Hacking STklos

Jeronimo Pellegrini

Table of Contents

This is a quick guide to STklos hacking. It’s not detailed, so the document doesn’t become huge, and also because after understanding the basics, hacking STklos should not be difficult.

Chapter 1. Basic editor configuration

There is a `.editorconfig` file in STklos' root folder, which describes the style to be used, and which is automatically used when editorconfig is configured ([editorconfig](https://editorconfig.org/) helps maintain consistentcoding styles for multiple developers working on the same project across various editors and IDEs).

Chapter 2. Directories

The subdirectories in the STklos source tree are:

- `doc` – documentation, written mostly in asciidoctor
- `etc` – various sample files for specific needs
- `examples` – examples (oh, who could tell?)
- `ffi` – `libffi` (a local copy)
- `gc` – the Boehm-Demers-Weiser garbage collector, libgc (a local copy)
- `gmp` – a slow compatible GNU MP
- `lib` – Scheme files, including from basic things like the boot program up to high-level things like modules implementing libraries and SRFIs
- `pcre2` – `libpcre` (a local copy)
- `pkgman` – the package manager
- `src` – the STklos core, written in C
- `tests` – the tests, of course!
- `utils` – utilities and wrappers

The "local copies" of `libffi`, `libgc` and `libpcre`, as well as the mini-GMP in `gmp/` are compiled when there’s no version of those available in the system, or when you force their use in the configure script with `--with-provided-gc`, `--with-gmp-light` and so on.

Chapter 3. Basic debugging

3.1. STK_DEBUG

STklos has conditionally-compiled debugging code, which is enabled when the `STK_DEBUG` variable is visible to the C compiler. To enable a debug-enabled binary of STklos, configure it passing `CFLAGS="-DSTK_DEBUG"` to the configure script:

```
./configure CFLAGS="-DSTK_DEBUG"
```

This will enable:

- `[misc.c]`: `(%%debug)` which toggles debugging on and off.
- `[misc.c]`: `(%c-backtrace)`, which produces a backtrace of C function calls.
- `[misc.c]`: `(%test proc)`, which applies `proc` without arguments.
- `[misc.c]`: `(%vm …)`, which you can customize in `src/vm.c` to your needs.
- `[src/utf8.c]`: `(%char-utf8-encoding c)`, which shows how the character `c` is encoded in UTF8.
- ``[`src/utf8.c]``: `(%dump-string s)`, which shows the bytes in the internal representation of the string `s`.
- `[src/promise.c]`: `(%promise-value p)`, which returns the value of promise `p`. When not yet forced, the value will be a procedure, which you can then call. But calling `%promise-value p` does **not** force `p`, and does not interfere with the rest of the program.
- `[src/promise.c]`: `(%promise-value-set! p v)`, which sets the value of promise `p` to `v`.

Clearly, you can add other primitives useful for debugging guarded by

```
#ifdef STK_DEBUG
...
#endif
```

as necessary.

3.2. Other debugging primitives in Scheme

Even without `STK_DEBUG`, you can use in your Scheme code:

- `(%vm-backtrace)` to obtain a trace of Scheme procedure calls

3.3. C debugging

When copiling the C part of STklos, it may be interesting to compile with `-g -O0 -Wall` also:

```
./configure CFLAGS="-DSTK_DEBUG -g -O0 -Wall"
```

And to use GCC’s static analyzer (with GCC version 11 or later),

```
./configure CFLAGS="-DSTK_DEBUG -g -O0 -Wall -fanalyzer"
```

To debug STklos, you can use gdb:

```
gdb -q src/stklos
```

Chapter 4. STklos initialization

`main` is in `src/stklos.c`, where command line options are parsed and the scheme interpreter is started:

- `STk_init_library` – performs library initialization. This is done in `src/lib.c`, which is a very simple file that just calls several initialization functions. Those functions are defined in different files under `src/`;
- `build_scheme_args` – collects the command line options in the variable **`%system-state-plist`**;
- `STk_load_boot` – loads the boot file (if one is to be loaded);
- `STk_boot_from_C` – actually boots the Scheme interpreter. This function is defined in `src/vm.c`, where the STklos virtual machine code is.

In order to include Scheme code for execution during STklos startup, edit `lib/boot.stk`.

