Chibi-Scheme

Minimal Scheme Implementation for use as an Extension Language

http://synthcode.com/wiki/chibi-scheme

Alex Shinn

Fri Jul 31 23:14:28 2020

Introduction

Chibi-Scheme is a very small library with no external dependencies, intended for use as an extension and scripting language in C programs. In addition to support for lightweight VM-based threads, each VM itself runs in an isolated heap allowing multiple VMs to run simultaneously in different OS threads.

The default language is the R7RS (scheme base) library, with support for all libraries from the small language. Support for additional languages such as JavaScript, Go, Lua and Bash are planned for future releases. Scheme is chosen as a substrate because its first class continuations and guaranteed tail-call optimization makes implementing other languages easy.

The system is designed in optional layers, beginning with a VM based on a small set of opcodes, a set of primitives implemented in C, a default language, a module system implementation, and a set of standard modules. You can choose whichever layer suits your needs best and customize the rest. Adding your own primitives or wrappers around existing C libraries is easy with the C FFI.

Chibi is known to build and run on 32 and 64-bit Linux, FreeBSD, DragonFly, OS X, iOS, Windows (under Cygwin) and Plan9.

Installation

To build, just run "make". This will provide a shared library "libchibi-scheme", as well as a sample "chibi-scheme" command-line repl. The "chibi-scheme-static" make target builds an equivalent static executable. If your make doesn't support GNU make conditionals, then you'll need to edit the top of the Makefile to choose the appropriate settings. On Plan9 just run "mk". You can test the build with "make test".

To install run "make install". If you want to try the executable out without installing, you will probably need to set LD_LIBRARY_PATH, depending on your platform. If you have an old version installed, run "make uninstall" first, or manually delete the directory.

You can edit the file chibi/features.h for a number of settings, mostly disabling features to make the executable smaller. You can specify standard options directly as arguments to make, for example

```
make CFLAGS=-Os CPPFLAGS=-DSEXP_USE_NO_FEATURES=1
```

to optimize for size, or

```
make LDFLAGS=-L/usr/local/lib CPPFLAGS=-I/usr/local/include
```

to compile against a library installed in /usr/local.

By default Chibi uses a custom, precise, non-moving GC (non-moving is important so you can maintain references from C code). You can link against the Boehm conservative GC by editing the features.h file, or directly from make with:

```
make SEXP_USE_BOEHM=1
```

To compile a static executable, use

```
make chibi-scheme-static SEXP_USE_DL=0
```

To compile a static executable with all C libraries statically included, first you need to create a clibs.c file, which can be done with:

```
make clibs.c
```

or edited manually. Be sure to run this with a non-static chibi-scheme. Then you can make the static executable with:

```
make -B chibi-scheme-static SEXP_USE_DL=0 CPPFLAGS=-DSEXP_USE_STATIC_LIBS
```

By default files are installed in /usr/local. You can optionally specify a PREFIX for the installation directory:

```
make PREFIX=/path/to/install/
sudo make PREFIX=/path/to/install/ install
```

Compile-Time Options

The include file `"chibi/features.h"` describes a number of C preprocessor values which can be enabled or disabled by setting to 1 or 0 respectively. For example, the above commands used the features `SEXP_USE_BOEHM`, `SEXP_USE_DL` and `SEXP_USE_STATIC_LIBS`. Many features are still experimental and may be removed from future releases, but the important features are listed below.

- `SEXP_USE_BOEHM` - link with the Boehm GC instead of the native Chibi GC
- `SEXP_USE_DL` - allow dynamic linking (enabled by default)
- `SEXP_USE_STATIC_LIBS` - compile the standard C libs statically
- `SEXP_USE_MODULES` - use the module system
- `SEXP_USE_GREEN_THREADS` - use lightweight threads (enabled by default)
- `SEXP_USE_SIMPLIFY` - use a simplification optimizer pass (enabled by default)
- `SEXP_USE_BIGNUMS` - use bignums (enabled by default)
- `SEXP_USE_FLONUMS` - use flonums (enabled by default)
- `SEXP_USE_RATIOS` - use exact ratios (enabled by default)
- `SEXP_USE_COMPLEX` - use complex numbers (enabled by default)
- `SEXP_USE_UTF8_STRINGS` - Unicode support (enabled by default)
- `SEXP_USE_STRING_INDEX_TABLE` - precompute offsets for O(1) `string-ref`
- `SEXP_USE_NO_FEATURES` - disable almost all features

Installed Programs

The command-line programs `chibi-scheme`, `chibi-doc` and `chibi-ffi` are installed by default, along with manpages. `chibi-scheme` provides a REPL and way to run scripts. Run -? for a brief list of options, or see the man page for more details. `chibi-doc` is the command-line interface to the literate documentation system described in (chibi scribble), and used to build this manual. `chibi-ffi` is a tool to build wrappers for C libraries, described in the FFI section below.

Default Language

Scheme Standard

The default language is the `(scheme base)` library from R7RS, which is mostly a superset of R5RS.

The reader defaults to case-sensitive, like R6RS and R7RS but unlike R5RS. You can specify the -f option on the command-line to enable case-folding. The default configuration includes the full numeric tower: fixnums, flonums, bignums, exact rationals and complex numbers, though this can be customized at compile time.

Full continuations are supported, but currently continuations don't take C code into account. This means that you can call from Scheme to C and then from C to Scheme again, but continuations passing through this chain may not do what you expect. The only higher-order C functions (thus potentially running afoul of this) in the standard environment are `load` and `eval`. The result of invoking a continuation created by a different thread is also currently unspecified.

In R7RS (and R6RS) semantics it is impossible to use two macros from different modules which both use the same auxiliary keywords (like `else` in `cond` forms) without renaming one of the keywords. By default Chibi considers all top-level bindings effectively unbound when matching auxiliary keywords, so this case will "just work". This decision was made because the chance of different modules using

the same keywords seems more likely than user code unintentionally matching a top-level keyword with a different binding, however if you want to use R7RS semantics you can compile with `SEXP_USE_STRICT_TOPLEVEL_BINDINGS=1`.

`load` is extended to accept an optional environment argument, like `eval`. You can also `load` shared libraries in addition to Scheme source files - in this case the function `sexp_init_library` is automatically called with the following signature:

```
sexp_init_library(sexp context, sexp self, sexp_sint_t n, sexp
environment, const char* version, sexp_abi_identifier_t abi);
```

Note, as R7RS (and earlier reports) states, "in contrast to other dialects of Lisp, the order of evaluation is unspecified [...]". Chibi is one of the few implementations which use a right-to-left evaluation order, which can be surprising to programmers coming from other languages.

Module System

Chibi supports the R7RS module system natively, which is a simple static module system. The Chibi implementation is actually a hierarchy of languages in the style of the Scheme48 module system, allowing easy extension of the module system itself. As with most features this is optional, and can be ignored or completely disabled at compile time.

