Extending the STk interpreter

Erick Gallesio
Université de Nice - Sophia-Antipolis
Laboratoire I3S - CNRS URA 1376 - ESSI.
Route des Colles
B.P. 145
06903 Sophia-Antipolis Cedex - FRANCE

email: eg@unice.fr

July 1995

Abstract

This document describes how to extend the STk interpreter with new primitives procedures and/or new types. Extending the interpreter can be done by writing new *modules* in C. New C code can be statically linked to the core interpreter or dynamically loaded on operating systems which support shared libraries. This document also presents how to integrate new Tk widgets written for the Tcl interpreter in STk.

Contents

1 Introduction

This document describes how to extend the STK[2] interpreter using the C language[4]. To begin, we will start with a simple extension which will only consist to add some simple new primitives to the interpreter. Second section will describe how to add a new type (and the primitives for manipulating this new type). Another interesting extension consists to add new kind of primitives (i.e. primitives which evaluate their argument in particular way). This kind of extension will be discussed in the third section. Fourth section discusses how to add a new widget to the interpreter. Calling some Scheme code from a C function is showed in section 5. And last, we will show how to load an extension at load time. This facility will permit to extend the STK interpreter without having to recompile it, on systems which support dynamic loading.

2 Adding new primitives

2.1 A simple example

One of the simpler extension one can wish to do consists to add new primitives procedures to the interpreter. To illustrate this, suppose we want to add two new primitives to the interpreter: `posix-time` and `posix-ctime`. The former function correspond to the POSIX.1[1] function `time`: it returns the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). The latter is a wrapper around the POSIX.1 function `ctime` which returns a string containing the current time in an human readable format.

First, we will see how to write the new Scheme primitive `posix-time`. Implementing a new primitive requires to write a new C function which will do the work. Here, we write the C function `posix_time` to implement the Scheme primitive `posix-time`. The code of this function is given below.

```
static PRIMITIVE posix_time(void)
{
  return STk_makeinteger((long) time(NULL));
}
```

This function uses the interpreter `STk_makeinteger` function which converts a C long integer to a STK integer. Once the `posix_time` C function is written, we have to bind this new primitive to the Scheme symbol `posix-time`. This is achieved by the following C function call.

```
STk_add_new_primitive("posix-time", tc_subr_0, posix_time);
```

Note: The C type `SCM` is used to describe the objects manipulated in Scheme. `PRIMITIVE` is an alias for this type; it is preferably used when defining a new primitive.

`STk_add_new_primitive` tells the interpreter that the Scheme symbol `posix-time` must be bound to the (C written) primitive `posix_time`. The constant `tc_subr_0` used as the second argument indicates the arity of this primitive. In this case, the arity of the primitive is 0.

Let's now have a look at the primitive `posix-ctime`. A first writing of this primitive could be

```
static PRIMITIVE posix_ctime(void)
{
  char *s;
  time_t t = time(NULL);

  s = ctime(&t);
  return STk_makestring(s);
}
```

This functions uses another interpreter routine (`STk_makestring`) which takes as parameter a null terminated string and returns a Scheme string.

Binding of the scheme symbol `time-string` to the C function `get_time` is done by the call

```
STk_add_new_primitive("posix-ctime", tc_subr_0, posix_ctime);
```

A complete listing of this code is given in Figure 2.1. Provided that we have done a shared object of this file, and that its name is `posix.so`, our two new primitives can be loaded dynamically by:

```
(load "time.so")
```

Notes:

- Suffix can be omitted. Suffixes given in the Scheme variable `*load-suffixes*` gives the order in which suffixes must be tried for loading a file. Default value for this variable is `("stk" "stklos" "scm" "so")`.
- When dynamic loading is used, the interpreter try to call a function whose name is equal to the string `"STk_init_"` followed by the name of the file, without suffix. Definitions of new primitives are generally done in this function. Here, the C function in charge of module initialization must be called `STk_init_posix`.

2.2 Passing arguments to a primitive

This section shows how to pass arguments to a new primitive written in C. To illustrate our purpose, we will rewrite the primitive `posix-ctime` to be conform to POSIX.1 (this function should take an integer, a count of seconds, and should return corresponding date as a string). A second writing of previous function could be:

```
static PRIMITIVE posix_ctime(SCM seconds)
{
  long sec;

  sec = STk_integer_value_no_overflow(seconds);
  return STk_makestring(ctime((time_t *)&sec));
}
```

This function has one parameter since Scheme primitive arity is one. The C primitives parameters are always `SCM` objects. An object of this type is a pointer to a `struct obj`: the type which permits to represent all the Scheme objects. The `SCM` and `struct obj` types definitions can be found in the `Src/stk.h` header file.