Chapter 5. Adding simple modules and SRFIs

5.1. Adding modules

- add your `fantastic-module.stk` to `lib/SUBDIR`, where `SUBDIR` could be `scheme`, `srfi` or `stklos` (see nect subsection)
- include `fantastic-module.stk` and `fantastic-module.ostk` in the variables `SRC_STK` and `scheme_OBJS`, in `lib/Makefile.am`
- Tests reside in the `tests` directory. Create a new file in `tests` directory and include it in the list of loaded files in `do-test.stk`

5.2. Module placement in the tree

- STklos modules go into `lib/stklos`
- Scheme (R7RS small or large) libraries go into `lib/scheme`
- SRFIs go into `lib/srfi`

5.3. Adding SRFIs

In order to add SRFI 9999 to STklos,

- add your `9999.stk` to `lib/srfi`
- include `9999.stk` and `9999.ostk` in the variables `SRC_STK` and `SRC_OSTK`, in `lib/srfi/Makefile.am`
- Add a line describing it in `lib/srfis.stk` (the format is described in the file itself).
- Tests reside in the `tests` directory. Add the tests in a file `tests/srfis/9999.stk`

For new SRFIs, adding its description in `lib/srfis.stk` suffices to update

- the `SUPPORTED-SRFIS` in the main directory
- launch the tests you added in `tests/srfis` directory, and
- add an automatically generated documentation for this SRFI

5.4. Mixed SRFIs (Scheme and C)

To add a mixed SRFI 8888,

- Write a `8888.c` file and put it in `lib/srfi`
- Write a `8888.stk` Scheme file and also put it in `lib/srfi`
- Add your mixed SRFI to `lib/srfi/Makefile.am`, in the section `` `SRFIs written in C and Scheme'' (variables `SRC_C `` , `SRC_C_STK`, and `SRC_SHOBJ`

5.4.1. Content of the Scheme file

The Scheme file will be compiled as a byte-code stream embedded in C. Here, the compiled file will be called `$DIR/srfi-170-incl.c`. It is built by the `utils/tmpcomp` script with

```
../../utils/tmpcomp -o srfi-170-incl.c $DIR/srfi-170.stk
```

Note: when the destination file ends with a `.c` suffix, the `tmpcomp` command produces a C file instead of a byte-code file.

You don't have to pay attention to any particular point in the writing of this file.

5.4.2. Content of the C file

The C file must follow the conventions of dynamically loadable code as shown in the example in the `/etc` directory.

In this C file, to use the previously compiled Scheme code, you have to (using SRFI 170 as an example):

- include the file `170-incl.c` at the top of your C file
- add a call to execute the Scheme code just before the `MODULE_ENTRY_END` directive. This is done with the following invocation:

```
    STk_execute_C_bytecode(__module_consts, __module_code);
```

- Add a directive `DEFINE_MODULE_INFO` at the end of the file. It permits to access some information of the module (STklos version used to compile the module, exported symbols, ...). For now, this information is not used, but omitting to add this directive will probably lead to a compiler warning about an unresolved reference.

As one more example, SRFI 25 has, at the end of the C file:

```
MODULE_ENTRY_START("srfi/25")
{
  SCM module =  STk_create_module(STk_intern("srfi/25"));
  STk_export_all_symbols(module);

  ADD_PRIMITIVE_IN_MODULE(...);
  ...
  ...

  /* Execute Scheme code */
  STk_execute_C_bytecode(__module_consts, __module_code);
}
MODULE_ENTRY_END
```

```
DEFINE_MODULE_INFO
```

See SRFI-25, SRFI-27 and SRFI-170 as a reference.

5.5. Documentation

5.5.1. Documenting SRFIs in `srfi.adoc`

General documentation is automatically generated for SRFIs. If you need to give a precision specific to a given SRFI, add it to the end of the `doc/refman/srfi.adoc` file using the `gen-srfi-documentation` function.

Note that the documentation is written in Skribe tool which is no more maintained. Consequently, the documentation will not be generated. The HTML and PDF documentation is rebuilt from time to time by @egallesio.

5.5.2. Documenting primitives written in C

Before `DEFINE_PRIMITIVE`, add a comment similar to the others you see in the C files. An example:

```
/*
<doc EXT bignum?
 * (bignum? x)
 *
 * This predicates returns |#t| if |x| is an integer number too large to be
 * represented with a native integer.
 * @lisp
 * (bignum? (expt 2 300))     => |#t|   (very likely)
 * (bignum? 12)               => |#f|
 * (bignum? "no")             => |#f|
 * @end lisp
doc>
*/
DEFINE_PRIMITIVE("bignum?", bignump, subr1, (SCM x))
{
  return MAKE_BOOLEAN(BIGNUMP(x));
}
```

Pay attention to the parts of this comment: it begins with the primitive name, then there's an explanation, then examples in Scheme. Wrap symbols/identifiers in `|.|`; use `@lisp` and `@end lisp@` to show an example of usage.

Chapter 6. Writing primitives in C

Use the macro `DEFINE_PRIMITIVE`:

```
DEFINE_PRIMITIVE("fixnum?", fixnump, subr1, (SCM obj))
{
  return MAKE_BOOLEAN(INTP(obj));
}
```

The arguments for this example are

- Scheme name
- C function name (its full name will have the string ```STk_'' prepended to it)
- the type of primitive (in this case, it is a subroutine with one parameter – ```subr1''
- the arguents, surrounded by parentheses. In this case there is only one argument, `` `obj'', and its type is `` `SCM'' (which is the type of all Scheme objects in STklos).