Modules names are hierarchical lists of symbols or numbers. A module definition uses the following form:

```
(define-library (foo bar baz)
  <library-declarations> ...)
```

where `<library-declarations>` can be any of

```
(export <id> ...)                    ;; specify an export list
(import <import-spec> ...)           ;; specify one or more imports
(begin <expr> ...)                   ;; inline Scheme code
(include <file> ...)                 ;; load one or more files
(include-ci <file> ...)              ;; as include, with case-folding
(include-shared <file> ...)          ;; dynamic load a library (non-R7RS)
(alias-for <library>)                ;; a library alias (non-R7RS)
```

`<import-spec>` can either be a module name or any of

```
(only <import-spec> <id> ...)
(except <import-spec> <id> ...)
(rename <import-spec> (<from-id> <to-id>) ...)
(prefix <import-spec> <prefix-id>)
(drop-prefix <import-spec> <prefix-id>)   ;; non-R7RS
```

These forms perform basic selection and renaming of individual identifiers from the given module. They may be composed to perform combined selection and renaming.

Some modules can be statically included in the initial configuration, and even more may be included in image files, however in general modules are searched for in a module load path. The definition of the module `(foo bar baz)` is searched for in the file `"foo/bar/baz.sld"`. The default module path includes the installed directories, `"."` and `"./lib"`. Additional directories can be specified with the command-line options `-I` and `-A` (see the command-line options below) or with the `add-modue-directory` procedure at runtime. You can search for a module file with `(find-module-file <file>)`, or load it with `(load-module-file <file> <env>)`.

Within the module definition, files are loaded relative to the .sld file, and are written with their extension (so you can use whatever suffix you prefer - .scm, .ss, .sls, etc.).

Shared modules, on the other hand, should be specified *without* the extension - the correct suffix will be added portably (e.g. .so for Unix and .dylib for OS X).

You may also use `cond-expand` and arbitrary macro expansions in a module definition to generate `<module-declarations>`.

Macro System

`syntax-rules` macros are provided by default, with the extensions from SRFI-46. In addition, low-level hygienic macros are provided with a syntactic-closures interface, including `sc-macro-transformer`, `rsc-macro-transformer`, and `er-macro-transformer`. A good introduction to syntactic-closures can be found at http://community.schemewiki.org/?syntactic-closures.

`identifier?`, `identifier->symbol`, `identifier=?`, and `make-syntactic-closure` and `strip-syntactic-closures` are also available.

Types

You can define new record types with SRFI-9, or inherited record types with SRFI-99. These are just syntactic sugar for the following more primitive type constructors:

```
(register-simple-type <name-string> <parent> <field-names>)
 => <type>    ; parent may be #f, field-names should be a list of symbols

(make-type-predicate <opcode-name-string> <type>)
 => <opcode>  ; takes 1 arg, returns #t iff that arg is of the type

(make-constructor <constructor-name-string> <type>)
 => <opcode>  ; takes 0 args, returns a newly allocated instance of type

(make-getter <getter-name-string> <type> <field-index>)
 => <opcode>  ; takes 1 args, retrieves the field located at the index

(make-setter <setter-name-string> <type> <field-index>)
 => <opcode>  ; takes 2 args, sets the field located at the index

(type-slot-offset <type> <field-name>)
 => <index>   ; returns the index of the field with the given name
```

Unicode

Chibi supports Unicode strings and I/O natively. Case mappings and comparisons, character properties, formatting and regular expressions are all Unicode aware, supporting the latest version 13.0 of the Unicode standard.

Internally strings are encoded as UTF-8. This provides easy interoperability with many C libraries, but means that `string-ref` and `string-set!` are O(n), so they should be avoided in performance-sensitive code (unless you compile Chibi with SEXP_USE_STRING_INDEX_TABLE).

In general you should use high-level APIs such as `string-map` to ensure fast string iteration. String ports also provide a simple and portable way to efficiently iterate and construct strings, by looping over an input string or accumulating characters in an output string.

The `in-string` and `in-string-reverse` iterators in the `(chibi loop)` module will also iterate over strings efficiently while hiding the low-level details.

In the event that you do need a low-level interface, such as when writing your own iterator protocol, you should use string cursors. `(srfi 130)` provides a portable API for this, or you can use `(chibi string)` which builds on the following core procedures:

- `(string-cursor-start str)`

  returns a start cursor for the string

- `(string-cursor-end str)`

  returns a cursor one past the last valid cursor

- `(string-cursor-ref str cursor)`

  get the char at the given cursor

- `(string-cursor-next str cursor)`

  increment to the next cursor

- `(string-cursor-prev str cursor)`

decrement to the previous cursor

- `(substring-cursor str cs1 [cs2])`

  take a substring from the given cursors

- `(string-cursor<? cs1 cs2)`

  cs1 is before cs2

- `(string-cursor<=? cs1 cs2)`

  cs1 is before or the same as cs2

- `(string-cursor=? cs1 cs2)`

  cs1 is the same as cs2

- `(string-cursor>? cs1 cs2)`

  cs1 is after cs2

- `(string-cursor>=? cs1 cs2)`

  cs1 is the same or after cs2

Embedding in C

Quick Start

To use Chibi-Scheme in a program you need to link against the "libchibi-scheme" library and include the "eval.h" header file:

```
#include <chibi/eval.h>
```

All definitions begin with a "sexp_" prefix, or "SEXP_" for constants (deliberately chosen not to conflict with other Scheme implementations which typically use "scm_"). In addition to the prototypes and utility macros, this includes the following type definitions:

- `sexp` - an s-expression, used to represent all Scheme objects
- `sexp_uint_t` - an unsigned integer using as many bits as sexp
- `sexp_sint_t` - a signed integer using as many bits as sexp

A simple program might look like:

```
void dostuff(sexp ctx) {
  /* declare and preserve local variables *
  sexp_gc_var2(obj1, obj2);
  sexp_gc_preserve2(ctx, obj1, obj2);

  /* load a file containing Scheme code *
  obj1 = sexp_c_string(ctx, "/path/to/source/file.scm", -1);
  sexp_load(ctx, obj1, NULL);

  /* eval a C string as Scheme code *
  sexp_eval_string(ctx, "(some scheme expression)", -1, NULL);

  /* construct a Scheme expression to eval *
  obj1 = sexp_intern(ctx, "my-procedure", -1);
  obj2 = sexp_cons(ctx, obj1, SEXP_NULL);
  sexp_eval(ctx, obj2, NULL);

  /* release the local variables *
  sexp_gc_release2(ctx);
}

int main(int argc, char** argv) {
  sexp ctx;
  sexp_scheme_init();
  ctx = sexp_make_eval_context(NULL, NULL, NULL, 0, 0);
  sexp_load_standard_env(ctx, NULL, SEXP_SEVEN);
  sexp_load_standard_ports(ctx, NULL, stdin, stdout, stderr, 1);
  dostuff(ctx);
  sexp_destroy_context(ctx);
}
```

Looking at `main`, `sexp_make_eval_context` and `sexp_destroy_context` create and destroy a "context", which manages the heap and VM state. The meaning of the arguments is explained in detail below, but these values will give reasonable defaults, in this case constructing an environment with the core syntactic forms, opcodes, and standard C primitives.

