-x 1 felix felix   34839 2010-02-22 20:19 x
```

Linking your application statically will include the runtime library in the executable, but this will increase its size substantially:

```
% ls myprogram
-rwxr-xr-x 1 felix felix 3566656 2010-02-22 20:30 myprogram
```

Programs distributed this way can only use Extensions if these extensions get linked in statically, which is basically supported but not available for all extensions.

Self contained applications

The solution to many of these problems is creating an application directory that contains the executable, the runtime libraries, extensions and additional support files needed by the program. The executable has to be linked specially to make sure the correct included runtime library is used. You do this by using the `-deploy` options provided by the compiler driver, `csc`:

```
% csc -deploy myprogram.scm
```

```
% ls -l myprogram
-rwxr-xr-x 1 felix felix 7972753 2010-02-22 20:19 libchicken.so.5
-rwxr-xr-x 1 felix felix   34839 2010-02-22 20:19 myprogram
% ldd myprogram
        linux-gate.so.1 =>  (0xb806a000)
        libchicken.so.5 => /home/felix/tmp/myprogram/libchicken.so.5 (0xb7c30000)
        libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7bfa000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7bf5000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7a92000)
        /lib/ld-linux.so.2 (0xb806b000)
```

As can be seen here, `myprogram` is prepared to load the contained `libchicken`, not any installed in the system that happens to have the same name.

You can even install extensions inside the application directory:

```
% chicken-install -deploy -p $PWD/myprogram defstruct
...
% ls -l myprogram
-rwxr-xr-x 1 felix felix   82842 2010-02-22 20:24 defstruct.import.so
-rw-r--r-- 1 felix felix     182 2010-02-22 20:24 defstruct.setup-info
-rwxr-xr-x 1 felix felix   11394 2010-02-22 20:24 defstruct.so
-rwxr-xr-x 1 felix felix 7972753 2010-02-22 20:19 libchicken.so.5
-rwxr-xr-x 1 felix felix   34839 2010-02-22 20:19 myprogram
```

We can check with ldd that those compiled extension libraries are linked with the correct library:

```
% ldd myprogram/*.so
/home/felix/tmp/myprogram/defstruct.import.so:
        linux-gate.so.1 =>  (0xb7f4f000)
        libchicken.so.5 => /home/felix/tmp/myprogram/libchicken.so.5 (0xb7b08000)
        libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7ad2000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7acd000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb796a000)
        /lib/ld-linux.so.2 (0xb7f50000)
/home/felix/tmp/myprogram/defstruct.so:
        linux-gate.so.1 =>  (0xb80c9000)
        libchicken.so.5 => /home/felix/tmp/myprogram/libchicken.so.5 (0xb7c8c000)
        libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7c56000)
        libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7c51000)
        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7aee000)
        /lib/ld-linux.so.2 (0xb80ca000)
```

The `-deploy` option passed to `csc` when compiling `myprogram.scm` has taken care of setting up the application directory as the "repository" for extensions that the program will use at runtime:

```
% myprogram/myprogram -:d
[debug] application startup...
[debug] heap resized to 500000 bytes
[debug] stack bottom is 0xbfdbdf60.
[debug] entering toplevel toplevel...
[debug] stack resized to 131072 bytes
[debug] entering toplevel library_toplevel...
[debug] entering toplevel eval_toplevel...
[debug] entering toplevel expand_toplevel...
[debug] loading compiled module `/home/felix/tmp/myprogram/defstruct.so' (handle i
...
```

There is one restriction that you should be aware of, though: any extension that you install inside an application directory must first be installed system-wide (unless you use a custom repository with the

`CHICKEN_REPOSITORY` environment variable), so that `csc/chicken` will find an import library for the extension. Just make sure you have all the extensions installed that you use in an application (something you will normally have anyway).

You can execute the program from its location, or you can install a symbolic link pointing to it - it will find the correct directory where the actual executable is located.

The application directory is fully "portable" in the sense that it will run directly from an USB-stick or any other removable media. At runtime the program can find out its location by invoking the `repository-path` procedure, which will return the full pathname in which the application is located.

Should the program depend on more libraries which are not available by default on the intended target systems, and which you would like to include in your application, you will have to hunt them down yourself and place them in the application directory. If these again have dependencies, things will get complicated and will involve things like patching binaries or writing "trampoline" shell scripts to run your application.

Deployment is fully compatible with "cross CHICKENs" (see Cross development).

Platform-specific notes

Linux

Deployment is fully supported on Linux

Windows

Deployment is fully supported on Windows. Since Windows looks up dynamic link libraries in the programs original location by default, adding third-party libraries to the application directory is no problem. The freely available Dependency Walker tool is helpful to find out what DLLs your application depends on.

MacOS X

On the Macintosh, passing the `-gui` option to `csc` will result in a true GUI application bundle (named `<your-program>.app`).

Invoking

```
% otool -L <yourprogram>
```

will list dynamic libraries that your application needs.

Other UNIX flavors

Setting up the application executable to load runtime libraries from the same directory is supported on FreeBSD, OpenBSD and Solaris. NetBSD supports this from version 5.0 onwards - this is currently disabled in `csc` for this particular platform.

Deploying source code

An alternative to deploying binaries is deployment as compiled C sources. Usually, you just need to ship your application code, compiled to `.c` files and the `chicken.h` and `runtime.c` files from the CHICKEN sources. You will also need the `.c` files of any library units your program uses. Compiling everything and linking it together should work on most systems. Consult the CHICKEN makefiles for more information about optimization options, etc.

Previous: Extensions

Next: Cross development

Cross Development

Since CHICKEN generates C code, it is relatively easy to create programs and libraries for a different architecture than the one the compiler is executing on, a process commonly called *cross compiling*. Basically you can simply compile Scheme code to C and then invoke your target-specific cross compiler. To automate the process of invoking the correct C compiler with the correct settings and to simplify the use of extensions, CHICKEN can be built in a special "cross-compilation" mode.

Note: in the following text we refer to the "target" as being the platform on which the software is intended to run in the end. We use the term "host" as the system that builds this software. Others use a different nomenclature or switch the meaning of the words.

Preparations

Make sure you have a cross-toolchain in your `PATH`. In this example, a Linux system is used to generate binaries for an ARM based embedded system.

Building the target libraries

First you need a version of the runtime system (`libchicken`), compiled for the target system. Obtain and unpack a tarball of the CHICKEN sources, or check out the code from the official code repository, then build the libraries and necessary development files:

```
make ARCH= \
    PREFIX=/usr \
    PLATFORM=linux
    HOSTSYSTEM=arm-none-linux-gnueabi \
    DESTDIR=$HOME/target \
    TARGET_FEATURES="-no-feature x86 -feature arm" \
    libs install-dev
```

This will build the CHICKEN libraries and install them in `~/target`, which we use as a temporary place to store the target files. A few things to note:

- `ARCH` is empty, since we don't want the build process to detect the architecture (since the target-architecture is likely to be different).
- `PREFIX` gives the prefix *on the target system*, under which the libraries will finally be installed. In this case it will be `/usr/lib`.
- `PLATFORM` determines the target platform. It must be one of the officially supported platforms CHICKEN runs on.
- `HOSTSYSTEM` is an identifier for the target system and will be used as the name prefix of the cross C compiler (in this case `arm-none-linux-gnueabi-gcc`). If your cross compiler does not follow this convention, pass `C_COMPILER` and `LIBRARIAN` to the `make(1)` invocation, with the names of the C

compiler and `ar(1)` tool, respectively.

- `DESTDIR` holds the directory where the compiled library files will temporarily installeds into.
- `TARGET_FEATURES` contains extra options to be passed to the target-specific Scheme translator; in this case we disable and enable features so that code like the following will do the right thing when cross-compiled:

```
(cond-expand
  (x86 <do this ...>)
  ...)
```

- If you obtained the sources from a source-code repository and not from an official release tarball, you will need a `chicken` executable to compile the Scheme sources of the runtime system. In this case pass yet another variable to the `make(1)` invocation: `CHICKEN=<where the "chicken" executable is`.
- You can also put all those variables into a file, say `config.mk` and run `make CONFIG=config.mk`.

You should now have these files on `~/target`:

```
 `-- usr
   |-- include
   |   |-- chicken-config.h
   |   `-- chicken.h
   |-- lib
   |   |-- chicken
   |   |   `-- 5
   |   |       `-- types.db
   |   |-- libchicken.a
   |   `-- libchicken.so
   `-- share
```

You should now transfer `libchicken.so` to the target system, and place it in `/usr`.

Building the "cross chicken"

Next, we will build another chicken, one that uses the cross C compiler to generate target-specific code that uses the target-specific runtime library we have just built.

Again, unpack a CHICKEN release tarball or a source tree and run `make(1)` once again:

```
make PLATFORM=linux \
    PREFIX=$HOME/cross-chicken \
    TARGETSYSTEM=arm-none-linux-gnueabi \
    PROGRAM_PREFIX=arm- \
    TARGET_PREFIX=$HOME/target/usr \
    TARGET_RUN_PREFIX=/usr \
    install
```

- `PREFIX` gives the place where the "cross chicken" should be installed into. It is recommended not to install into a standard location (like `/usr/local` or `$HOME`) - some files will conflict with a normal CHICKEN installation.
- `TARGETSYSTEM` gives the name-prefix of the cross C compiler.
- `PROGRAM_PREFIX` determines the name-prefix of the CHICKEN tools to be created.
- `TARGET_PREFIX` specifies where the target-specific files (libraries and headers) are located. This is the location where we installed the runtime system into.
- `TARGET_RUN_PREFIX` holds the PREFIX that will be effective at runtime (so `libchicken.so` will be found in `$TARGET_RUN_PREFIX/lib`).
- Make sure to use the same version of the CHICKEN sources for the target and the cross build.
- If you build the cross chicken from repository sources, the same note about the `CHICKEN` variable applies as given above.

In `~/cross-chicken`, you should find the following:

```
|-- bin
|   |-- arm-chicken
|   |-- arm-chicken-bug
|   |-- arm-chicken-install
|   |-- arm-chicken-profile
|   |-- arm-chicken-status
|   |-- arm-chicken-uninstall
|   |-- arm-csc
|   `-- arm-csi
|-- include
|   |-- chicken-config.h
|   `-- chicken.h
|-- lib
|   |-- chicken
|   |   `-- 5
|   |       :
|   |
|   |-- libchicken.a
|   |-- libchicken.so -> libchicken.so.5
|   `-- libchicken.so.5
`-- share
    |-- chicken
    |   |-- doc
    :   ;   :
    |   |
    |   `-- setup.defaults
    `-- man
        `-- man1
            :
```

To make sure that the right C compiler is used, we ask `arm-csc` to show the name of the cross C compiler:

```
% ~/cross-chicken/arm-csc -cc-name
arm-none-linux-gnueabi-gcc
```

Looks good.

Using it

Compiling simple programs

```
% ~/cross-chicken/arm-csc -v hello.scm
/home/felix/cross-chicken/arm-cross-chicken/bin/arm-chicken hello.scm -output-file
arm-none-linux-gnueabi-gcc hello.c -o hello.o -c -fno-strict-aliasing -DHAVE_CHICK
  -Wno-unused -I /home/felix/cross-chicken/arm-chicken/include
rm hello.c
arm-none-linux-gnueabi-gcc hello.o -o hello -L/home/felix/cross-chicken/arm-chicke
  -ldl -lchicken
rm hello.o
```

Is it an ARM binary?