Then add it:

```
ADD_PRIMITIVE(fixnump);
```

The name passed to `ADD_PRIMITIVE` is the C function name.

6.1. Calling Scheme primitives

Recall that a primitive is defined like this:

```
DEFINE_PRIMITIVE("fixnum?", fixnump, subr1, (SCM obj))
{ ... }

ADD_PRIMITIVE(fixnump);
```

To use this primitive later in C code, add the `STk_` prefix to its C function name:

```
if (STk_fixnump(obj) == STk_false) ...
```

6.2. Returning multiple values

`STk_n_values(n, v1, v2, ⋯, vn)` returns `n` values from a procedure.

For example, `read-line` (defined in `port.c`) has these two lines:

```
return STk_n_values(2, res, STk_eof)
```

for when it found the end of the file, and

```
return STk_n_values(2, res, delim);
```

for when it did not yet reach EOF, so it returns the line delimiter as second value.

6.3. Using multiple returned values

Just as one can use `STk_n_values` to produce values, it is also possible to call (from C) a Scheme procedure that produces a sequence of values and use them from the C code. The function `STk_values2vector` (defined in `vm.c`) does this.

In Scheme, one could to this:

```
(define (my-proc x y z)         ;; takes three arguments
  (values (+ x y) (- y z)))     ;; returns two values
```

If we assume that the C `SCM` variable `proc` points to the closure `my-proc`, then we can call it like this:

```
SCM a = MAKE_INT(10);
SCM b = MAKE_INT(20);
SCM c = MAKE_INT(30);

/* Define a Scheme vector to hold EXACTLY two values: */
SCM results = STk_makevect(2, NULL);

VECTOR_DATA(results)[0] = STk_false;
VECTOR_DATA(results)[1] = STk_false;

/* Call the procedure proc, passing 3 arguments; proc */
STk_values2vector ( STk_C_apply(proc, 3, a, b, c),
                    results );
```

The Scheme vector `results` will then hold the two returned values.

- If you pass `NULL` as second argument to `STk_values2vector` instead of passing a vector, the VM will allocate a vector with the size of the number of values returned.
- If you do pass a vector to `STk_values2vector`, then the procedure being called **must** produce **exactly** that number of values (not more, not less), otherwise the VM will signal an error.

6.4. Errors

The C function that raises errors is

- `STk_error(fmt, arg1, arg2, ...)` – the STklos error procedure. `fmt` is a format string, and after it there are arguments.

But as you can see in the top of several C files, it is useful to define wrappers:

```
static void error_bad_number(SCM n)
{
  STk_error("~S is a bad number", n);
}

static void error_at_least_1(void)
{
  STk_error("expects at least one argument");
}

static void error_cannot_operate(char *operation, SCM o1, SCM o2)
{
  STk_error("cannot perform %s on ~S and ~S", operation, o1, o2);
}
```

6.5. Unboxed types

The trditional way to representa data in Lisp languages is by *tagged objects*. A long enough machine word is used to represent all types, and some bits are reserved to distinguish the type of the object. In STklos, the *two least significant bits* are used for this.

- `00` - pointer on an object descriptor (a box)
- `01` - fixnum
- `10` - small object (characters and others)
- `11` - small constant (`#t`, `#f`, `'()`, `#eof`, `#void`, dot, close-parenthesis)

The idea is that checking the type of these should be very fast, because it is done at runtime, so to check wether an object is `#eof`, one needs only check if `obj & 0x4 == 0x3` (but usually, we have macros for that).

STklos uses C `long` words so, for example, in a machine where `long int` is 32 bits long the bit sequence

```
0000 0000 0000 0000 0000 0000 0010 0101
```

is a *fixnum* (because its two least significant digits are `01`, and the value of the fixnum is 9 (because after discarding the `01` that is on the right of the sequence, the number left is `1001`).

6.5.1. Booleans

- `STk_true` is the SCM object for `#t`
- `STk_false` is the SCM object for `#f`
- `BOOLEANP(o)` checks wether the object `o` is boolean (the macro actually does `(o) == STk_true) || ((o) == STk_false`
- `MAKE_BOOLEAN(_cond)` expands to a conditional statement: if `_cond` is true, then the value is `STk_true`, otherwise it is `STk_false`.

6.5.2. Fixnums

Fixnums are not allocated but have their two least significant bits set to `01` (in Lisp-parlance, it has `01` as its *tag*).