This is still a fairly bare environment, so we call `sexp_load_standard_env` to find and load the default initialization file.

The resulting context can then be used to construct objects, call functions, and most importantly evaluate code, as is done in `dostuff`. The default garbage collector for Chibi is precise, which means we need to declare and preserve references to any temporary values we may generate, which is what the `sexp_gc_var2`, `sexp_gc_preserve2` and `sexp_gc_release2` macros do (there are similar macros for values 1-6). Precise GCs prevent a class of memory leaks (and potential attackes based thereon), but if you prefer convenience then Chibi can be compiled with a conservative GC and you can ignore these.

The interesting part is then the calls to `sexp_load`, `eval_string` and `eval` which evaluate code stored in files, C strings, or represented as s-expressions respectively.

Destroying a context runs any finalizers for all objects in the heap and then frees the heap memory (but has no effect on other contexts you or other users of the library may have created).

Contexts and Evaluation

Contexts represent the state needed to perform evaluation. This includes keeping track of the heap (when using precise GC), a default environment, execution stack, and any global values. A program being evaluated in one context may spawn multiple child contexts, such as when you call `eval`, and each child will share the same heap and globals. When using multiple interpreter threads, each thread has its own context.

You can also create independent contexts with their own separate heaps. These can run simultaneously in multiple OS threads without any need for synchronization.

- `sexp_make_context(sexp ctx, size_t size, size_t max_size)`

  Creates a new context object. The context has no associated environment, and so cannot be used for evaluation, but can be used to construct Scheme objects and call primitive C functions on them. If `ctx` is non-NULL it becomes the "parent" context. The resulting context will share the same heap as its parent, and when using a precise GC preserve any variables preserved by the parent, but the parent will not preserve the child context by default. Typically you either preserve the child manually or use it to perform a single sub-task then discard it and return to using only the parent. Otherwise, a new heap is allocated with `size` bytes, expandable to a maximum of `max_size` bytes, using the system defaults if either is 0. Note this context is not a malloced pointer (it resides inside a malloced heap), and therefore can't be passed to `free()`, or stored in a C++ smart pointer. It can only be reclaimed with `sexp_destroy_context`.

- `sexp_make_eval_context(sexp ctx, sexp stack, sexp env, sexp_uint_t size, sexp_uint_t max_size)`

  Similar to sexp_make_context, but also associates a stack, environment, and additional globals necessary to evaluate code. Either or both of `stack` and `env` may be NULL, in which case defaults will be generated. The default environment includes the compiled-in C primitives, as well as the 10 core forms: `define`, `set!`, `lambda`, `if`, `begin`, `quote`, `syntax-quote`, `define-syntax`, `let-syntax`, and `letrec-syntax`.

- `sexp_load_standard_env(sexp ctx, sexp env, sexp version)`

  Loads the standard parameters for `env`, constructs the feature list from pre-compiled defaults, and loads the installed initialization file for `version`, which should be the value `SEXP_SEVEN`. Also creates an `interaction-environment` parameter and sets `env` itself to that.

- `sexp_load_standard_ports(sexp ctx, sexp env, FILE* in, FILE* out, FILE* err, int leave_open)`

Creates `current-input-port`, `current-output-port`, and `current-error-port` parameters from `in`, `out` and `err`, and binds them in `env`. If `env` is NULL the default context environment is used. Any of the `FILE*` may be NULL, in which case the corresponding port is not set. If `leave_open` is true, then the underlying `FILE*` is left open after the Scheme port is closed, otherwise they are both closed together. If you want to reuse these streams from other vms, or from C, you should specify leave_open.

- `sexp_load(sexp ctx, sexp file, sexp env)`

  Searches the installation path for the `file` and loads it in the environment `env`. `file` may be a dynamic library or source code.

- `sexp_eval(sexp ctx, sexp obj, sexp env)`

  Evaluates `obj` as a source form in the environment `env` and returns the result.

- `sexp_eval_string(sexp ctx, const char* str, int len, sexp env)`

  Reads a s-expression from the C string `str` (or the first `len` bytes if `len` is non-negative), evaluates the resulting form in the environment `env`, and returns the result.

- `sexp_apply(sexp ctx, sexp proc, sexp args)`

  Applies the procedure `proc` to the arguments in the list `args` and returns the result.

- `sexp_context_env(sexp ctx)`

  Returns the current default environment associated with the context `ctx`.

- `sexp_env_define(sexp ctx, sexp env, sexp sym, sexp val)`

  Adds a new binding for `sym` in `env` with value `val`.

- `sexp_env_ref(sexp env, sexp sym, sexp dflt)`

  Returns the current binding of `sym` in `env`, or `dflt` if there is no binding.

- `sexp_env_import(sexp ctx, sexp to, sexp from, sexp ls, sexp immutp)`

  Imports the bindings from environment `from` into environment `to`. `ls` is the list of bindings to import - if it is `#f` then import all bindings. If `immutp` is true the imported bindings are immutable and cannot be redefined.

- `sexp_parameter_ref(sexp ctx, sexp param)`

  Returns the current dynamic value of the parameter `param` in the given context.

Garbage Collection

Chibi uses a precise garbage collector by default, which means when performing multiple computations on the C side you must explicitly preserve any temporary values. You can declare variables to be preserved with sexp_gc_var*n*, for n from 1 to 6.

`sexp_gc_var`*n*`(obj`$_1$`, obj`$_2$`, ..., obj`$_n$`)`

You can declare additional macros for larger values of n if needed.

This is equivalent to the declaration

`sexp obj`$_1$`, obj`$_2$`, ..., obj`$_n$`;`

except it makes preservation possible. Because it is a declaration it must occur at the beginning of your function, and because it includes assignments (in the macro-expanded form) it should occur after all other declarations.

To preserve these variables for a given context, you can then use sexp_gc_preserve*n*:

`sexp_gc_preserve`*n*`(ctx, obj`$_1$`, obj`$_2$`, ..., obj`$_n$`)`

This can be delayed in your code until you know a potentially memory-allocating computation will be performed, but once you call sexp_gc_preserve*n* it *must* be paired with a matching sexp_gc_release*n*:

```
sexp_gc_releasen(ctx);
```

Note each of these have different signatures. sexp_gc_var*n* just lists the variables to be declared. sexp_gc_preserve*n* prefixes these with the context in which they are to be preserved, and sexp_gc_release*n* just needs the context.