The first job of this function consists to convert the Scheme parameter (`seconds`) to a C integer `long int`. This is done with the function `STk_integer_value_no_overflow`which takes a `SCM` and

```
#include <sys/types.h>
#include <sys/time.h>
#include <time.h>
#include <stk.h> /* Declaration of STk objects/primitives */

static PRIMITIVE posix_time(void)
{
  return STk_makeinteger((long) time(NULL));
}

static PRIMITIVE posix_ctime(void)
{
  char *s;
  time_t t = time(NULL);

  s = ctime(&t);
  return STk_makestring(s);
}

PRIMITIVE STk_init_posix(void)
{
  STk_add_new_primitive("posix-time",  tc_subr_0, posix_time);
  STk_add_new_primitive("posix-ctime", tc_subr_0, posix_ctime);
  return UNDEFINED;
}
```

Figure 1: A first version of file `posix.c`

returns a `long int`. This functions returns `LONG_MIN` if the argument is not a an integer number (or a number which don't fit in the C representation of a C `long int`). Once this conversion is done, the rest of the job is similar to the code presented above.

To add this primitive to the global Scheme environment, we have to change the previous `STk_add_new_primitive` for this primitive by:

```
STk_add_new_primitive("posix-ctime", tc_subr_1, posix_ctime);
```

in the init section. This call states that the type of this primitive is fixed to a `tc_subr_1` (a arity-1 primitive).

However, this function is not too satisfying, even if close to the POSIX definition: it obliges to pass a parameter which will be probably most of the time the result of the primitive `posix-time` (i.e. the most frequent usage of this function will be

```
(posix-ctime (posix-time))
```

which is not very elegant). A better approach consists to allow this primitive to have a optional parameter. This permits to be at the same time conform to the POSIX convention and close to Scheme habits. The following version implements the `posix-ctime` with a optional parameter:

```
static PRIMITIVE posix_ctime(SCM seconds)
{
  long sec;

  sec = (seconds==UNBOUND) ? time(NULL)
                           : STk_integer_value_no_overflow(seconds);
  return STk_makestring(ctime((time_t *) &sec));
}
```

If the Scheme `posix-ctime` primitive is called with one parameter, it will be passed to the C function in the `seconds` parameter. If `posix-ctime` is called without parameter, `seconds` is set to the special value `UNBOUND`. So, the first test in this function consists to set a correct value to the variable `sec`; this value is either the current time, either the given integer, depending of the number of parameters given to `posix-ctime`.

Of course, the type of this new primitive must be changed to allow 0 or 1 parameter. This is done by changing the `tc_subr_0` in the previous `STk_add_new_primitive` by `tc_subr_0_or_1`.

The following types are available for C primitives:

- `tc_subr_0` for arity-0 primitives
- `tc_subr_1` for arity-1 primitives
- `tc_subr_2` for arity-2 primitives
- `tc_subr_3` for arity-3 primitives
- `tc_subr_0_or_1` for primitives which have 0 or 1 parameter (e.g. `read`). On the C side you have to declare a function which takes one `SCM` argument. This argument is set to the (evaluated) parameter if present, to `UNBOUND` otherwise.
- `tc_subr_1_or_2` for primitives which have 1 or 2 parameters (e.g. `write`). Here you have to declare a C function with two `SCM` parameters. The first one will contain the first Scheme argument and the second will contain the second argument value or `UNBOUND` if omitted.
- `tc_subr_2_or_3` for primitives which have 2 or 3 parameters (there's no primitive of this type in core interpreter). Of course, you'll have to declare a C function with three `SCM` parameters. Apart that, conventions are the same has before.
- `tc_lsubr` for primitives which have a variable number of arguments. Actuals arguments are collected in a list which is given as the first argument of the C primitive. The second argument of the C function is an integer counting the actual number of arguments given to the primitive. Hence, the signature of the C function which implement a `tc_lsubr` must be

  ```
  PRIMITIVE function(SCM arglist, int argcount);
  ```

  Note that all the Scheme arguments are evaluated during the construction of the list which is passed to the C function.
- `tc_fsubr` is similar to `tc_lsubr` except that arguments are not evaluated. On the C side, you have to declare a function with three `SCM` parameters: the list of (non evaluated) arguments, the current environment and the length of the arguments list. The signature of the C function which implement a `tc_fsubr` must be

  ```
  PRIMITIVE function(SCM arglist, SCM env, int argcount);
  ```

  See ?? more details about `tc_fsubr`.
- `tc_tkcommand` for primitives which follow the Tcl command argument passing style (i.e. *à la* `argc/argv`). This is this kind of procedure that will be used for to add new widgets in the STk interpreter. See ?? and [5] for more details.