- `INTP(o)` - returns STklos_true if `o` is a Scheme integer or `STklos_false` otherwise
- `MAKE_INT(n)` - takes a `long` C number and turns it into an `SCM` integer object. Actually, this will shift the number to the left by two positions and insert the tag If we could represent numbers as binary in C, it would be like this:

```
MAKE_INT( 000011000 )  // --> 001100001
```

- `INT_VAL(o)` - returns the value of the fixnum `o`, as a C `long` value (the opposite of the previous operation)

6.6. Boxed types

Boxed types are anything except for fixnums, small objects and small constants. They are tagged with `00`.

- `BOXED_OBJP(o)` – true if `o` is a boxed object
- `BOXED_TYPE_EQ(o,t)` – checks wether `o` is a boxed object of type `t`
- `BOXED_TYPE(o)` – returns the type of boxed object `o`
- `BOXED_INFO` – returns the information of boxed object `o`

The type definition for all possible types, in `stklos.h`, is self-explanatory:

```
typedef enum {
  tc_not_boxed=-1,
  tc_cons, tc_integer, tc_real, tc_bignum,  tc_rational,                /* 0 */
  tc_complex, tc_symbol, tc_keyword, tc_string, tc_module,              /* 5 */
  tc_instance, tc_closure, tc_subr0, tc_subr1, tc_subr2,                /* 10 */
  tc_subr3, tc_subr4, tc_subr5, tc_subr01, tc_subr12,                   /* 15 */
  tc_subr23, tc_vsubr, tc_apply, tc_vector, tc_uvector,                 /* 20 */
  tc_hash_table, tc_port, tc_frame, tc_next_method, tc_promise,         /* 25 */
  tc_regexp, tc_process, tc_continuation, tc_values, tc_parameter,      /* 30 */
```

```
  tc_socket, tc_struct_type, tc_struct, tc_thread, tc_mutex,            /* 35 */
  tc_condv, tc_box, tc_ext_func, tc_pointer, tc_callback,               /* 40 */
  tc_last_standard /* must be last as indicated by its name */
} type_cell;
```

6.6.1. Lists

Here are some primitives for lists, for example:

- `CAR(p)` – equivalent to Scheme `car`: returns the car of `p` (an SCM object)
- `CDR(p)` – equivalent to Scheme `cdr`: returns the car of `p` (an SCM object, which certainly is a list)
- `CONSP(p)` - equivalent to Scheme `cons?`
- `NULLP(p)` - equivalent to Scheme `null?`
- `STk_cons` - equivalent to Scheme `cons`

6.6.2. Strings

Another example are strings. They are defined as the following structure:

```
struct string_obj {
  stk_header header;
  int space;            /* allocated size  */
  int size;             /* # of bytes used */
  int length;           /* "external" length of the string */
  char *chars;
};
```

Then, some primitives:

```
#define STRING_SPACE(p)  (((struct string_obj *) (p))->space)
#define STRING_SIZE(p)   (((struct string_obj *) (p))->size)
#define STRING_LENGTH(p) (((struct string_obj *) (p))->length)
#define STRING_CHARS(p)  (((struct string_obj *) (p))->chars)
#define STRINGP(p)       (BOXED_TYPE_EQ((p), tc_string))
```

The following primitives are defined in a `str.c`, but `stklos.h` is used by several files use them, so they're included with `EXTERN_PRIMITIVE`:

```
EXTERN_PRIMITIVE("string=?", streq, subr2, (SCM s1, SCM s2));
EXTERN_PRIMITIVE("string-ref", string_ref, subr2, (SCM str, SCM index));
EXTERN_PRIMITIVE("string-set!", string_set, subr3, (SCM str, SCM index, SCM value));
EXTERN_PRIMITIVE("string-downcase!", string_ddowncase, vsubr, (int argc, SCM *argv));
```

6.7. Dynamically loadable modules

See some examples in etc/

6.8. Input and output from C

The input and output functions are defined in sio.c, and declared in stklos.h. For example,

- STk_getc(SCM port) for reading a single character
- STk_get_character(SCM port) for reading a single character (result may be a wide char)
- STk_putc(int c, SCM port) for printing a single character
- STk_put_character(int c, SCM port) for printing a single character (maybe a wide char)
- STk_puts(const char *s, SCM port) for printing a C string
- STk_putstring(const char *s, SCM port) for printing a Scheme string
- STk_print(SCM exp, SCM port, int mode) for printing Scheme objects
- STk_print_star(SCM exp, SCM port, int mode) for circular structures

All printing procedures have a port argument. This should be a Scheme object of the type port, and there are also already defined ports for standard output and error, STk_stdout and STk_stderr. For reading there is also STk_stdin. These standard ports are defined in fport.c, and declared (as extern) in stklos.h. They are all initialized in the function STk_init_fport in fport.c.