A typical usage for these is:

```
sexp foo(sexp ctx, sexp bar, sexp baz) {
  /* variable declarations *
  int i, j;
  ...
  sexp_gc_var3(tmp1, tmp2, res);

  /* asserts or other shortcut returns *
  sexp_assert_type(ctx, sexp_barp, SEXP_BAR, bar);
  sexp_assert_type(ctx, sexp_bazp, SEXP_BAZ, baz);

  /* preserve the variables in ctx *
  sexp_gc_preserve3(ctx, tmp1, tmp2, res);

  /* perform your computations *
  tmp1 = ...
  tmp2 = ...
  res = ...

  /* release before returning *
  sexp_gc_release3(ctx);

  return res;
}
```

If compiled with the Boehm GC, sexp_gc_var*n* just translates to the plain declaration, while sexp_gc_preserve*n* and sexp_gc_release*n* become noops.

When interacting with a garbage collection system from another language, or communicating between different Chibi managed heaps, you may want to manually ensure objects are preserved irrespective of any references to it from other objects in the same heap. This can be done with the `sexp_preserve_object` and `sexp_release_object` utilities.

```
sexp_preserve_object(ctx, obj)
```

Increment the absolute reference count for `obj`. So long as the reference count is above 0, `obj` will not be reclaimed even if there are no references to it from other object in the Chibi managed heap.

```
sexp_release_object(ctx, obj)
```

Decrement the absolute reference count for `obj`.

C API Index

The above sections describe most everything you need for embedding in a typical application, notably creating environments and evaluating code from sexps, strings or files. The following sections expand on additional macros and utilities for inspecting, accessing and creating different Scheme types, and for performing port and string I/O. It is incomplete - see the macros and SEXP_API annotated functions in the include files (sexp.h, eval.h, bignum.h) for more bindings.

Being able to convert from C string to sexp, evaluate it, and convert the result back to a C string forms the basis of the C API. Because Chibi is aimed primarily at minimal size, there are relatively few other utilities or helpers. It is expected most high-level code will be written in Scheme, and most low-level code will be written in pure, Scheme-agnostic C and wrapped via the FFI.

Type Predicates

The sexp represents different Scheme types with the use of tag bits for so-called "immediate" values, and a type tag for heap-allocated values. The following predicates can be used to distinguish these types. Note the predicates in C all end in

"p". For efficiency they are implemented as macros, and so may evaluate their arguments multiple times.

Note also that the non-immediate type checks will segfault if passed a NULL value. At the Scheme level (and the return values of any exported primitives) NULLs are never exposed, however some unexposed values in C can in certain cases be NULL. If you're not sure you'll need to check manually before applying the predicate.

- `sexp_booleanp(obj)` - `obj` is `#t` or `#f`
- `sexp_fixnump(obj)` - `obj` is an immediate integer
- `sexp_flonump(obj)` - `obj` is an inexact real
- `sexp_bignump(obj)` - `obj` is a heap-allocated integer
- `sexp_integerp(obj)` - `obj` is an integer
- `sexp_numberp(obj)` - `obj` is any kind of number
- `sexp_charp(obj)` - `obj` is a character
- `sexp_stringp(obj)` - `obj` is a string
- `sexp_string_cursorp(obj)` - `obj` is a string cursor
- `sexp_bytesp(obj)` - `obj` is a bytevector
- `sexp_symbolp(obj)` - `obj` is a symbol
- `sexp_idp(obj)` - `obj` is a symbol or hygienic identifier
- `sexp_nullp(obj)` - `obj` is the null value
- `sexp_pairp(obj)` - `obj` is a pair
- `sexp_vectorp(obj)` - `obj` is a vector
- `sexp_iportp(obj)` - `obj` is an input port
- `sexp_oportp(obj)` - `obj` is an output port
- `sexp_portp(obj)` - `obj` is any kind of port
- `sexp_procedurep(obj)` - `obj` is a procedure
- `sexp_opcodep(obj)` - `obj` is a primitive opcode
- `sexp_applicablep(obj)` - `obj` is valid as the first arg to apply
- `sexp_typep(obj)` - `obj` is a type
- `sexp_exceptionp(obj)` - `obj` is an exception
- `sexp_contextp(obj)` - `obj` is a context
- `sexp_envp(obj)` - `obj` is an environment
- `sexp_corep(obj)` - `obj` is a special form
- `sexp_macrop(obj)` - `obj` is a macro
- `sexp_synclop(obj)` - `obj` is a syntactic closure
- `sexp_bytecodep(obj)` - `obj` is compiled bytecode
- `sexp_cpointerp(obj)` - `obj` is an opaque C pointer

Constants

The following shortcuts for various immediate values are available.

- `SEXP_FALSE` - the false boolean
- `SEXP_TRUE` - the true boolean
- `SEXP_NULL` - the empty list
- `SEXP_EOF` - the end-of-file object
- `SEXP_VOID` - an undefined value often returned by mutators
- `SEXP_ZERO` - shortcut for sexp_make_fixnum(0)
- `SEXP_ONE` - shortcut for sexp_make_fixnum(1)
- ...
- `SEXP_TEN` - shortcut for sexp_make_fixnum(10)
- `SEXP_NEG_ONE` - shortcut for sexp_make_fixnum(-1)

String Handling

Scheme strings are length bounded C strings which can be accessed with the following macros:

- `char* sexp_string_data(sexp s)` - the raw bytes of the string
- `sexp_uint_t sexp_string_size(sexp s)` - the number of raw bytes in the string
- `sexp_uint_t sexp_string_length(sexp s)` - the number of characters encoded in `s`

Currently all Scheme strings also happen to be NULL-terminated, but you should not rely on this and be sure to use the size as a bounds check. The runtime does not prevent embedded NULLs inside strings, however data after the NULL may be ignored.

By default (unless you compile with -DSEXP_USE_UTF8_STRING=0), strings are interpreted as UTF-8 encoded on the Scheme side, as describe in section Unicode above. In many cases you can ignore this on the C side and just treat the string as an opaque sequence of bytes. However, if you need to you can use the following macros to safely access the contents of the string regardless of the options Chibi was compiled with:

- `sexp sexp_string_ref(sexp ctx, sexp s, sexp i)` - returns the character at index i
- `sexp sexp_string_set(sexp ctx, sexp s, sexp i, sexp ch)` - sets the character at index i
- `sexp sexp_string_cursor_ref(sexp ctx, sexp s, sexp i)` - returns the character at raw offset i (a fixnum)
- `sexp sexp_string_cursor_set(sexp ctx, sexp s, sexp i, sexp ch)` - sets the character at raw offset i (a fixnum)
- `sexp sexp_string_cursor_next(sexp s, sexp i)` - returns the next cursor after raw offset `i`
- `sexp sexp_string_cursor_prev(sexp s, sexp i)` - returns the previous cursor before raw offset `i`
- `sexp sexp_substring(sexp ctx, sexp s, sexp i, sexp j)` - returns the substring between indices `i` and `j`
- `sexp sexp_substring_cursor(sexp ctx, sexp s, sexp i, sexp j)` - returns the substring between raw offsets `i` and `j`

When UTF-8 support is not compiled in the cursor and non-cursor variants are equivalent.