To illustrate how to write a `tc_lsubr` primitive, let's have a look at the code, given below, of the function which implement the Scheme primitive `vector`:

```
PRIMITIVE STk_vector(SCM arglist, int argcount)
{
  int j;
  SCM z = STk_makevect(argcount, NULL);

  for (j = 0; j < argcount; j++, arglist=CDR(arglist)) {
    VECT(z)[j] = CAR(arglist);
  }
  return z;
}
```

This function receives the values passed to the `vector` primitives in the list arglist (the length of this list is stored in `argcount`). This function uses `STk_makevect` which returns a Scheme vector. Its first argument is the length of the vector and its second argument is the initial value of the vector's elements. Next section will show how to implement a primitive which evaluates itself its parameters (i.e. a `tc_fsubr` primitive.

2.3 Evaluating arguments

In some circumstances it could be useful to add new primitives which don't evaluate their arguments. This permits to add new control structures to the interpreter. To illustrate this, we will add two new primitives to the STk interpreter: `when` and `unless`. As explained in the preceding section, the C functions which will implement those control structures must be of type `tc_fsubr`. A `tc_fsubr` primitive, on the C side, is given three parameters when called:

1. a list of its (non evaluated) parameters,
2. the local environment when it was called (and in which evaluations should generally take place),
3. the length of the parameters list.

The C function can step through its parameter list using the C macros `CAR`, `CDR` and `NULLP` (which do the obvious work) and evaluates elements of this list as needed. Evaluation of an expression can be done with the `STk_eval` C function. `STk_eval` takes two parameters: the expression to evaluate and the environment in which evaluation takes place (the `NIL` variable, by convention, denotes the Indexglobal environment).

Note: a list of arguments is always a proper list. You don't need to test if it is well formed.

Hereafter is the code of the `when` primitive.

```
static PRIMITIVE when(SCM l, SCM env, int argcount)
{
  SCM res = UNDEFINED;

  if (argcount > 1) {
    if (STk_eval(CAR(l), env) != Ntruth) {
      for (l = CDR(l); !NULLP(l); l = CDR(l)) {
        res = STk_eval(CAR(l), env);
      }
    }
  }
  return res;
}
```

```
#include <stk.h>

static PRIMITIVE when(SCM l, SCM env, int argcount)
{
  SCM res = UNDEFINED;

  if (argcount > 1) {
    if (STk_eval(CAR(l), env) != Ntruth) {
      for (l = CDR(l); !NULLP(l); l = CDR(l)) {
        res = STk_eval(CAR(l), env);
      }
    }
  }
  return res;
}

static PRIMITIVE when(SCM l, SCM env, int argcount)
{
  SCM res = UNDEFINED;

  if (argcount > 1) {
    if (STk_eval(CAR(l), env) == Ntruth) {
      for (l = CDR(l); !NULLP(l); l = CDR(l)) {
        res = STk_eval(CAR(l), env);
      }
    }
  }
  return res;
}

PRIMITIVE STk_init_when_unless(void)
{
  add_new_primitive("when",         tc_fsubr, when);
  add_new_primitive("unless",       tc_fsubr, when);
  return UNDEFINED;
}
```

Figure 2: Source listing of file `when_unless.c`

Some points to note here:

- `UNDEFINED` is an interpreter *constant*. It serves to denote the notion of *"unspecified result"* of R^4RS.
- `Truth` and `Ntruth` are two global *constants* of the interpreter which denote respectively the `#t` and `#f` Scheme constants.

Figure 2.3 shows a complete implementation of `when` and `unless`.

2.4 Signaling errors

For now, only one function is provided to signal errors: `STk_err`. This function takes two parameters: a C string which constitutes the body of the message and a Scheme object (a `SCM` pointer) designating the *erroneous* object. If the second argument is `NIL`, it will not be printed. Execution of the function `STk_err` never returns. It provokes a jump at the start of the top-level loop. Hereafter, is a new implementation of the `when` function which uses `STk_err` when given an erroneous arguments list.

```
static PRIMITIVE when(SCM l, SCM env, int argcount)
{
  SCM res = UNDEFINED;

  switch (argcount) {
    case 0:  STk_err("when: no argument list given", NIL);
    case 1:  STk_err("when: null body", NIL);
    default:  /* Argument list is well formed.
               * Evaluate each expression of the body
               */
             if (STk_eval(CAR(l), env) != Ntruth)
               for (l = CDR(l); !NULLP(l); l = CDR(l))
                 res = STk_eval(CAR(l), env);
  }
  return res;
}
```

3 Variables

This section shows how you can access a Scheme variable within C code. It also shows how you can connect a Scheme and C variable such that modifying it in Scheme will modify the associated variable and *vice versa*.