Some printing procedures have a mode argument. The two allowed values for this are WRT_MODE and DSP_MODE, which correspond to "write mode" (which will write the raw representation of objects) and "display mode" (which will do pretty-printing). The difference can be clearly seen in the printstring function in print.c:

```
static void printstring(SCM s, SCM port, int mode)
{
  if (mode == DSP_MODE) {
    STk_putstring(s, port);
  } else {
    /* lots of code dealing with character escapes */
  }
```

6.9. Creating new types

6.9.1. Example: SRFI-25

We'll be using SRFI-25 as an example. In that SRFI, am array type is created.

- Create a C struct whose first field is of type stk_header

```
struct array_obj {
```

```
  stk_header header;
  int shared;               /* does this array share data with another? */
  int *orig_share_count;    /* pointer to original array share counter */
#ifndef THREADS_NONE
  MUT_FIELD(share_cnt_lock); /* lock for share counter */
  MUT_FIELD(*share_cnt_lock_addr); /* pointer to mutex - ours or of original array's
*/
#endif
  long size;                /* size of data */
  long length;              /* # of elements */
  int  rank;                /* # of dimensons */
  long offset;              /* offset from zero, to be added when calculaing index */
  long *shape;              /* pairs of bounds for each dimenson */
  long *multipliers;        /* size of each dimension stride */
  SCM  *data_ptr;           /* pointer to data */
};
```

The fields in the struct may contain both C and Scheme elements (the Scheme elements have `SCM` types).

- Maybe create some accessor macros

```
#define ARRAYP(p)             (BOXED_TYPE_EQ((p), tc_array))
#define ARRAY_SHARED(p)       (((struct array_obj *) (p))->shared)
#define ARRAY_SHARE_COUNT(p)  (((struct array_obj *) (p))->orig_share_count)
#define ARRAY_LOCK(p)         (*(((struct array_obj *) (p))->share_cnt_lock_addr))
#define ARRAY_SIZE(p)         (((struct array_obj *) (p))->size)
#define ARRAY_LENGTH(p)       (((struct array_obj *) (p))->length)
#define ARRAY_RANK(p)         (((struct array_obj *) (p))->rank)
#define ARRAY_OFFSET(p)       (((struct array_obj *) (p))->offset)
#define ARRAY_SHAPE(p)        (((struct array_obj *) (p))->shape)
#define ARRAY_MULTS(p)        (((struct array_obj *) (p))->multipliers)
#define ARRAY_DATA(p)         (((struct array_obj *) (p))->data_ptr)
```

Be mindful of thread-related things: not all STklos builds have threading enabled!

```
#ifdef THREADS_NONE
#  define ARRAY_MUTEX(p)
#  define ARRAY_MUTEX_SIZE 1
#else
#  define ARRAY_MUTEX(p) (((struct array_obj *) (p))->share_cnt_lock)
#  define ARRAY_MUTEX_SIZE (sizeof(pthread_mutex_t))
#  define ARRAY_MUTEX_PTR_SIZE (sizeof(pthread_mutex_t*))
#endif
```

- Create an extended type descriptor which contains the type name, and pointers to functions to print and compare elements:

```c
static void print_array(SCM array, SCM port, int mode)
{
  /*
    Here goes the code for printing array.
    Use the functions
      - STk_puts(char *str, SCM port)
      - STk_print(SCM obj, SCM port, int mode)
    It may be useful to first create a buffer, use snprintf on it, then
    use STk_puts to print it.
   */
}
```

```c
static SCM test_equal_array(SCM x, SCM y)
{
 /*
   Code that retruns STk_true if x and y are to be considered `equal?`,
   and STk_false othereise.

   NOTE: remember to *NOT* return 0 or 1. The return value should be a Scheme
         object, not a C value with the intended boolean value. This is
         particularly important because the compiler will *NOT* warn you if you
         return "0":
         - `SCM` is defined as a pointer to `void`
         - '0' can be interpreted as a pointer, so the compiler thinks it's OK
         - '0' is *not* the same as `STk_void`
  */
}
```

```c
static struct extended_type_descr xtype_array = {
  .name  = "array",
  .print = print_array,
  .equal = test_equal_array
};
```

- At the end of your C code, inside the MODULE_ENTRY_START part, initialize an element of the new type: `tc_array = STk_new_user_type(&xtype_array);`
- Create a describing procedure:

```scheme
(%user-type-proc-set! 'array 'describe
                      (lambda (x port)
                        (format port "an array of rank ~A and size ~A"
                                (array-rank x)
                                (array-size x))))
```

- Define a class, and associate it with the type name you have created.

```
(define-class <array> (<top>) ())
(export <array>)

(%user-type-proc-set! 'array 'class-of <array>)
```

- If objects of the new type will have a printed representation, create a reader procedure:

```
(define-reader-ctor '<array>
  (lambda args
    (apply array (apply shape (car args)) (cdr args))))
```

6.9.2. More about creating new types

The structure for extended type descriptors is defined in `stklos.h`, in section "EXTEND.C":

```
struct extended_type_descr {
  char *name;
  void (*print)(SCM exp, SCM port, int mode);
  SCM  (*equal)(SCM o1, SCM o2);
  SCM  (*eqv)(SCM o1, SCM o2);
  SCM  class_of;
  SCM  describe_proc;
};
```

As can be seen, there are other fields besides `name`, `print` and `equal` that can be customized. For example, the `describe` behavior, which was defined in Scheme for SRFI-25, could have been implemented in C.