Accessors

The following macros provide access to the different components of the Scheme types. They do no type checking, essentially translating directly to pointer offsets, so you should be sure to use the above predicates to check types first. They only evaluate their arguments once.

- `sexp_make_boolean(n)` - `#f` if `n` is 0, `#t` otherwise
- `sexp_unbox_boolean(obj)` - 1 if `obj` is `#t`, 0 otherwise
- `sexp_make_fixnum(n)` - creates a new fixnum representing int `n`
- `sexp_unbox_fixnum(obj)` - converts a fixnum to a C integer
- `sexp_make_character(ch)` - creates a new character representing char `ch`
- `sexp_unbox_character(obj)` - converts a character to a C char
- `sexp sexp_make_string_cursor(int offset)` - creates a string cursor for the given byte offset
- `int sexp_unbox_string_cursor(sexp sc)` - returns the offset for the given string cursor
- `sexp_car(pair)` - the car of `pair`
- `sexp_cdr(pair)` - the cdr of `pair`
- `sexp_ratio_numerator(q)` - the numerator of the ratio `q`
- `sexp_ratio_denominator(q)` - the denominator of the ratio `q`
- `sexp_complex_real(z)` - the real part of the complex `z`
- `sexp_complex_imag(z)` - the imaginary part of the complex `z`
- `sexp_string_length(str)` - the byte length of `str` as an int
- `sexp_string_ref(str, i)` - the `i`'th byte of string `str`
- `sexp_string_set(str, i, ch)` - set the `i`'th byte of string `str`
- `sexp_bytes_length(bv)` - the length of `bv` as an int
- `sexp_bytes_data(bv)` - the raw char* data of `bv`
- `sexp_vector_length(vec)` - the length of `vec` as an int
- `sexp_vector_ref(vec, i)` - the `i`'th object of vector `vec`
- `sexp_vector_set(vec, i, obj)` - set the `i`'th object of vector `vec`
- `sexp_bytes_length(bv)` - the number of bytes in bytevector `bv`
- `sexp_bytes_ref(bv, i)` - the `i`'th byte of bytevector `bv`
- `sexp_bytes_set(bv, i, k)` - set the `i`'th byte of bytevector `bv`

Constructors

Constructors allocate memory and so must be passed a context argument. Any of these may fail and return the OOM exception object.

- `sexp_cons(sexp ctx, sexp obj1, sexp obj2)` - create a new pair whose car is `obj1` and whose cdr is `obj2`
- `sexp_list1(sexp ctx, sexp obj)` - alias for sexp_cons(ctx, obj, SEXP_NULL)

- `sexp_list2(sexp ctx, sexp obj1, sexp obj2)` - create a list of two elements
- `sexp_make_string(sexp ctx, sexp len, sexp ch)` - create a new Scheme string of `len` characters, all initialized to `ch`
- `sexp_c_string(sexp ctx, const char* str, int len)` - create a new Scheme string copying the first `len` characters of the C string `str`. If `len` is -1, uses strlen(`str`).
- `sexp_intern(sexp ctx, const char* str, int len)` - interns a symbol from the first `len` characters of the C string `str`. If `len` is -1, uses strlen(`str`).
- `sexp_make_bytes(sexp ctx, sexp len, sexp i)` - create a new Scheme bytevector of `len` bytes, all initialized to `i`
- `sexp_make_vector(sexp ctx, sexp len, sexp obj)` - create a new vector of `len` elements, all initialized to `obj`
- `sexp_make_integer(sexp ctx, sexp_sint_t n)` - create an integer, heap allocating as a bignum if needed
- `sexp_make_unsigned_integer(sexp ctx, sexp_uint_t n)` - create an unsigned integer, heap allocating as a bignum if needed

I/O

- `sexp_read(sexp ctx, sexp in)` - read a single datum from port `in`
- `sexp_write(sexp ctx, sexp obj, sexp out)` - write `obj` to port `out`
- `sexp_write_string(sexp ctx, char* str, sexp out)` - write the characters in `str` to port `out`
- `sexp_newline(sexp ctx, sexp out)` - write a newline to port `out`
- `sexp_print_exception(sexp ctx, sexp exn, sexp out)` - print an error message for `exn` to port `out`
- `sexp_current_input_port(sexp ctx)` - the `current-input-port`
- `sexp_current_output_port(sexp ctx)` - the `current-output-port`
- `sexp_current_error_port(sexp ctx)` - the `current-error-port`
- `sexp_debug(sexp ctx, char* msg, sexp obj)` - write `obj` with a debug message prefix to `current-error-port`
- `sexp_read_from_string(sexp ctx, char* str, int len)` - read a single datum from `str`, using at most `len` bytes if `len` is non-negative
- `sexp_write_to_string(sexp ctx, sexp obj)` - return a Scheme string representation of `obj`
- `sexp_open_input_string(sexp ctx, sexp str)` - equivalent to `open-input-string`
- `sexp_open_output_string(sexp ctx)` - equivalent to `open-output-string`
- `sexp_get_output_string(sexp ctx, sexp port)` - equivalent to `get-output-string`

Utilities

- `sexp_equalp(sexp ctx, sexp x, sexp y)` - `equal?`
- `sexp_length(sexp ctx, sexp ls)` - `length`
- `sexp_listp(sexp ctx, sexp x)` - `list?`
- `sexp_memq(sexp ctx, sexp x, sexp ls)` - `memq`
- `sexp_assq(sexp ctx, sexp x, sexp ls)` - `assq`
- `sexp_reverse(sexp ctx, sexp ls)` - `reverse`
- `sexp_nreverse(sexp ctx, sexp ls)` - `reverse!`
- `sexp_append2(sexp ctx, sexp ls)` - `append` for two arguments
- `sexp_copy_list(sexp ctx, sexp ls)` - return a shallow copy of `ls`
- `sexp_list_to_vector(sexp ctx, sexp ls)` - `list->vector`
- `sexp_symbol_to_string(sexp ctx, sexp sym)` - `symbol->string`
- `sexp_string_to_symbol(sexp ctx, sexp str)` - `string->symbol`
- `sexp_string_to_number(sexp ctx, sexp str)` - `string->number`

Exceptions

Exceptions can be created with the following:

- `sexp sexp_make_exception (sexp ctx, sexp kind, sexp message, sexp irritants, sexp procedure, sexp source)`

  Create an exception of the given `kind` (a symbol), with the string `message`, and `irritants` list. `procedure` and `source` provide information about the error location. From a C function, `procedure` should generally be `self`.