3.1 Scheme Symbols and Variables

Defining a symbol:

Interning a symbol in the global table of symbols is done with the `STk_intern` C function. Since this function is often used, you can use the C macro `Intern` as a shortcut. The result of Intern is the `SCM` object which denotes the scheme symbol associated to the C string passed as parameter. For example, assigning the list

```
'(green orange red)
```

to the C variable `fire` can be done by

```
SCM fire =  Cons(Intern("green"),
                 Cons(Intern("orange"),
                      Cons(Intern("red"), NIL)));
```

Since this notation is difficult to read, some macros have been defined in `Src/stk.h` for building list. These macros are called `LISTx` where x is a number (comprised between 1 and 9) which represent the length of the list to create. Thus, the previous example could have been written as

```
SCM fire =  LIST3(Intern("green"), Intern("orange"), Intern("red"));
```

Reading a variable:

Reading a variable in Scheme corresponds in fact to look at the value associated to a symbol. The value associated to a symbol can be obtained with the `STk_get_symbol_value` C macro. This macro returns a `SCM` object which correspond to the value associated to the symbol whose name is equal to the parameter string. `STk_get_symbol_value` returns the special value `UNBOUND` is this symbols has no value in the global environment. The following piece of code

```
{
  SCM val = STk_get_symbol_value("foo");

  if (val == UNBOUND)
    STk_err("foo is undefined", NIL);
  else
    STk_display(val, UNBOUND);
}
```

displays the value of the `foo` symbol, or a message is `foo` is undefined in the global environment. Note the use of the `STk_display` function which implement the behavior of the Scheme `display` primitive. This call correspond to a call to `display` with only one parameter, since second parameter is set to `UNBOUND` (output is done on the standard output port in this case).

Setting a variable:
Setting a Scheme variable corresponds to associate a new value to a symbol. The value of a symbol can be set with the `STk_set_symbol_value` C macro. For example,

```
STk_set_symbol_value("bar", STk_makeinteger(3L));
```

sets the value of the `bar` symbol to the integer 3. Note that you can set a symbol in C without using a `define` form as it is necessary in Scheme.

3.2 Connecting Scheme and C variables

When building a specialized interpreter, it could be useful to have a variable you can access both in Scheme an in C. Modifying such a variable in C must modify the Scheme associated variable and, symmetrically, modifying it in Scheme must modify the corresponding C variable. One way to do this connection consists to create a special Scheme variable whose content is read/written by a special getter/setter. Definition of such a variable, is done by calling the function `STk_define_C_variable`. The C prototype for this function is

```
void STk_define_C_variable(char *var,
                           SCM  (*getter)(char *var),
                           void (*setter)(char *var, SCM value));
```

The following piece of code shows how we can connect the Scheme variable `*errno*` to the C variable `errno`:

```
static SCM get_errno(char *s)
{
  return STk_makeinteger((long) errno);
}

static void set_errno(char *s, SCM value)
{
  long n = STk_integer_value_no_overflow(value);

  if (n == LONG_MIN) Err("setting *errno*: bad integer", value);
  errno = n;
}

{
  ...
  STk_define_C_variable("*errno*", get_errno, set_errno);
  ...
}
```

After this call to `STk_define_C_variable`, reading (*resp.* writing) the value of the `*errno*` Scheme variable calls the `get_errno` (*resp.* `set_errno`) C function.

4 Calling Scheme from C

Sometimes, it could be necessary to execute some Scheme code from a C function. If the Scheme function you have to call is a primitive, it is preferred to call directly the C function which implement it. To know the name of the C function which implement a Scheme primitive, you'll have to look in the C file `primitivec` which contains the list of all the primitives of the core interpreter. If the Scheme code you want to execute is not a call to a primitive, it is generally easier to put your code in a C string and call the C function `STk_eval_C_string`. This function takes two parameters: the string to evaluate and the environment in which evaluation must take place. As for `STk_eval`, a `NIL` value for the environment denotes the global environment. Suppose, for instance, that you have already written in Scheme the `fact` procedure; evaluating the factorial of 10 can be done in C with:

```
STk_eval_string("(fact 10)", NIL);
```

This call returns a pointer on a Scheme object (a `SCM` pointer) containing the result of the evaluation. If an error occurs during evaluation. It is signaled to the user and the constant NULL is returned by `STk_eval_string`.

5 Adding new types

This sections discusses how to add a new type to the STk interpreter. Interested reader can find some new types definitions in the `Extensions` directory of STk. STklos, in particular, is written as an extended type whose definition is dynamically done as soon as *objects* are needed. Hash tables, processes and sockets are other examples of extended types.