Immediately below the definition of this structure, there are also some useful macros and function declarations for dealing with extended types.

Chapter 7. Continuations

One macro and two functions are declared in `vm.h` that can be used to capture, check and restore continuations:

- `CONTP(k)` verifies (as expected) wether `k` is a continuation object
- `SCM STk_make_continuation(void)` returns the current continuation
- `SCM STk_restore_cont(SCM cont, SCM val)` restores continuation `cont`, passing it the value `val`

There is also one function in `vm.c` which is not exported by `vm.h`, but is available as a Scheme primitive:

```
DEFINE_PRIMITIVE("%fresh-continuation?", fresh_continuationp, subr1, (SCM obj))
{
  return MAKE_BOOLEAN(CONTP(obj) && (((struct continuation_obj *) obj)->fresh));
}
```

Their Scheme counterparts, `%continuation?`, `%make-continuation`, and `%restore-continuation` are used to implement the Scheme procedure `call/cc` (in `lib/callcc.stk`). The implementation of `call/cc` is actually complex because it needs to be intertwined with the implementation of `dynamic-wind`, but in the same file there is another procedure, `%call/cc`, which does not do winding, and is therefore very simple (and it should be the starting point to understand the full-blown `call/cc`). We reproduce it here with some comments.

```
(define (%call/cc proc)
  (let ((k  (%make-continuation)))
    (if  (%fresh-continuation? k)

         ;; In the first time we get here, we create a closure (the lambda
         ;; below) that will take a value v and restore the continuation
         ;; k with it. So when we call
         ;; (%call/cc (lambda (kont) ... (kont x) ...)),
         ;; 'proc' below is the '(lambda (kont) ...)' in our code. And the
         ;; '(lambda v ...)' below is kont. 'v' is the argument that will be
         ;; given to kont.

         (proc (lambda v (%restore-continuation k v)))

         ;; Next time and everytime again, we just return values applied to k,
         ;; because in this case, k will *not* be a continuation, but a list
         ;; with the values passed (this is because the lambda above accepts
         ;; 'v' as the arg list, and this list is passed to %restore-continuation
         ;; as the value to be returned).
         (apply values k))))
```

The `%call/cc` procedure is used in the same way the Scheme `call/cc` is used:

```
stklos> (define c #f)
(let ((a 1)
      (b 2))
  (format #t "start~%")
  (set! b (%call/cc (lambda (k)
                      (set! c k)
                      -1)))
  (set! a (+ 1 a))
  (format #t "~a ~a ~a~%" a b c))

start
2 -1 #[closure 7fbcd9a122c0]

stklos> (c 15)
3 15 #[closure 7fbcd9a122c0]

stklos> (c 'x)
4 x #[closure 7fbcd9a122c0]
```

The behavior of the fundamental continuation procedures is better illustrated by an example in Scheme, which mimics the example of `%call/cc` given above, **ecxept** that it does not have the return value of `%call/cc`, so it does not set the value of `b`:

```
stklos> (define c #f)  ; to be set later
(let ((a 1)
      (b 2))
  (format #t "start~%")
  (set! c (%make-continuation))
  (set! a (+ 1 a))
  (format #t "~a ~a ~a~%" a b c))

start
2 2

stklos> (%continuation? c)
#t

stklos> c
#[continuation (C=3992 S=1512) c069e000]      ;; addresses: C stack, Scheme stack,
                                              ;; continuation object

stklos> (%fresh-continuation? c)
#t

stklos> (%restore-continuation c c)           ;; since this is the continuation of
                                              ;; "(set! c ...)", we put "c" as value,
                                              ;; so we can use the continuation again
3 2 #[continuation (C=3992 S=1512) c069e000]
```

```
stklos> (%fresh-continuation? c)
#f

stklos> (%restore-continuation c c)
4 2 #[continuation (C=3992 S=1512) c069e000]

stklos> (%restore-continuation c c)
5 2 #[continuation (C=3992 S=1512) c069e000]

stklos> (%restore-continuation c c)
6 2 #[continuation (C=3992 S=1512) c069e000]
```

Chapter 8. The virtual machine

See the file `vm.adoc` for a description of the opcodes.

Chapter 9. Compiler and optimizations

9.1. The compiler

The compiler is in the file `lib/compiler.stk`.