- `sexp sexp_user_exception (sexp ctx, sexp self, const char *msg, sexp x)`

  Shortcut for an exception of kind `user`, with the given message and single irritant.

- `sexp sexp_type_exception (sexp ctx, sexp self, sexp_uint_t type_id, sexp x)`

  Shortcut for an exception of kind `type`, where `x` was expected to be of type `type_id` but wasn't.

- `sexp sexp_xtype_exception (sexp ctx, sexp self, const char *msg, sexp x)`

  Shortcut for an exception of kind `type`, for more general domain errors, where `x` failed to meet the restrictions in `msg`.

Returning an exception from a C function by default *raises* that exception in the VM. If you want to pass an exception as a first class value, you have to wrap it first:

```
sexp sexp_maybe_wrap_error (sexp ctx, sexp obj)
```

Customizing

You can add your own types and primitives with the following functions.

- `sexp sexp_define_foreign(sexp ctx, sexp env, const char* name, int num_args, sexp_proc1 func)`

  Defines a new primitive procedure with the name `name` in the environment `env`. The procedure takes `num_args` arguments and passes them to the C function `func`. The C function must take the standard calling convention: `sexp func(sexp ctx, sexp self, sexp n, sexp` $\texttt{arg}_1$`, ..., sexp` $\texttt{arg}_{\texttt{num\_args}}$`)` where `ctx` is the current context, `self` is the procedure itself, and `n` is the number of arguments passed. `func` is responsible for checking its own argument types.

- `sexp sexp_define_foreign_opt(sexp ctx, sexp env, const char* name, int num_args, sexp_proc1 func, sexp dflt)`

  Equivalent to `sexp_define_foreign`, except the final argument is optional and defaults to the value `dflt`.

- `sexp sexp_define_foreign_param(sexp ctx, sexp env, const char* name, int num_args, sexp_proc1 func, const char* param)`

  Equivalent to `sexp_define_foreign_opt`, except instead of a fixed default argument `param` should be the name of a parameter bound in `env`.

- `sexp sexp_register_simple_type(sexp ctx, sexp name, sexp parent, sexp slots)`

  Defines a new simple record type having `slots` new slots in addition to any inherited from the parent type `parent`. If `parent` is false, inherits from the default `object` record type.

- `sexp sexp_register_c_type(sexp ctx, sexp name, sexp finalizer)`

  Shortcut to defines a new type as a wrapper around a C pointer. Returns the type object, which can be used with sexp_make_cpointer to wrap instances of the type. The finalizer may be sexp_finalize_c_type in which case managed pointers are freed as if allocated with malloc, NULL in which case the pointers are never freed, or otherwise a procedure of one argument which should release any resources.

- `sexp sexp_make_cpointer(sexp ctx, sexp_uint_t type_id, void* value, sexp parent, int freep)`

  Creates a new instance of the type indicated by type_id wrapping value. If parent is provided, references to the child will also preserve the parent, important e.g. to preserve an enclosing struct when wrapped references to nested structs are still in use. If freep is true, then when reclaimed by the GC

the finalizer for this type, if any, will be called on the instance. You can retrieve the id from a type object with sexp_type_tag(type).

See the C FFI for an easy way to automate adding bindings for C functions.

C FFI

The "chibi-ffi" script reads in the C function FFI definitions from an input file and outputs the appropriate C wrappers into a file with the same base name and the ".c" extension. You can then compile that C file into a shared library:

```
chibi-ffi file.stub
cc -fPIC -shared file.c -lchibi-scheme
```

(or using whatever flags are appropriate to generate shared libs on your platform) and the generated .so file can be loaded directly with `load`, or portably using `(include-shared "file")` in a module definition (note that include-shared uses no suffix).

The goal of this interface is to make access to C types and functions easy, without requiring the user to write any C code. That means the stubber needs to be intelligent about various C calling conventions and idioms, such as return values passed in actual parameters. Writing C by hand is still possible, and several of the core modules provide C interfaces directly without using the stubber.

Includes and Initializations

- `(c-include header)` - includes the file `header`
- `(c-system-include header)` - includes the system file `header`
- `(c-declare args ...)` - outputs `args` directly in the top-level C source
- `(c-init args ...)` - evaluates `args` as C code after all other library initializations have been performed, with ctx and env in scope

Struct Interface

C structs can be bound as Scheme types with the `define-c-struct` form:

```
(define-c-struct struct_name
  [predicate: predicate-name]
  [constructor: constructor-name]
  [finalizer: c_finalizer_name]
  (type c_field_name getter-name setter-name) ...)
```

`struct_name` should be the name of a C struct type. If provided, `predicate-name` is bound to a procedure which takes one object and returns `#t` iff the object is of type `struct_name`.

If provided, `constructor-name` is bound to a procedure of zero arguments which creates and returns a newly allocated instance of the type.

If a finalizer is provided, `c_finalizer_name` must be a C function which takes one argument, a pointer to the struct, and performs any cleanup or freeing of resources necessary.

The remaining slots are similar to the SRFI-9 syntax, except they are prefixed with a C type (described below). The `c_field_name` should be a field name of `struct_name`. `getter-name` will then be bound to a procedure of one argument, a struct_name type, which returns the given field. If provided, `setter-name` will be bound to a procedure of two arguments to mutate the given field.

The variants `define-c-class` and `define-c-union` take the same syntax but define types with the `class` and `union` keywords respectively. `define-c-type` just defines accessors to an opaque type without any specific struct-like keyword.

```
;; Example: the struct addrinfo returned by getaddrinfo.

(c-system-include "netdb.h")

(define-c-struct addrinfo
  finalizer: freeaddrinfo
  predicate: address-info?
  (int              ai_family    address-info-family)
  (int              ai_socktype  address-info-socket-type)
```

```
(int             ai_protocol  address-info-protocol)
((link sockaddr) ai_addr      address-info-address)
(size_t          ai_addrlen   address-info-address-length)
((link addrinfo) ai_next      address-info-next))
```

Function and Constant Interface

C functions are defined with:

```
(define-c return-type name-spec (arg-type ...))
```

where `name-space` is either a symbol name, or a list of `(scheme-name c_name)`. If just a symbol is used, the C name is generated automatically by replacing any dashes (-) in the Scheme name with underscores (_).