5.1 Definition of a Scheme extended type

Defining a scheme extended type is a little bit more complicated than defining new primitives since it implies to take into account how this new type interact with the GC (Garbage Collector). Note that until now we have not discussed about GC problems since the interpreter is able to hide you it, as far as you don't define new types.

To illustrate the discussion, we will show how to add the *stack* type to the STk interpreter in this section. The complete code for this section can be found in appendix.

5.1.1 How the GC works

Before showing how to define a new Scheme type, it is important to understand how the GC works. First a certain number of cells are created[1]. When the interpreter needs a new cell, in the `cons` primitive for instance, it will take an unused cell in the pool of pre-allocated cells. If no more cell is available in this area, the GC is called. Its works is divided in two phases. First phase consists to mark all the cells which are currently in use. Finding the cells which are in used is done by marking recursively all the object which are accessible from

[1] by default 20 000; Use the -cells option of the interpreter to change this default

- the Scheme symbol table,
- the registers used by the program,
- the C stack,
- global variables of type `SCM`.

Marking phase is recursive; that means that if a variable denotes a list, all the elements of this list have to be marked , to avoid that the GC frees some of them. Of course, the recursive call for marking the component of a cell depends on the cell's type. This first phase is called the *marking phase*.

The second phase of the GC is called the *sweeping phase*. It is relatively simple: each allocated cells whose mark bit is unset is placed in the list of free cells, since nobody points anymore on it.

If no cells can be obtained when the sweeping phase terminates, the pool of pre-allocated cells will be extended by a new bank of cells.

5.1.2 The Extended type data structure

Defining a new Scheme type consists mainly to define a new `STk_extended_scheme_type` structure and fill in this fields. This structure is defined as:

```
The typedef struct {
  char *type_name;
  int  flags;
  void (*gc_mark_fct)(SCM x);
  void (*gc_sweep_fct)(SCM x);
  SCM  (*apply_fct)(SCM x, SCM args, SCM env);
  void (*display_fct)(SCM x, SCM port, int mode);
} STk_extended_scheme_type;
```

Each field of this structure is defined below

type_name:
is a string. It denotes the external name of the new type. The purpose of this field is mainly for debugging.

flags: is the union of binary constants. For now, only two constants are defined:

- `EXT_ISPROC` must be set if the new type is a procedure (i.e. if the Scheme procedure must answer #t when called with this object).
- `EXT_EVALPARAM` must be set if the new type must evaluates its parameters when used as a function.

gc_mark_fct:
is a pointer to the function which marks objects of the extended type. The code associated to this function is simple. It consists to call on each field whose type is `SCM` in the type associated data. This function is automatically called by the interpreter when it scans all the used cells in the GC marking phase. One example of `gc_mark_fct` is given below.

gc_sweep_fct:
is a pointer to the function which frees the resources allocated for representing the new type of object. This function is automatically called by the GC in the sweeping phase for each cell which is unused.

apply_function:
is a pointer to a function which is called when applying this object to a list of arguments. This function can only be called if the bit `EXT_ISPROC` is set. The arguments given to this function are evaluated if the `EXT_EVALPARAM` bit is set. Finally, the environment in which the call is done is passed as the third argument of the `apply_function`. It serves principally when the `EXT_EVALPARAM` bit is unset.

Set the `apply_function` to NULL to use the interpreter default apply function. The default function raises an error when called. You can use the default function when the new type you define is not a function.

display_fct:
is a pointer to a C function which displays objects of the new type. The display function has three parameters. The first parameter is the object to print. The second parameter is the port to which the object must be printed. Printing an object must be done with one of the following functions

- `int STk_getc(FILE *f);`
- `int STk_ungetc(int c, FILE *f);`
- `int STk_putc(int c, FILE *f);`
- `int STk_puts(char *s, FILE *f);`
- `int STk_eof(FILE *f);`

Those functions are extensions of their C equivalent: they are able to handle the STk string ports.

The third parameter of the `display_fct` is a mode constant which can take three different values.

- `DSP_MODE` is used when the object must be *displayed* in a human readable format (as with `display`);
- `WRT_MODE` is used when the object must be *written* in a machine readable format (as with `write`);
- `TK_MODE` is used when the object must be passed to a Tk command. This permits to customize the way a Scheme object is converted to a string when discussing with the Tk library.

Note: `display_fct` can be set to NULL. In this case the interpreter uses a default printing format. This default format print the name of the type (found in the `type_name`) followed by an hexadecimal address.

5.1.3 Registering the new type

Once a `STk_extended_scheme_type` structure is defined, the new type can be registered into the interpreter. Registering a new type is done by the `STk_add_new_type` function. The prototype of this function is given below

```
int STk_add_new_type(STk_extended_scheme_type *p)
```

The integer returned by this function is the (unique) key associated to the new type. This key is stored in each cell of the new type.