There is a `compile` procedure at the end of the file, whose logic is very simple:

1. expand macros
2. compile special forms
3. if what's left is a symbol, compile a call
4. if it's not a symbol, compile it as a constant

In the rest of the file, there are procedures to compile different special forms and inlinable primitives.

The code is generated as a list, in the **`code-instr`** global variable in the `STKLOS-COMPILER` module. The procedure `emit` conses one more instruction on the code (which will later be reversed, of course)

9.2. Peephole optimizer

STklos uses a peephole optimzier, located in the file `lib/peephole.stk`. This optimizer will transform several instruction patterns in the generated code into more efficient ones. For example:

```
;; [SMALL-INT, PUSH] => INT-PUSH
((and (eq? i1 'SMALL-INT) (eq? i2 'PUSH))
 (replace-2-instr code (list 'INT-PUSH (this-arg1 code))))
```

This transforms two instructions (`` `load a small integer into `val ``, then push it onto the stack") into one single instruction (push an integer onto the stack).

The peephole optimizer also reduces the size of the bytecode:

```
;; [RETURN; RETURN] => [RETURN]
((and (eq? i1 'RETURN) (eq? i2 'RETURN))
 (replace-2-instr code (list 'RETURN)))
```

This will turn two adjacent `RETURN` instructions into a single one, making the object file smaller. This is valid because there won't be any `GOTO` pointing to the second instruction; if this was the case, then the code would have a label between the two \`RETURN\`s.

Another example is `GOTO` optimization:

```
;; [GOTO x], ... ,x: GOTO y => GOTO y
;; [GOTO x], ... ,x: RETURN => RETURN
```

```
             ((eq? i1 'GOTO)
              (set! code (optimize-goto code)))
```

The procedure `optimize-goto-code`, also in the file `peephole.stk`, will perform the transformations indicated in the comments.

The input code is represented as a list of the form

```
( (instruction1 op1 op2)     ;; one instruction with two operands
  (instruction2 op1)         ;; one instruction with one operand
  (instruction3)             ;; one instruction with no operands
  ...
  (instruction10 op1 op2)
  10                         ;; this is a label!
  (instruction11)
  ... )
```

Some relevant definitions are in the beginning of the file:

```
(label? code)      ; is the current instruction a label?
(this-instr code)  ; the current instruction (reference to a position in the list)
(next-instr code)  ; the next instruction (cdr of the current one)
(this-arg1 code)   ; argument 1 of current instruction
(this-arg2 code)   ; argument 2 of current instruction
(next-arg1 code)   ; argument 1 of next instruction
(next-arg2 code)   ; argument 2 of next instruction
```

There are only procedures for dealing with the current and the next instruction because the peephole optimizer currently only substitutes sequences of two instructions. It is an interesting exercise to try to implement three-instruction peephole operation. As a suggestion, the reader can try the following:

- Implement `third-instr`. Be careful to not try to take the `cdr` of an empty list…
- Include one more optimization clause in the optimizer that performs the substitution `[IN-CDR; IN-CDR; IN-CDR] ⇒ IN-CDDDR`
- And of course, implement `IN-CDDDR`:
  - Change `lib/assembler.stk`
  - Change `src/vm-instr.h`
  - Add one more case to the VM state machine (use the case for `IN_CDR` as a starting point).
- Finally, write some benchmark to verify if the new optimization is worth the trouble (and the use of a new opcode).

9.3. Source rewriter

The file `lib/comprewrite.stk` contains rules for code rewriting.

All the rewriting rules are stored in an compiler internal hash table, whose keys are symbols The value stored for key `SYMBOL` is a procedure that transforms the form `(SYMBOL …)` into something else. For example, it will transform `(IF 1 2 3)` into `2`. The procedure takes as parameters:

- An expression (whose first element is the symbol that was used as key in the hash table);
- The length of the expression;
- The environment.

The procedure should, of course, return the optimized expression. This procedure can be obtained by the function `compiler:find-rewriter`, as seen below:

```
(define rewrite-car (compiler:find-rewriter 'car))
(rewrite-car '(car '(1 2)) 2 (interaction-environment)) => '1

(define rewrite-not (compiler:find-rewriter 'not))
(rewrite-not '(not #t) 1 (interaction-environment))
```

If the expression doesn't seems correct, or cannot be simplified, nothing is done (since the rewriter is not where syntax or semantic errors are detected):

```
(rewrite-car '(car '(1 2)) 2 (interaction-environment)) => '1
(rewrite-car '(car 1 2 3)  4 (interaction-environment)) => '(car 1 2 3)
(rewrite-car '(car a-list) 1 (interaction-environment)) => '(car a-list)
```