Each `arg-type` is a type suitable for input validation and conversion as discussed below.

```
;; Example: define connect(2) in Scheme
(define-c int connect (int sockaddr int))
```

Constants can be defined with:

```
(define-c-const type name-space)
```

where `name-space` is the same form as in `define-c`. This defines a Scheme variable with the same value as the C constant.

```
;; Example: define address family constants in Scheme
(define-c-const int (address-family/unix "AF_UNIX"))
(define-c-const int (address-family/inet "AF_INET"))
```

C Types

Basic Types

- `void`
- `boolean`
- `char`
- `sexp` (no conversions)

Integer Types

- `signed-char`
- `short`
- `int`
- `long`
- `unsigned-char`
- `unsigned-short`
- `unsigned-int`
- `unsigned-long`
- `size_t`
- `pid_t`
- `uid_t`
- `gid_t`
- `time_t` (in seconds, but using the chibi epoch of 2010/01/01)
- `errno` (as a return type returns `#f` on error)

Float Types

- `float`
- `double`
- `long-double`

String Types

- `string` - a null-terminated char*
- `env-string` - a `VAR=VALUE` string represented as a `(VAR . VALUE)` pair in Scheme
- `(array char)` is equivalent to `string`

Port Types

- `input-port`
- `output-port`

Struct Types

Struct types are by default just referred to by the bare `struct_name` from `define-c-struct`, and it is assumed you want a pointer to that type. To refer to the full struct, use the struct modifier, as in `(struct struct-name)`.

Type modifiers

Any type may also be written as a list of modifiers followed by the type itself. The supported modifiers are:

- `const`

  Prepends the "const" C type modifier. As a return or result parameter, makes non-immediates immutable.

- `free`

  It's Scheme's responsibility to "free" this resource. As a return or result parameter, registers the freep flag this causes the type finalizer to be run when GCed.

- `maybe-null`

  This pointer type may be NULL. As a result parameter, NULL is translated to #f normally this would just return a wrapped NULL pointer. As an input parameter, #f is translated to NULL normally this would be a type error.

- `pointer`

  Create a pointer to this type. As a return parameter, wraps the result in a vanilla cpointer. As a result parameter, boxes then unboxes the value.

- `reference`

  A stack-allocated pointer to this type. As a result parameter, passes a stack-allocated pointer to the value, then returns the dereferenced pointer.

- `struct`

  Treat this struct type as a struct, not a pointer. As an input parameter, dereferences the pointer. As a type field, indicates a nested struct.

- `link`

  Add a gc link. As a field getter, link to the parent object, so the parent won't be GCed so long as we have a reference to the child. This behavior is automatic for nested structs.

- `result`

  Return a result in this parameter. If there are multiple results (including the return type), they are all returned in a list. If there are any result parameters, a return type of errno returns #f on failure, and as eliminated from the list of results otherwise.

- `(value <expr>)`

  Specify a fixed value. As an input parameter, this parameter is not provided in the Scheme API but always passed as <expr>.

- `(default <expr>)`

  Specify a default value. As the final input parameter, makes the Scheme parameter optional, defaulting to <expr>.

- `(array <type> [<length>])`

An array type. Length must be specified for return and result parameters. If specified, length can be either an integer, indicating a fixed size, or the symbol null, indicating a NULL-terminated array.

Standard Modules

A number of SRFIs are provided in the default installation. Note that SRFIs 0, 6, 23, 46 and 62 are built into the default environment so there's no need to import them. SRFI 22 is available with the "-r" command-line option. This list includes popular SRFIs or SRFIs used in standard Chibi modules (many other SRFIs are available on snow-fort):

- (srfi 0) - cond-expand
- (srfi 1) - list library
- (srfi 2) - and-let*
- (srfi 6) - basic string ports
- (srfi 8) - receive
- (srfi 9) - define-record-type
- (srfi 11) - let-values/let*-values
- (srfi 14) - character-set library
- (srfi 16) - case-lambda
- (srfi 18) - multi-threading support
- (srfi 22) - running scheme scripts on Unix
- (srfi 23) - error reporting mechanism
- (srfi 26) - cut/cute partial application
- (srfi 27) - sources of random bits
- (srfi 33) - bitwise operators
- (srfi 38) - read/write shared structures
- (srfi 39) - parameter objects
- (srfi 41) - streams
- (srfi 46) - basic syntax-rules extensions
- (srfi 55) - require-extension
- (srfi 62) - s-expression comments
- (srfi 69) - basic hash tables
- (srfi 95) - sorting and merging
- (srfi 98) - environment access
- (srfi 99) - ERR5RS records
- (srfi 101) - purely functional random-access pairs and lists
- (srfi 111) - boxes
- (srfi 113) - sets and bags
- (srfi 115) - Scheme regular expressions
- (srfi 116) - immutable list library
- (srfi 117) - mutable queues
- (srfi 121) - generators
- (srfi 124) - ephemerons
- (srfi 125) - intermediate hash tables
- (srfi 127) - lazy sequences
- (srfi 128) - comparators (reduced)
- (srfi 129) - titlecase procedures
- (srfi 130) - cursor-based string library
- (srfi 132) - sort libraries
- (srfi 133) - vector library
- (srfi 134) - immutable deques
- (srfi 135) - immutable texts
- (srfi 139) - syntax parameters
- (srfi 141) - integer division
- (srfi 142) - bitwise operations
- (srfi 143) - fixnums
- (srfi 144) - flonums
- (srfi 145) - assumptions
- (srfi 147) - custom macro transformers
- (srfi 151) - bitwise operators
- (srfi 154) - first-class dynamic extents
- (srfi 158) - generators and accumulators
- (srfi 160) - homogeneous numeric vector libraries
- (srfi 165) - the environment Monad
- (srfi 166) - monadic formatting
- (srfi 188) - splicing binding constructs for syntactic keywords

Additional non-standard modules are put in the `(chibi)` module namespace.

- (chibi app) - Unified option parsing and config
- (chibi ast) - Abstract Syntax Tree and other internal data types
- (chibi base64) - Base64 encoding and decoding
- (chibi bytevector) - Bytevector Utilities
- (chibi config) - General configuration management
- (chibi crypto md5) - MD5 hash
- (chibi crypto rsa) - RSA public key encryption
- (chibi crypto sha2) - SHA-2 hash
- (chibi diff) - LCS Algorithm and diff utilities
- (chibi disasm) - Disassembler for the virtual machine
- (chibi doc) - Chibi documentation utilities
- (chibi edit-distance) - A levenshtein distance implementation
- (chibi equiv) - A version of `equal?` which is guaranteed to terminate
- (chibi filesystem) - Interface to the filesystem and file descriptor objects
- (chibi generic) - Generic methods for CLOS-style object oriented programming
- (chibi heap-stats) - Utilities for gathering statistics on the heap
- (chibi io) - Various I/O extensions and custom ports
- (chibi iset base) - Compact integer sets
- (chibi iset base) - Compact integer sets
- (chibi iset constructors) - Compact integer set construction
- (chibi iset iterators) - Iterating over compact integer sets
- (chibi json) - JSON reading and writing
- (chibi loop) - Fast and extensible loop syntax
- (chibi match) - Intuitive and widely supported pattern matching syntax
- (chibi math prime) - Prime number utilities
- (chibi memoize) - Procedure memoization
- (chibi mime) - Parse MIME files into SXML
- (chibi modules) - Introspection for the module system itself
- (chibi net) - Simple networking interface
- (chibi net http-server) - Simple http-server with servlet support
- (chibi net servlet) - HTTP servlets for http-server or CGI
- (chibi parse) - Parser combinators with convenient syntax
- (chibi pathname) - Utilities to decompose and manipulate pathnames
- (chibi process) - Interface to spawn processes and handle signals
- (chibi repl) - A full-featured Read/Eval/Print Loop
- (chibi scribble) - A parser for the scribble syntax used to write this manual
- (chibi string) - Cursor-based string library (predecessor to SRFI 130)
- (chibi stty) - A high-level interface to ioctl
- (chibi sxml) - SXML utilities
- (chibi system) - Access to the host system and current user information
- (chibi temp-file) - Temporary file and directory creation
- (chibi test) - A simple unit testing framework
- (chibi time) - An interface to the current system time
- (chibi trace) - A utility to trace procedure calls
- (chibi type-inference) - An easy-to-use type inference system
- (chibi uri) - Utilities to parse and construct URIs
- (chibi weak) - Data structures with weak references