We have now enough material to define the `STk_extended_scheme_type` for the new type *stack*. This declaration can be done in the following way:

```
static void mark_stack(SCM p);
static void free_stack(SCM p);
static void display_stack(SCM s, SCM port, int mode);
static int tc_stack;

static STk_extended_scheme_type stack_type = {
  "stack",              /* name */
  0,                    /* is_procp */
  mark_stack,           /* gc_mark_fct */
  free_stack,           /* gc_sweep_fct */
  NULL,                 /* apply_fct */
  display_stack         /* display_fct */
};
```

This definition tells the interpreter thatthe new type is not a procedure (field `is_procp` is set to 0). Consequently, the `apply_fct` is set to NULL. Note that a display function is provided here. It permits to used a customized printing function.

5.1.4 New type instances creation

Creation of a new instance of the extended type necessitates the definition of a constructor function. This constructor obeys always the same framework. First you have to create a new cell with the `NEWCELL` macro. This macro has two parameters, a `SCM` object which will point the new cell and the type of the cell to create. The second argument is generally equal to the value returned by `STk_add_new_type`. Once the cell is created, we have generally to (dynamically) allocate a C structure which contains the informations which are necessary to implement the new type. Dynamic allocation can be done with the function `STk_must_malloc`. The area returned by `STk_must_malloc` must be stored in the `data` field of the new cell. This field can be accessed with the `EXTDATA` macro.

Let's go back to the stack example. We can now define a new primitive function to make a new stack. Provided that the global variable `tc_stack` already contains the value returned by `STk_add_new_type`, we can write

```
#define STACKP(x)       (TYPEP(x, tc_stack))
#define NSTACKP(x)      (NTYPEP(x, tc_stack))
#define STACK(x)        ((Stack *) EXTDATA(x))

typedef struct {
  int len;
  SCM values;
} Stack;

static PRIMITIVE make_stack(void)
{
  SCM z;

  NEWCELL(z, tc_stack);
  EXTDATA(z)       = STk_must_malloc(sizeof(Stack));
  STACK(z)->len    = 0;
  STACK(z)->values = NIL;
  return z;
}
```

Here, the `Stack` structure is used to represent a stack. This structure contains two fields: `len` and `values`. Since the latter field is a `SCM` object, it must be recursively marked when a stack is marked. We can now define the utility function necessary for the GC:

```
static void mark_stack(SCM p)
{
  STk_gc_mark(STACK(p)->values);
}

static void free_stack(SCM p)
{
  free(EXTDATA(p));
}
```

To terminate with this example, we give below the code of the primitive `stack_push!`. Other primitive are built in the same fashion and will not be described here. A complete listing of the stack implementation is given in appendix.

```
static PRIMITIVE stack_push(SCM s, SCM val)
{
  Stack *sp;

  if (NSTACKP(s)) STk_err("stack-push: bad stack", s);

  sp         =  STACK(s);
  sp->len   += 1;
  sp->values = Cons(val, sp->values);

  return UNDEFINED;
}
```

5.2 Definition of a C extended type

The STk interpreter permits to handle C pointers as first class objects. [for eg: find an example to explain how it works: gdbm?]

5.3 About memory: Common pitfalls

6 Loading an extension

STk support dynamic loading for several architectures/systems. The way to provide dynamic loadable modules is different from one system to another and you will have to adapt what is said here to the conventions used by your system, architecture or compiler. Static loading can be used for systems which doesn' support dynamic loading (such as Ultrix) or for which the interpreter doesn't support yet dynamic loading.

Note: STk) also supports the DLD Gnu package for dynamic loading. DlD is a library package of C functions that performs "dynamic link editing". Since the time to load dynamically a module with this package is rather long, it is preferred to avoid to use it. However, this package is the only way to provide dynamic loading on Linux systems which don't support the ELF format (versions 1.0 to 1.2). Since the ELF format is becoming the new standard for Linux, this package will be no more necessary in the future.

The last version of the DLD package can be found at several places:

- `ftp-swiss.ai.mit.edu:pub/scm`
- `prep.ai.mit.edu:/pub/gnu/jacal`

- `ftp.cs.indiana.edu:/pub/scheme-repository/imp/SCM-support`

We suppose here that we want to include the `posix` module defined in section2.1 into the STk interpreter.