The function `compiler:add-rewriter!` adds a new rewriting rule to the compiler. For instance, we can add a rule that transforms the calls to the `eof-object` standard primitive to the STklos constant `#eof` (this rewriter is already defined in the compiler)

```
(compiler:add-rewriter!            ;; 'EOF-OBJECT' rewriter
 'eof-object
 (lambda (expr len env)
   (if (= len 1)
       #eof
       expr)))

(define eof-rewriter (compiler:find-rewriter 'eof-object))
(eof-rewriter '(eof-object)   1 (interaction-environment)) => #efo
(eof-rewriter '(eof-object 1) 2 (interaction-environment)) => (eof-object 1)
```

The parameter object `compiler:source-rewrite` can be used to control source rewriting.

```
stklos> (compiler:source-rewrite #f)
stklos> (disassemble-expr '(car '(1 2)))

000:  CONSTANT            0      ;; that is the list '(1 2)
002:  IN-CAR
stklos> (compiler:source-rewrite #t)
stklos> (disassemble-expr '(car '(1 2)))

000:  IM-ONE
```

Rewriting rules can be defined without modifying the compiler thanks to the following functions

- `(compiler:const-expr? e)` which returns `#t` if the expression `e` is constant
- `(compiler:const-value e)` which returns the value of the constant expression `e`
- `(compiler:rewrite-expression e env)` which returns a simplified version of expression `e` in the environment `e`.

We are now able to write a rewriting rule for `not`:

```
(compiler:add-rewriter!            ;; 'NOT' rewriter
 'not
 (lambda (expr len env)
   (if (= len 2)
       (let ((val (compiler:rewrite-expression (cadr expr) env)))
         (if (compiler:const-expr? val)
             (not (compiler:const-value val))
             expr))
       expr)))

(compiler:rewrite-expression '(not 42)            (interaction-environment))
          => #f
(compiler:rewrite-expression '(not (not 42))      (interaction-environment))
          => #t
(compiler:rewrite-expression '(if (not 42) 10 12) (interaction-environment))
          => 12
```

Chapter 10. Garbage collection

STklos uses the Boehm-Demers-Weiser garbage collector. The wrapper for the GC is located in the header file `src/stklos.h`:

```
#define STk_must_malloc(size)           GC_MALLOC(size)
#define STk_must_malloc_atomic(size)    GC_MALLOC_ATOMIC(size)
#define STk_must_realloc(ptr, size)     GC_REALLOC((ptr), (size))
#define STk_free(ptr)                   GC_FREE(ptr)
#define STk_register_finalizer(ptr, f)  GC_REGISTER_FINALIZER(
                                            (void *) (ptr),
                                            (GC_finalization_proc)(f),
                                            0, 0, 0)
#define STk_gc()                        GC_gcollect()

void STk_gc_init(void);
```

- `STk_must_malloc` - used to allocate structured objects.
- `STk_must_malloc_atomic` - used when there won't be any pointers inside the object, and we don't want to confuse the GC with patterns that are supposed to be just a bignum, but ``look like apointer''. Used for strings, numbers etc.
- `STk_register_finalizer` will register a finalizer function `f`, which will be called when the object at `ptr` is collected.

Chapter 11. C variables for conditional compilation

These are simple to understand, but we ist them here anyway.

11.1. Detecting libffi, libgmp, dynamic loading

- `libffi`: the `configure` script will set the `HAS_FFI` variable when libffi is available. In `ffi.c`, for example, the code that actually uses `libffi` is guarded by an `ifdef`

```
#ifdef HAVE_FFI
    /* use libffi here */
#else /* HAVE_FFI */
static void error_no_ffi(void)
{
  STk_error("current system does not support FFI");
}
...
DEFINE_PRIMITIVE("make-callback", make_callback, subr3, (SCM p1, SCM p2, SCM p3))
{ error_no_ffi(); return STk_void;}
...
#endif
```

- `libgmp`: In `number.c`, STklos includes “gmp.h”. This header file may be provided either by `mini-gmp` or by the full GMP. WHen the `mini-gmp` is used, the variable `MINI_GMP_H` is defined, so for example this is tone in `number.c`:

```
#ifdef __MINI_GMP_H__
    /* BEGIN code for compiling WITH MINI GMP (*no* rationals!) */
    ...
#else
    /* BEGIN code for compiling WITH FULL GMP (*with* rationals!) */
    ...
#endif /* __MINI_GMP_H__ */
```

- In `dynload.c`, the variable `HAVE_DLOPEN` is used to guard the full contents of the file.

11.2. Statistics gathering

In `vm.c`, code that does statistics gathering is guarded by `STAT_VM`. For example,

```
#ifdef STAT_VM
static int couple_instr[NB_VM_INSTR][NB_VM_INSTR];
static int cpt_inst[NB_VM_INSTR];
static double time_inst[NB_VM_INSTR];
```

```c
static int collect_stats = 0;
static void tick(STk_instr b, STk_instr *previous_op, clock_t *previous_time);
#endif
```