```
% file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for GNU/Linux 2.6.16, dyr
```

Yes, looks good.

Compiling extensions

By default, the tools that CHICKEN provides to install, list and uninstall extensions will operate on both the host and the target repository. So running `arm-chicken-install` will compile and install the extension for the host system and for the cross-target. To selectively install, uninstall or list extensions for either the host or the target system use the `-host` and `-target` options for the tools.

"Target-only" extensions

Sometimes an extension will only be compilable for the target platform (for example libraries that use system-dependent features). In this case you will have to work around the problem that the host-compiler still may need compile-time information from the target-only extension, like the import library of modules. One option is to copy the import-library into the repository of the host compiler:

```
# optionally, you can compile the import library:
# ~/cross-chicken/arm-csc -O3 -d0 -s target-only-extension.import.scm
cp target-only-extension.import.scm ~/cross-chicken/lib/chicken/5
```

Final notes

Cross-development is a very tricky process - it often involves countless manual steps and it is very easy to forget an important detail or mix up target and host systems. Also, full 100% platform neutrality is hard to achieve. CHICKEN tries very hard to make this transparent, but at the price of considerable complexity in the code that manages extensions.

Previous: Deployment Next: Data representation

Data representation

There exist two different kinds of data objects in the CHICKEN system: immediate and non-immediate objects.

Immediate objects

Immediate objects are represented by a single machine word, 32 or 64 bits depending on the architecture. They come in four different flavors:

fixnums, that is, small exact integers, where the lowest order bit is set to 1. This gives fixnums a range of 31 bits for the actual numeric value (63 bits on 64-bit architectures).

characters, where the four lowest-order bits are equal to `C_CHARACTER_BITS`, currently 1010. The Unicode code point of the character is encoded in the next 24 bits.

booleans, where the four lowest-order bits are equal to `C_BOOLEAN_BITS`, currently 0110. The next bit is one for #t and zero for #f.

other values: the empty list, the value of unbound identifiers, the undefined value (void), and end-of-file. The four lowest-order bits are equal to `C_SPECIAL_BITS`, currently 1110. The next four bits contain an identifying number for this type of object, one of: `C_SCHEME_END_OF_LIST`, currently 0000; `C_SCHEME_UNDEFINED`, currently 0001; `C_SCHEME_UNBOUND`, currently 0010; or `C_SCHEME_END_OF_FILE`, currently 0011.

Non-immediate objects

Collectively, the two lowest-order bits are known as the *immediate mark bits*. When the lowest bit is set, the object is a fixnum, as described above, and the next bit is part of its value. When the lowest bit is clear but the next bit is set, it is an immediate object other than a fixnum. If neither bit is set, the object is non-immediate, as described below.

Non-immediate objects are blocks of data represented by a pointer into the heap. The pointer's immediate mark bits must be zero to indicate the object is non-immediate; this guarantees the data block is aligned on a 4-byte boundary, at minimum. Alignment of data words is required on modern architectures anyway, so we get the ability to distinguish between immediate and non-immediate objects for free.

The first word of the data block contains a header, which gives information about the type of the object. The header is a single machine word.

The 24 lowest-order bits contain the length of the data object, which is either the number of bytes in a string or byte-vector, or the the number of elements for a vector or record type.

The remaining bits are placed in the high-order end of the header. The four highest-order bits are used for garbage collection or internal data type dispatching.

C_GC_FORWARDING_BIT
: Flag used for forwarding garbage collected object pointers.

C_BYTEBLOCK_BIT
: Flag that specifies whether this data object contains raw bytes (a string or byte-vector) or pointers to other data objects.

C_SPECIALBLOCK_BIT
Flag that specifies whether this object contains a *special* non-object pointer value in its first slot. An example for this kind of objects are closures, which are a vector-type object with the code-pointer as the first item.

C_8ALIGN_BIT
Flag that specifies whether the data area of this block should be aligned on an 8-byte boundary (floating-point numbers, for example).

After these four bits comes a 4-bit type code representing one of the following types:

vectors: vector objects with type bits `C_VECTOR_TYPE`, currently 0000.

symbols: vector objects with type bits `C_SYMBOL_TYPE`, currently 0001. The three slots contain the toplevel variable value, the print-name (a string), and the property list of the symbol.

strings: byte-vector objects with type bits `C_STRING_TYPE`, currently 0010.

pairs: vector-like object with type bits `C_PAIR_TYPE`, currently 0011). The car and the cdr are contained in the first and second slots, respectively.

closures: special vector objects with type bits `C_CLOSURE_TYPE`, currently 0100. The first slot contains a pointer to a compiled C function. Any extra slots contain the free variables (since a flat closure representation is used).

flonums: byte-vector objects with type bits `C_FLONUM_BITS`, currently 0101. Slots one and two (or a single slot on 64 bit architectures) contain a 64-bit floating-point number, in the representation used by the host systems C compiler.

ports: special vector objects with type bits `C_PORT_TYPE`, currently 0111. The first slot contains a pointer to a file- stream, if this is a file-pointer, or NULL if not. The other slots contain housekeeping data used for this port.

structures: vector objects with type bits `C_STRUCTURE_TYPE`, currently 1000. The first slot contains a symbol that specifies the kind of structure this record is an instance of. The other slots contain the actual record items.

pointers: special vector objects with type bits `C_POINTER_TYPE`, currently 1001. The single slot contains a machine pointer.

locatives: special vector objects with type bits `C_LOCATIVE_TYPE`, currently 1010. FIXME FIXME FIXME.

tagged pointers: special vector objects with type bits `C_TAGGED_POINTER_TYPE`, currently 1011, Tagged pointers are similar to pointers, but the object contains an additional slot with a tag (an arbitrary data object) that identifies the type of the pointer.

SWIG pointers: special vector objects with type bits `C_SWIG_POINTER_TYPE`, currently 1100.

lambda infos: byte-vector objects with type-bits `C_LAMBDA_INFO_TYPE`, currently 1101.

buckets: vector objects with type-bits `C_BUCKET_TYPE`, currently 1111.

The actual data follows immediately after the header. Note that block addresses are always aligned to the native machine-word boundary.

Data objects may be allocated outside of the garbage collected heap, as long as their layout follows the above mentioned scheme. But care has to be taken not to mutate these objects with heap-data (i.e. non-immediate objects), because this will confuse the garbage collector.

For more information see the header file `chicken.h`.

Previous: Cross development

Next: Bugs and limitations

Bugs and limitations

- Compiling large files takes too much time.
- If a known procedure has unused arguments, but is always called without those parameters, then the optimizer *repairs* the procedure in certain situations and removes the parameter from the lambda-list.
- `port-position` currently works only for input ports.
- Leaf routine optimization can theoretically result in code that thrashes, if tight loops perform excessively many mutations.

Previous: Data representation

Next: FAQ

FAQ

This is the list of Frequently Asked Questions about Chicken Scheme. If you have a question not answered here, feel free to post to the chicken-users mailing list; if you consider your question general enough, feel free to add it to this list.

General

Why yet another Scheme implementation?

Since Scheme is a relatively simple language, a large number of implementations exist and each has its specific advantages and disadvantages. Some are fast, some provide a rich programming environment. Some are free, others are tailored to specific domains, and so on. The reasons for the existence of CHICKEN are:

- CHICKEN is portable because it generates C code that runs on a large number of platforms.
- CHICKEN is extendable, since its code generation scheme and runtime system/garbage collector fits neatly into a C environment.
- CHICKEN is free and can be freely distributed, including its source code.
- CHICKEN offers better performance than nearly all interpreter based implementations, but still provides full Scheme semantics.
- As far as we know, CHICKEN is the first implementation of Scheme that uses Henry Baker's Cheney on the M.T.A concept.

What should I do if I find a bug?

Fill a ticket at bugs.call-cc.org with some hints about the problem, like version/build of the compiler, platform, system configuration, code that causes the bug, etc.

Specific

Why are values defined with `define-foreign-variable` or `define-constant` or `define-inline` not seen outside of the containing source file?

Accesses to foreign variables are translated directly into C constructs that access the variable, so the Scheme name given to that variable does only exist during compile-time. The same goes for constant- and inline-definitions: The name is only there to tell the compiler that this reference is to be replaced with the actual value.

How does `cond-expand` know which features are registered in used units?

Each unit used via `(declare (uses ...))` is registered as a feature and so a symbol with the unit-name can be tested by `cond-expand` during macro-expansion-time. Features registered using the `register-feature!` procedure are only available during run-time of the compiled file. You can use the `eval-when` form to register features at compile time.

Why are constants defined by `define-constant` not honoured in `case` constructs?

`case` expands into a cascaded `if` expression, where the first item in each arm is treated as a quoted list. So the `case` macro can not infer whether a symbol is to be treated as a constant-name (defined via `define-constant`) or a literal symbol.

How can I enable case sensitive reading/writing in user code?

To enable the `read` procedure to read symbols and identifiers case sensitive, you can set the parameter `case-sensitivity` to `#t`.

Why doesn't CHICKEN support the full numeric tower by default?

The short answer:

```
% chicken-install numbers
% csi -q
#;1> (use numbers)
```

The long answer:

There are a number of reasons for this:

- For most applications of Scheme fixnums (exact word-sized integers) and flonums (64-bit floating-point numbers) are more than sufficient;

- Interfacing to C is simpler;

- Dispatching of arithmetic operations is more efficient.

There is an extension based on the GNU Multiprecision Package that implements most of the full numeric tower, see numbers.

Does CHICKEN support native threads?

Native threads are not supported for two reasons. One, the runtime system is not reentrant. Two, concurrency implemented properly would require mandatory locking of every object that could be potentially shared between two threads. The garbage-collection algorithm would then become much more complex and inefficient, since the location of every object has to be accessed via a thread synchronization protocol. Such a design would make native threads in Chicken essentially equivalent to Unix processes and shared memory.

For a different approach to concurrency, please see the mpi egg.

Does CHICKEN support Unicode strings?

The system does not directly support Unicode, but there is an extension for UTF-8 strings: utf8.

Why are `dynamic-wind' thunks not executed when a SRFI-18 thread signals an error?

Here is what Marc Feeley, the author of SRFI-18 has to say about this subject:

```
    >No the default exception handler shouldn't invoke the after
    > thunks of the current continuation.  That's because the
    > exception handler doesn't "continue" at the initial
    > continuation of that thread.  Here are the relevant words of
    > SRFI 18:

    >
    >  Moreover, in this dynamic environment the exception handler
    >  is bound to the "initial exception handler" which is a unary
    >  procedure which causes the (then) current thread to store in
    >  its end-exception field an "uncaught exception" object whose
    >  "reason" is the argument of the handler, abandon all mutexes
```

```
>  it owns, and finally terminate.
>

>The rationale is that, when an uncaught exception occurs in a
>thread the thread is in bad shape and things have gone
>sufficiently wrong that there is no universally acceptable way to
>continue execution.  Executing after thunks could require a
>whole lot of processing that the thread is not in a shape to do.
>So the safe thing is to terminate the thread.  If the programmer
>knows how to recover from an exception, then he can capture the
>continuation early on, and install an exception handler which
>invokes the continuation.  When the continuation is invoked the
>after thunks will execute.
```

Platform specific

How do I generate a DLL under MS Windows (tm) ?

Use `csc` in combination with the `-dll` option:

```
C:\> csc foo.scm -dll
```

How do I generate a GUI application under Windows(tm)?

Invoke `csc` with the `-gui` option. In GUI-mode, the runtime system displays error messages in a message box and does some rudimentary command-line parsing.

Compiling very large files under Windows with the Microsoft C compiler fails with a message indicating insufficient heap space.

It seems that the Microsoft C compiler can only handle files up to a certain size, and it doesn't utilize virtual memory as well as the GNU C compiler, for example. Try closing running applications. If that fails, try to break up the Scheme code into several library units.

When I run `csi` inside an emacs buffer under Windows, nothing happens.

Invoke `csi` with the `-:c` runtime option. Under Windows the interpreter thinks it is not running under control of a terminal and doesn't print the prompt and does not flush the output stream properly.

On Windows, `csc.exe` seems to be doing something wrong.

The Windows development tools include a C# compiler with the same name. Either invoke `csc.exe` with a full pathname, or put the directory where you installed CHICKEN in front of the MS development tool path in the `PATH` environment variable.

On Windows source and/or output filenames with embedded whitespace are not found.

There is no current workaround. Do not use filenames with embedded whitespace for code. However, command names with embedded whitespace will work correctly.

Customization

How do I run custom startup code before the runtime-system is invoked?

When you invoke the C compiler for your translated Scheme source program, add the C compiler option `-DC_EMBEDDED`, or pass `-embedded` to the `csc` driver program, so no entry-point function will be generated (`main()`). When your are finished with your startup processing, invoke:

```
CHICKEN_main(argc, argv, C_toplevel);
```

where `C_toplevel` is the entry-point into the compiled Scheme code. You should add the following declarations at the head of your code:

```
#include "chicken.h"
extern void C_toplevel(C_word,C_word,C_word) C_noret;
```

How can I add compiled user passes?

To add a compiled user pass instead of an interpreted one, create a library unit and recompile the main unit of the compiler (in the file `chicken.scm`) with an additional `uses` declaration. Then link all compiler modules and your (compiled) extension to create a new version of the compiler, like this (assuming all sources are in the current directory):

```
  % cat userpass.scm
  ;;;; userpass.scm - My very own compiler pass

  (declare (unit userpass))

  ;; Perhaps more user passes/extensions are added:
  (let ([old (user-pass)])
    (user-pass
      (lambda (x)
        (let ([x2 (do-something-with x)])
    (if old
        (old x2)
        x2) ) ) ) ) )
```

```
% csc -c -x userpass.scm
% csc chicken.scm -c -o chicken-extended.o -uses userpass
% gcc chicken-extended.o support.o easyffi.o compiler.o optimizer.o batch-driver.o
c-backend.o userpass.o `csc -ldflags -libs` -o chicken-extended
```

On platforms that support it (Linux ELF, Solaris, Windows + VC++), compiled code can be loaded via `-extend` just like source files (see `load` in the User's Manual).

Macros

Where is `define-macro`?

With CHICKEN 4, the macro-expansion subsystem is now hygienic where old Lisp-style low-level macros are not available anymore. `define-syntax` can define hygienic macros using `syntax-rules` or low-level macros with user-controlled hygienic with *explicit renaming* macros. Translating old-style macros into ER-macros isn't that hard, see Macros for more information.

Why are low-level macros defined with `define-syntax` complaining about unbound variables?

Macro bodies that are defined and used in a compiled source-file are evaluated during compilation and so have no access to anything created with `define`. Use `define-for-syntax` instead.

Why isn't `load` properly loading my library of macros?

During compile-time, macros are only available in the source file in which they are defined. Files included via `include` are considered part of the containing file.

Warnings and errors

Why does my program crash when I use callback functions (from Scheme to C and back to Scheme again)?

There are two reasons why code involving callbacks can crash out of no apparent reason:

1. It is important to use `foreign-safe-lambda/foreign-safe-lambda*` for the C code that is to call back into Scheme. If this is not done than sooner or later the available stack space will be exhausted.
2. If the C code uses a large amount of stack storage, or if Scheme-to-C-to-Scheme calls are nested deeply, then the available nursery space on the stack will run low. To avoid this it might be advisable to run the compiled code with a larger nursery setting, i.e. run the code with `-:s...` and a larger value than the default (for example `-:s300k`), or use the `-nursery` compiler option. Note that this can decrease runtime performance on some platforms.

Why does the linker complain about a missing function `_C_..._toplevel`?

This message indicates that your program uses a library-unit, but that the object-file or library was not supplied to the linker. If you have the unit `foo`, which is contained in `foo.o` than you have to supply it to the linker like this (assuming a GCC environment):

```
% csc program.scm foo.o -o program
```

Why does the linker complain about a missing function `_C_toplevel`?

This means you have compiled a library unit as an application. When a unit-declaration (as in `(declare (unit ...))`) is given, then this file has a specially named toplevel entry procedure. Just remove the declaration, or compile this file to an object-module and link it to your application code.

Why does my program crash when I compile a file with `-unsafe` or unsafe declarations?

The compiler option `-unsafe` or the declaration `(declare (unsafe))` disable certain safety-checks to improve performance, so code that would normally trigger an error will work unexpectedly or even crash the running application. It is advisable to develop and debug a program in safe mode (without unsafe declarations) and use this feature only if the application works properly.

Why don't toplevel-continuations captured in interpreted code work?

Consider the following piece of code:

```
(define k (call-with-current-continuation (lambda (k) k)))
(k k)
```

When compiled, this will loop endlessly. But when interpreted, `(k k)` will return to the read-eval-print loop! This happens because the continuation captured will eventually read the next toplevel expression from the standard-input (or an input-file if loading from a file). At the moment `k` was defined, the next expression was `(k k)`. But when `k` is invoked, the next expression will be whatever follows after `(k k)`. In other words, invoking a captured continuation will not rewind the file-position of the input source. A solution is to wrap the whole code into a `(begin ...)` expression, so all toplevel expressions will be loaded together.

Why does `define-reader-ctor` not work in my compiled program?

The following piece of code does not work as expected:

```
 (eval-when (compile)
 (define-reader-ctor 'integer->char integer->char) )
```

```
(print #,(integer->char 33))
```

The problem is that the compiler reads the complete source-file before doing any processing on it, so the sharp-comma form is encountered before the reader-ctor is defined. A possible solution is to include the file containing the sharp-comma form, like this:

```
(eval-when (compile)
(define-reader-ctor 'integer->char integer->char) )

(include "other-file")
```

```
;;; other-file.scm:
(print #,(integer->char 33))
```

Why do built-in units, such as srfi-1, srfi-18, and posix fail to load?

When you try to `use` a built-in unit such as `srfi-18`, you may get the following error:

```
#;1> (use srfi-18)
; loading library srfi-18 ...
Error: (load-library) unable to load library
srfi-18
"dlopen(libchicken.dylib, 9): image not found"              ;; on a Mac
"libchicken.so: cannot open shared object file: No such file or directory"  ;; Lin
```

Another symptom is that `(require 'srfi-18)` will silently fail.

This typically happens because the Chicken libraries have been installed in a non-standard location, such as your home directory. The workaround is to explicitly tell the dynamic linker where to look for your libraries:

```
export DYLD_LIBRARY_PATH=~/scheme/chicken/lib:$DYLD_LIBRARY_PATH ;; Mac
export LD_LIBRARY_PATH=~/scheme/chicken/lib:$LD_LIBRARY_PATH    ;; Linux
```

How can I increase the size of the trace shown when runtime errors are detected?

When a runtime error is detected, Chicken will print the last entries from the trace of functions called (unless your executable was compiled with the `-no-trace` option. By default, only 16 entries will be shown. To increase this number pass the `-:aN` parameter to your executable.

Optimizations

How can I obtain smaller executables?

If you don't need `eval` or the stuff in the `extras` library unit, you can just use the `library` unit:

```
(declare (uses library))
(display "Hello, world!\n")
```

(Don't forget to compile with the `-explicit-use` option) Compiled with Visual C++ this generates an executable of around 240 kilobytes. It is theoretically possible to compile something without the library, but a program would have to implement quite a lot of support code on its own.

How can I obtain faster executables?

There are a number of declaration specifiers that should be used to speed up compiled files: declaring `(standard-bindings)` is mandatory, since this enables most optimizations. Even if some standard procedures should be redefined, you can list untouched bindings in the declaration. Declaring `(extended-bindings)` lets the compiler choose faster versions of certain internal library functions. This might give another speedup. You can also use the the `usual-integrations` declaration, which is identical to declaring `standard-bindings` and `extended-bindings` (note that `usual-integrations` is set by default). Declaring `(block)` tells the compiler that global procedures are not changed outside the current compilation unit, this gives the compiler some more opportunities for optimization. If no floating point arithmetic is required, then declaring `(number-type fixnum)` can give a big performance improvement, because the compiler can now inline most arithmetic operations. Declaring `(unsafe)` will switch off most safety checks. If threads are not used, you can declare `(disable-interrupts)`. You should always use maximum optimizations settings for your C compiler. Good GCC compiler options on Pentium (and compatible) hardware are: `-Os -fomit-frame-pointer -fno-strict-aliasing` Some programs are very sensitive to the setting of the nursery (the first heap-generation). You should experiment with different nursery settings (either by compiling with the `-nursery` option or by using the `-:s...` runtime option).

Which non-standard procedures are treated specially when the `extended-bindings` or `usual-integrations` declaration or compiler option is used?

The following standard bindings are handled specially, depending on optimization options and compiler settings:

`+ * - / quotient eq? eqv? equal? apply c...r values call-with-values list-ref null? length not char? string? symbol? vector? pair? procedure? boolean? number? complex? rational? real? exact? inexact? list? eof-object? string-ref string-set! vector-ref vector-set! char=? char<? char>? char<=? char>=? char-numeric? char-alphabetic? char-whitespace? char-upper-case? for-each char-lower-case? char-upcae char-downcase list-tail assv memv memq assoc member set-car! set-cdr! abs exp sin cos tan log asin acos atan sqrt zero? positive? negative? vector-length string-length char->integer integer->char inexact->exact = > < >= <= for-each map substring string-append gcd lcm list exact->inexact string->number number->string even? odd? remainder floor ceiling truncate round cons vector string string=? string-ci=? make-vector call-with-current-continuation write-char read-string`

The following extended bindings are handled specially:

`bitwise-and bitwise-ior bitwise-xor bitwise-not bit-set? add1 sub1 fx+ fx- fx* fx/ fxmod fx= fx> fx>= fixnum? fxneg fxmax fxmin fxodd? fxeven? fxand fxior fxxor fxnot fxshl fxshr finite? fp= fp> fp< fp>= fp<= fpinteger? flonum? fp+ fp- fp* fp/ atom? fp= fp> fp>= fpneg fpmax fpmin fpfloor fpceiling fpround fptruncate fpsqrt fpabs fplog fpexp fpexpt fpsin fpcos fptan fpasin fpacos fpatan fpatan2 arithmetic-shift signum flush-output thread-specific thread-specific-set! not-pair? null-list? print print* u8vector->blob/shared s8vector->blob/shared u16vector->blob/shared s16vector->blob/shared u32vector->blob/shared s32vector->blob/shared f32vector->blob/shared f64vector->blob/shared block-ref blob-size u8vector-length s8vector-length u16vector-length s16vector-length u32vector-length s32vector-length f32vector-length f64vector-length u8vector-ref s8vector-ref u16vector-ref s16vector-ref u32vector-ref s32vector-ref f32vector-ref f64vector-ref u8vector-set! s8vector-set! u16vector-set! s16vector-set! u32vector-set! s32vector-set! hash-table-ref block-set! number-of-slots first second third fourth null-pointer? pointer->object pointer+ pointer=? pointer-u8-ref pointer-s8-ref pointer-u16-ref pointer-s16-ref pointer-u32-ref pointer-s32-ref pointer-f32-ref pointer-f64-ref pointer-u8-set! pointer-s8-set! pointer-u16-set! pointer-s16-set! pointer-u32-set! pointer-s32-set! pointer-f32-set! pointer-f64-set! make-record-instance locative-ref locative-set! locative? locative->object identity cpu-time error call/cc any? substring=? substring-ci=? substring-index substring-index-ci printf sprintf fprintf format o`

What's the difference betweem "block" and "local" mode?

In `block` mode, the compiler assumes that definitions in the current file are not visible from outside of the current compilation unit, so unused definitions can be removed and calls can be inlined. In `local` mode, definitions are not hidden, but the compiler assumes that they are not modified from other compilation units (or code evaluated at runtime), and thus allows inlining of them.

Can I load compiled code at runtime?

Yes. You can load compiled at code at runtime with `load` just as well as you can load Scheme source code. Compiled code will, of course, run faster.

To do this, pass to `load` a path for a shared object. Use a form such as `(load "foo.so")` and run `csc -shared foo.scm` to produce `foo.so` from `foo.scm` (at which point `foo.scm` will no longer be required).

If you have compiled code that contains a `module` definition, then executing the code will "register" the module to allow importing the bindings provided by the module into a running Scheme process. The information required to use a module is in this case embedded in the compiled code. Compiling another program that uses this (compiled) module is more difficult: the used module will not necessarily be loaded into the compiler, so the registration will not be executed. In this case the information about what bindings the compiled module exports must be separated from the actual code that executes at runtime. To make this possible, compiling a module can be done in such a manner that an "import library" is created. This is a file that contains the binding information of the module and we can use it to compile a file that refers to that module. An example can perhaps make this clearer:

```
;; my-module.scm

(module my-module (...) ...)
```

```
;; use-my-module.scm
```

```
(import my-module)
...
```

Compile the module and generate an import library for the "my-module" module:

```
% csc -s my-module.scm -emit-import-library my-module
```

Compile the program that uses the module:

```
% csc use-my-module.scm
```

Why is my program which uses regular expressions so slow?

The regular expression engine has recently be replaced by alex shinn's excellent `irregex` library, which is fully implemented in Scheme. Precompiling regular expressions to internal form is somewhat slower than with the old PCRE-based regex engine. It is advisable to use `regexp` to precompile regular expressions outside of time-critical loops and use them where performance matters.

Garbage collection

Why does a loop that doesn't `cons` still trigger garbage collections?

Under CHICKENs implementation policy, tail recursion is achieved simply by avoiding to return from a function call. Since the programs are CPS converted, a continuous sequence of nested procedure calls is performed. At some stage the stack-space has to run out and the current procedure and its parameters (including the current continuation) are stored somewhere in the runtime system. Now a minor garbage collection occurs and rescues all live data from the stack (the first heap generation) and moves it into the the second heap generation. Then the stack is cleared (using a `longjmp`) and execution can continue from the saved state. With this method arbitrary recursion (in tail- or non-tail position) can happen, provided the application doesn't run out of heap-space. (The difference between a tail- and a non-tail call is that the tail-call has no live data after it invokes its continuation - and so the amount of heap-space needed stays constant)

Why do finalizers not seem to work in simple cases in the interpeter?

Consider the following interaction in CSI:

```
#;1> (define x '(1 2 3))
#;2> (define (yammer x) (print x " is dead"))
#;3> (set-finalizer! x yammer)
(1 2 3)
#;4> (gc #t)
157812
#;5> (define x #f)
#;6> (gc #t)
157812
#;7>
```

While you might expect objects to be reclaimed and "*(1 2 3) is dead*" printed, it won't happen: the literal list gets held in the interpreter history, because it is the result value of the set-finalizer! call. Running this in a normal program will work fine.

When testing finalizers from the interpreter, you might want to define a trivial macro such as

```
(define-syntax v
  (syntax-rules ()
    ((_ x) (begin (print x) (void)))))
```

and wrap calls to `set-finalizer!` in it.

Interpreter

Does CSI support history and autocompletion?

CSI doesn't support it natively but it can be activated with the readline egg. After installing the egg, add the following to your `~/.csirc` or equivalent file:

```
 (require-extension readline)
 (current-input-port (make-gnu-readline-port))
 (gnu-history-install-file-manager (string-append (or (getenv "HOME") ".") "/.csi.h
```

Users of *nix-like systems (including Cygwin), may also want to check out rlwrap. This program lets you "wrap" another process (e.g. `rlwrap csi`) with the readline library, giving you history, autocompletion, and the ability to set the keystroke set. Vi fans can get vi keystrokes by adding "set editing-mode vi" to their `.inputrc` file.

Does code loaded with `load` run compiled or interpreted?

If you compile a file with a call to `load`, the code will be loaded at runtime and, if the file loaded is a Scheme source code file (instead of a shared object), it will be interpreted (even if the caller program is compiled).

How do I use extended (non-standard) syntax in evaluated code at run-time?

Normally, only standard Scheme syntax is available to the evaluator. To use the extensions provided in the CHICKEN compiler and interpreter, add:

```
(require-library chicken-syntax)
```

Extensions

Where is "chicken-setup" ?

`chicken-setup` has been rewritten from scratch and its functionality is now contained in the three tools `chicken-install`, `chicken-uninstall` and `chicken-status`. See the Extensions chapter for more information.

How can I install Chicken eggs to a non-default location?

You can just set the `CHICKEN_REPOSITORY` environment variable. It should contain the path where you want eggs to be installed:

```
$ export CHICKEN_REPOSITORY=~/eggs/lib/chicken/5
$ chicken-install -init ~/eggs/lib/chicken/5
$ chicken-install -p ~/eggs/ extensionname
```

In order to make programs (including csi) see these eggs, you should set this variable when you run them. See the Extensions/Changing repository location section of the manual for more information on that.

Alternatively, you can call the `repository-path` Scheme procedure before loading the eggs, as in:

```
(repository-path "/home/azul/eggs")
(use format-modular)
```

Note, however, that using `repository-path` as above hard-codes the location of your eggs in your source files. While this might not be an issue in your case, it might be safe to keep this configuration outside of the source code (that is, specifying it as an environment variable) to make it easier to maintain.

The repository needs to be initialized before use. See the documentation for the `-init` option to `chicken-install`, in Extensions.

Can I install chicken eggs as a non-root user?

Yes, just install them in a directory you can write to by using `CHICKEN_REPOSITORY` (see above).

Why does downloading an extension via `chicken-install` fail on Windows Vista?

Possibly the Windows Firewall is active, which prevents `chicken-install` from opening a TCP connection to the egg repository. Try disabling the firewall temporarily.

Previous: Bugs and limitations

Next: Acknowledgements

Acknowledgements

Many thanks to Nico Amtsberg, Alonso Andres, William Annis, Marc Baily, Peter Barabas, Jonah Beckford, Arto Bendiken, Kevin Beranek, Peter Bex, Jean-Francois Bignolles, Oivind Binde, Alaric Blagrave-Snellpym, Dave Bodenstab, Fabian Boehlke, T. Kurt Bond, Ashley Bone, Dominique Boucher, Terence Brannon, Roy Bryant, Adam Buchbinder, Hans Bulfone, Category 5, Taylor Campbell, Naruto Canada, Mark Carter, Esteban U. Caamano Castro, Semih Cemiloglu, Franklin Chen, Thomas Chust, Gian Paolo Ciceri, Fulvio Ciriaco, Tobia Conforto, John Cowan, Grzegorz Chrupala, James Crippen, Tollef Fog Heen, Drew Hess, Alejandro Forero Cuervo, Peter Danenberg, Linh Dang, Brian Denheyer, Sean D'Epagnier, "dgym", "Don", Chris Double, Brown Dragon, Jarod Eells, Petter Egesund, Stephen Eilert, Steve Elkins, Daniel B. Faken, Will Farr, Graham Fawcett, Marc Feeley, "Fizzie", Matthew Flatt, Kimura Fuyuki, Tony Garnock-Jones, Martin Gasbichler, Abdulaziz Ghuloum, Joey Gibson, Stephen C. Gilardi, Mario Domenech Goulart, Joshua Griffith, Johannes Groedem, Damian Gryski, Andreas Gustafsson, Sven Hartrumpf, Jun-ichiro itojun Hagino, Ahdi Hargo, Matthias Heiler, Karl M. Hegbloom, William P. Heinemann, Bill Hoffman, Bruce Hoult, Hans Huebner, Markus Huelsmann, Goetz Isenmann, Paulo Jabardo, Wietse Jacobs, David Janssens, Christian Jaeger, Matt Jones, Dale Jordan, Valentin Kamyshenko, Daishi Kato, Peter Keller, Brad Kind, Ron Kneusel, Matthias Koeppe, Krysztof Kowalczyk, Andre Kuehne, Todd R. Kueny Sr, Goran Krampe, David Krentzlin, Ben Kurtz, Micky Latowicki, John Lenz, Kirill Lisovsky, Juergen Lorenz, Kon Lovett, Lam Luu, Vitaly Magerya, Leonardo Valeri Manera, Dennis Marti, Charles Martin, Bob McIsaac, Alain Mellan, Eric Merrit, Perry Metzger, Scott G. Miller, Mikael, Bruce Mitchener, Fadi Moukayed, Chris Moline, Eric E. Moore, Julian Morrison, Dan Muresan, David N. Murray, "nicktick", Lars Nilsson, Ian Oversby, "o.t.", Gene Pavlovsky, Levi Pearson, Jeronimo Pellegrini, Nicolas Pelletier, Derrell Piper, Carlos Pita, Robin Lee Powell, "Pupeno", Davide Puricelli, "presto", Doug Quale, Imran Rafique, Eric Raible, Ivan Raikov, Joel Reymont, Chris Roberts, Eric Rochester, Paul Romanchenko, Andreas Rottman, David Rush, Lars Rustemeier, Daniel Sadilek, Oskar Schirmer, Burton Samograd, Reed Sheridan, Ronald Schroeder, Spencer Schumann, Ivan Shcheklein, Alex Shinn, Ivan Shmakov, "Shmul", Tony Sidaway, Jeffrey B. Siegal, Andrey Sidorenko, Michele Simionato, Volker Stolz, Jon Strait, Dorai Sitaram, Robert Skeels, Jason Songhurst, Clifford Stein, Sunnan, Zbigniew Szadkowski, Rick Taube, Nathan Thern, Mike Thomas, Minh Thu, Christian Tismer, Andre van Tonder, John Tobey, Henrik Tramberend, Vladimir Tsichevsky, Neil van Dyke, Sam Varner, Taylor Venable, Sander Vesik, Jaques Vidrine, Panagiotis Vossos, Shawn Wagner, Peter Wang, Ed Watkeys, Brad Watson, Thomas Weidner, Goeran Weinholt, Matthew Welland, Drake Wilson, Joerg Wittenberger, Peter Wright, Mark Wutka, Richard Zidlicky and Houman Zolfaghari for bug-fixes, tips and suggestions.

CHICKEN uses the "irregex" regular expression package written by Alex Shinn.

Special thanks to Brandon van Every for contributing the (now defunct) CMake support and for helping with Windows build issues.

Also special thanks to Benedikt Rosenau for his constant encouragement.

Felix especially wants to thank Dunja Winkelmann for putting up with all of this and for her awesome support.

CHICKEN contains code from several people:

Richard Kelsey, Jonathan Rees and Taylor Campbell
: `syntax-rules` expander

Mikael Djurfeldt
: topological sort used by compiler.

Marc Feeley
: pretty-printer.

Aubrey Jaffer
: implementation of `dynamic-wind`.

Richard O'Keefe
: sorting routines.

Olin Shivers
implementation of `let-optionals[*]` and reference implementations of SRFI-1, SRFI-13 and SRFI-14.

Andrew Wilcox
queues.

The documentation and examples for explicit renaming macros was taken from the following paper:

William D. Clinger. *Hygienic macros through explicit renaming*. Lisp Pointers. IV(4). December 1991.

Home Download Manual Eggs API Browser Tests Bugs

show edit history

Free Text Identifier search
Search Help

Bibliography

Henry Baker: *CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A.*
http://home.pipeline.com/~hbaker1/CheneyMTA.html

Revised^5 Report on the Algorithmic Language Scheme
http://www.schemers.org/Documents/Standards/R5RS

Previous: Acknowledgements

manual