Dynamic Loading:

If the system running STk supports dynamic loading (and if the interpreter has been compiled with dynamic loading support), you compile your source file file to make a *shared object* file. On SunOs 4.1, for instance, this can be done by compiling the module with the *pic compilation option* (pic stands for *position independent code*). Once compilation is done, you can pre-load your file with the line

```
ld -assert pure-text -o time.so time.o
```

This will produce a file name `posix.so` which can be loaded with the `load` Scheme primitive procedure. The `load` primitive recognizes that this file is a shared object and calls a function whose name the concatenation of the string `STk_init_` and the base name of file loaded. Thus, loading the file `posix.so` impies the call of a pimitive whose name is `STk_init_posix`.

Look at `Src/Extensions` directory to see some examples of shared object construction.

Note: when the STk is built, the `Makefile` in the `Src/Extensions` is customized for your system/compiler. A simple way to determine the options you have to use for compiling your program consist to run the `make` command on one of the file present in this directory. For instance, issuing the following command

```
make -n posix.so
```

on a Linux box using the DLD package will output the following lines:

```
gcc  -g -DSTk_CODE -DUSE_DLD -DLINUX   -DHAVE_UNISTD_H=1 \
        -DHAVE_SIGACTION=1 -I../Tk -I../Tcl -I../Src -I../Mp \
        -I/usr/X11R6/include  -c posix.c -o posix.o
ld -r -o posix.so posix.o
```

Static loading:

A C module which define a new type can also be statically loded in the interepreter. To load your module, you have to modify the `Src/Makefile` or `Snow/Makefile`. Once yo have added your extension object in the `USER_OBJ` variable , you must modify the file `Src/userinit.c` to add you initialization (and eventually cleanup) code. The call to your initialization function must be done in the `STk_user_init` C function. Once this is done, you can run the `make` command again to build the extended interpreter.

7 Adding new Tk widgets

7.1 Widget compilation

Adding a new Tk widget to the STk interpreter is generally a simple *hack*. Most of the time, extension widgets written for Tcl/Tk can be added to the STk interpreter without modifying the source code of the widget. However, there is no unique method to add a widget to the Tcl interpreter; consequently, what is given below is a set of hints to widget integration rather than a *always working recipe*. To illustrate this section, we will see how we can add the *fscale* widget (a floating-point scale widget available on the Tcl/Tk repository in the `tkFScale-?.?.tar.gz` file)[2].

[2] This widget is now integrated in the standard Tk4.0; it is provided as an extension widget with Tk3.6.

Generally, the code of a Tcl/Tk extension widget can be divided in two parts: the code which implement widget's behavior and the extension initialization code. Extension initialization code, in Tcl/Tk, must be placed in the procedure `Tcl_AppInit` which is located in the file `tkAppInit.c`. If the extension package adds a lot of widgets, it generally defines a function to do all the initializations. On the other hand, if the extension only defines a single widget, the extension code generally consists to call the C function `Tcl_CreateCommand` for each new widget defined in the extension. `Tcl_CreateCommand` is the Tcl standard way to add a new command. This function also exists in the STk interpreter; it creates a new Tk command object [3]. The prototype of this function is:

```
void Tcl_CreateCommand(Tcl_Interp *interp,
                       char *cmdName,
                       Tcl_CmdProc *proc,
                       ClientData clientData,
                       Tcl_CmdDeleteProc *deleteProc));
```

For STk,

- the `interp` is always the global variable `Stk_main_interp`
- `cmdName` is the name of the widget in the Scheme world.
- `proc` is the name of the C function which implement the *Tk command*
- `clientData` are informations which are associated to the widget code. For a new widget, `clientData` can generally set to the result of

  ```
  Tk_MainWindow(Stk_main_interp);
  ```

- `deleteProc` is a function which is called when the widget is destroyed. You generally don't need to change the value of this parameter (which is often set to `NULL`).

The usual way to integrate this initialization code in a Tcl interpreter consists to patch the `tkAppInit.c` file to add the call to the initialization (or the `Tcl_CreateCommand`) function. To add an extension written for Tcl/Tk to STk, all that is needed consists to adapt the initialization code for STk. For example, the `fscale` widget initialization code adds the following call in the body of the `Tcl_AppInit` function:

```
Tcl_CreateCommand(interp, "fscale", Tk_FScaleCmd,
                  (ClientData) main,
                  (void (*)()) NULL);
```

For STk, this call can be written

```
Tcl_CreateCommand(STk_main_interp, "fscale", Tk_FScaleCmd,
                  (ClientData) Tk_MainWindow(STk_main_interp),
                  (void (*)()) NULL);
```

This call must executed before trying to create a new `fscale` widget.