Snow Package Manager

Beyond the distributed modules, Chibi comes with a package manager based on Snow2 which can be used to share R7RS libraries. Packages are distributed as tar gzipped files called "snowballs," and may contain multiple libraries. The program is installed as `snow-chibi`. The "help" subcommand can be used to list all subcommands and options. Note by default `snow-chibi` uses an image file to speed-up loading (since it loads many libraries) - if you have any difficulties with image files on your platform you can run

```
snow-chibi --noimage
```

to disable this feature.

Querying Packages and Status

By default `snow-chibi` looks for packages in the public repository http://snow-fort.org/, though you can customize this with the `--repository-uri` option. Packages can be browsed on the site, but you can also search and query from the command-line tool.

- search terms ... - search for packages

  Print a list of available packages matching the given keywords.

- show names ... - show package descriptions

  Show detailed information for the listed packages, which can be sexp library names or the dotted shorthand used by chibi. For example, `snow-chibi show "(chibi match)"` can be shortened as `snow-chibi show chibi.match`.

- status names ... - print package status

  Print the installed version of the given packages. Uninstalled packages will not be shown. If no names are given, prints all currently installed packages.

- implementations - print list of available implementations

  Print the currently installed Scheme implementations supported by `snow-chibi`. If an implementation is found but has an older version, a warning is printed.

Managing Packages

The basic package management functionality, installing upgrading and removing packages.

- install names ... - install packages

  Install the given packages. Package names can be sexp lists or use the dotted shorthand. Explicit names for packages are optional, as a package can always be referred to by the name of any library it contains. If multiple packages provide libraries with the same name, you will be asked to confirm which implementation to install.

  You can also bypass the repository and install a manually downloaded snowball by giving a path to that file instead of a name.

- upgrade names ... - upgrade installed packages

  Upgrade the packages if new versions are available. If no names are given, upgrades all eligible packages.

- remove names ... - remove packages

  Uninstalls the given packages. If the packages were not installed with `snow-chibi` they cannot be removed.

- update - update local cache of remote repository

  `snow-chibi` keeps a local cache of the remote repository and updates only periodically for performance, but you can force an update with this command.

Authoring Packages

Creating packages can be done with the `package` command, though other commands allow for uploading to public repositories.

- package files ... - create a package

  Create a package snowball from the given files, which should be R7RS library files containing `define-library` forms. Include files are inferred and packaged automatically. You can share packages directly, or upload them to a snow repository for easy automated install.

- upload files ... - upload packages

  Sign and upload to the default snow host. The files may either be .tgz package files, or files containing define-library forms as in the `package` command, from which packages are generated automatically. Before you can upload to the default host a key must be generated and registered first with the `gen-key` and `reg-key` commands.

- gen-key - create a new key

  Create a new key, with your name, email address, and optionally an RSA public key pair (disabled by default in the current implementation). This is saved locally to ~/.snow/priv-key.scm - you need to register it with reg-key before it can be used for uploads.

- reg-key - register a key

  Register your key on the default snow host.

- sign file - sign a package

  Sign a file with your key and write it to the .sig file. This can be used with the verify command for testing, but otherwise is not needed as the upload command generates the signature automatically.

- verify sig-file - verify a signature

  Print a message verifying if a signature is valid.

Easy Packaging

To encourage sharing code it's important to make it as easy as possible to create packages, while encouraging documentation and tests. In particular, you should never need to duplicate information anywhere. Thus the `package` command automatically locates and packages include files (and data and ffi files) and determines dependencies for you. In addition, it can automatically handle versions, docs and tests:

- version - can come explicitly from the `--version` option, or the `--version-file=<file>` option
- docs - can come explicitly from the `--doc=<file>` option, or be extracted automatically from literate documentation with `doc-for-scribble`
- tests - can come explicitly from the `--test=<prog-file>` option, or the `--test-library=<lib-name>` which will generate a program to run just the `run-tests` thunk in that library

Other useful meta-info options include:

- `--authors` - specify the package authors (comma-delimited)
- `--maintainers` - specify the package maintainers (comma-delimited)
- `--license` - specify the package licence

These three are typically always the same, so it's useful to save them in your ~/.snow/config.scm file. This file contains a single sexp and can specify any option, for example:

```
((repository-uri "http://alopeke.gr/repo.scm")
 (command
  (package
   (authors "Socrates &lt;hemlock@aol.com>")
   (doc-from-scribble #t)
   (version-file "VERSION")
   (test-library (append-to-last -test))
   (license gpl))))
```

Top-level snow options are represented as a flat alist. Options specific to a command are nested under `(command (name ...))`, with most options here being for `package`. Here unless overridden on the command-line, all packages will use the given author and license, try to extract literate docs from the code, look for a version in the file "VERSION", and try to find a test with the same library name appended with `-test`, e.g. for the library `(socratic method)`, the test library would be `(socratic method-test)`. This form is an alternate to using an explicit test-library name, and encourages you to keep your tests close to the code they test. In the typical case, if using these conventions, you can thus simply run `snow-chibi package <lib-file>` without any other options.

Other Implementations

Although the command is called `snow-chibi`, it supports several other R7RS implementations. The `implementations` command tells you which you currently have installed. The following are currently supported:

- chibi - native support as of version 0.7.3
- chicken - version >= 4.9.0 with the `r7rs` egg
- foment - version >= 0.4
- gauche - version >= 0.9.4
- kawa - version >= 2.0; you need to add the install dir to the search path, e.g. `-Dkawa.import.path=/usr/local/share/kawa`
- larceny - version 0.98; you need to add "lib/Snow" to the paths in startup.sch