7.2 Widget linking

The cleaner way to add a new widget to STk consists to define a special C module for this widget. Defining the widget in a C module allows us to make the new widget dynamically loadable. The code for making the `fscale` widget dynamically loadable could be:

```
/* Contents of the file fscale.c */
#ifndef USE_TK
#define USE_TK
#endif

#include <stk.h>
/*
 *include the widget source code. Ugly but this avoid to have two
 * source files to link
 */
#include "tkFScale.c"

PRIMITIVE STk_init_fscale(void)
{
  Tcl_CreateCommand(STk_main_interp,
                    "fscale",
                    Tk_FScaleCmd,
                    (ClientData) Tk_MainWindow(STk_main_interp),
                    (void (*)()) NULL);
}
```

8 Extending the interpreter with C++

[for eg : Identify the problems]

9 Embedding the STk interpreter

[for eg: This parts need some work in the interpreter]

Appendix

Hereafter is the complete code for the stack type discussed in 5.1

```
/*
 *
 * s t a c k  . c -- Implementation of the extended type Stack
 *
 */

#include <stk.h>

static void mark_stack(SCM p);
static void free_stack(SCM p);
static void display_stack(SCM s, SCM port, int mode);
static int tc_stack;

static STk_extended_scheme_type stack_type = {
  "stack",/* name */
  0,/* is_procp */
  mark_stack,/* gc_mark_fct */
  free_stack,/* gc_sweep_fct */
  NULL,/* apply_fct */
  display_stack /* display_fct */
};

#define STACKP(x) (TYPEP(x, tc_stack))
#define NSTACKP(x) (NTYPEP(x, tc_stack))
#define STACK(x) ((Stack *) EXTDATA(x))

typedef struct {
  int len;
  SCM values;
} Stack;

static void mark_stack(SCM p)
{
  STk_gc_mark(STACK(p)->values);
}

static void free_stack(SCM p)
{
  free(EXTDATA(p));
}

static void display_stack(SCM s, SCM port, int mode)
{
  char buffer[100];
  if (mode == DSP_MODE) {
    /* A verbose display */
    if (STACK(s)->len) {
      sprintf(buffer, "Stack length = %d\nValues = ", STACK(s)->len);
      Puts(buffer, FILEPTR(port));
      STk_display(STACK(s)->values, port);
    }
    else
      Puts("Stack is empty", FILEPTR(port));
  }
  else { /* WRT_MODE or TK_MODE */
    sprintf(buffer, "#<stack (length=%d) %ld>", STACK(s)->len, s);
    Puts(buffer, FILEPTR(port));
  }
}

static PRIMITIVE make_stack(void)
{
  SCM z;
```

```
  NEWCELL(z, tc_stack);
  EXTDATA(z)        = STk_must_malloc(sizeof(Stack));
  STACK(z)->len     = 0;
  STACK(z)->values = NIL;
  return z;
}

static PRIMITIVE stackp(SCM s)
{
  return STACKP(s)? Truth: Ntruth;
}

static PRIMITIVE stack_push(SCM s, SCM val)
{
  Stack *sp;

  if (NSTACKP(s)) STk_err("stack-push: bad stack", s);

  sp          =  STACK(s);
  sp->len    += 1;
  sp->values = Cons(val, sp->values);

  return UNDEFINED;
}

static PRIMITIVE stack_pop(SCM s)
{
  Stack *sp;
  SCM res;

  if (NSTACKP(s)) STk_err("stack-pop: bad stack", s);

  sp =  STACK(s);

  if (sp->len == 0) STk_err("stack-pop: empty stack", s);
  res          = CAR(sp->values);
  sp->len     -= 1;
  sp->values  = CDR(sp->values);

  return res;
}

static PRIMITIVE stack_emptyp(SCM s)
{
  if (NSTACKP(s)) STk_err("stack-empty?: bad stack", s);
  return (STACK(s)->len) ? Truth: Ntruth;
}


PRIMITIVE STk_init_stack(void)
{
  /* Register the new type */
  tc_stack = STk_add_new_type(&stack_type);

  /* Declare new primitives */
  STk_add_new_primitive("make-stack",     tc_subr_0,      make_stack);
  STk_add_new_primitive("stack?", tc_subr_1,      stackp);
  STk_add_new_primitive("stack-push!",    tc_subr_2,      stack_push);
  STk_add_new_primitive("stack-pop",      tc_subr_1,      stack_pop);
  STk_add_new_primitive("stack-empty?",   tc_subr_1,      stack_emptyp);

  return UNDEFINED;
}
```

References

[1] POSIX Committee. *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990.

[2] Erick Gallesio. Embedding a scheme interpreter in the tk toolkit. In Lawrence A. Rowe, editor, *First Tcl/Tk Workshop, Berkeley*, pages 103–109, June 1993.

[3] Erick Gallesio. STk reference manual (version 2.2). Technical Report RT 95-31, I3S CNRS / Université de Nice - Sophia Antipolis, juillet 1995.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[5] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, isbn: 0-201-63337-x edition, 1994.

Index