Sizzle

Embedding Manual
Version 0.0.30

Martin Grabmueller

Copyright © 1999, 2000 Martin Grabmueller

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

1 Introduction

Sizzle is an interpreter for a programming language. It is designed to be used as an embedded extension language and as a scripting language for a wide variety of purposes.

This manual documents how to master the embedding process needed to make use of Sizzle's features in a C application. For more information on Sizzle, on Scheme programming and a reference of the variables and procedures available in Sizzle see section "Introduction" in *The Sizzle Reference Manual*.

Chapter 2 deals with embedding, containing a tutorial which shows how to use Sizzle to parse the initialization file for a little example program.

In Chapter 3, the C API for embedding is documented.

The following Chapter 4 deals with mechanisms and concepts for extending the Sizzle interpreter for special purposes by adding data types and primitive procedures.

2 Embedding Sizzle

The Sizzle interpreter is implemented in the library 'libsizzle'. This library is installed when Sizzle is installed and actually is used by the Sizzle interpreter 'sizzle' for all its work. 'libsizzle' can be embedded in other programs very easily, and this chapter will show how to do it.

2.1 Compiling and Linking

When compiling against the Sizzle library, you have to tell the compiler where to find the Sizzle include files, and the linker, where to find the library itself. When you installed Sizzle in the standard location (with $prefix set to '/usr/local/lib'), the files are likely to be found, but embedding Sizzle should be possible when installed in another location too. That is the reason the script 'sizzle-config' was included in the distribution. This script gets installed in the same location as the 'sizzle' interpreter, and can be used to obtain information about the Sizzle installation.

'sizzle-config' understands three commands, which must be given on the command line. Either command causes the script to print out a line of options, suitable for inclusion in the compiler or linker command.

compile
: Print the -I compiler option for finding the include files.

link
: Print the options for linking against the Sizzle library (for example, -L/usr/local/lib -R/usr/local/lib -lsizzle -lm -ldl)

static-link
: Print the options for linking statically against the Sizzle library (for example, /usr/local/lib/libsizzle.a -lm -ldl)

2.2 Parsing Initialization files

One design goal for Sizzle was to make it easy to use the interpreter for parsing initialization files. Using a complete programming language for init files has the advantage that a lot intelligence for setting up a program can be put into these init files. One popular example is the Emacs editor. This chapter will explain in tutorial style which steps are necessary to use Sizzle for this purpose.

The four steps in the following sub-section describe in detail what to do if you want to include Sizzle into your application for the purpose of parsing init files. In the 'examples' directory of the source distribution of Sizzle are included the files 'example.c' and 'startup.scm', which illustrate the following steps.

2.2.1 Initializing the library

Before you can use the 'libsizzle' library, you have to initialize it. Two steps are necessary to do that. First, you have to tell it the address of the top of the C stack. This is done by declaring a local dummy variable and passing its address to the function zzz_set_top_of_stack().

```
int
main (int argc, char * argv[])
{
  zzz_scm_t dummy;

  /* Announce top of stack to Sizzle library, so conservative marking
     works.  */
  zzz_set_top_of_stack (&dummy);
```

Then the library initialization is performed by a call to zzz_initialize().

```
/* Initialize the library.  */
zzz_initialize ();
```

The last step is not mandatory, but very useful. You can pass the command line options given to your program to the interpreter to make it available to your Scheme code.

```
/* Make command line options available to Scheme code.  The zero'th
   argument is handled specially to make pre-processing of the
   arguments easier (see src/sizzle.c for details).  */
zzz_set_arguments (argc - 1, argv[0], argv + 1);
```

2.2.2 Declaring variables

When the library is correctly initialized, you can bind variables in your C code to Scheme variables. Thus you can directly modify variables in your C program from Scheme code.

Three different data types are currently supported: integers, strings and boolean values. Whenever you bind the address of a C variable to a Scheme variable you have to tell the interpreter of which data type the variable is. The interpreter will then type-check assignments and signal errors whenever a value of a wrong type is stored into a variable. Additionaly, it is considered an error to store a string which is too long into a string variable.

The example program declares five variables of different types.

```
/* Bind some variables of different types.  The last argument is a
   flag whether the variable is allowed to get changed by Scheme
   code.  */
zzz_bind_int_variable ("binary-port", &binary_port, 0);
zzz_bind_int_variable ("command-port", &command_port, 0);
zzz_bind_bool_variable ("verbose", &verbose, 0);
zzz_bind_string_variable ("hostname", hostname, sizeof (hostname), 0);
zzz_bind_string_variable ("download-area", download_area,
                          sizeof (download_area), 0);
zzz_bind_scm_variable ("kill-lines", &kill_lines, 0);
```

The first argument in each of these calls is the name which will be given to the Scheme variable, the second argument is the address of the C variable and the last is a flag whether the variable should be read-only on the Scheme level. When declaring string variables, you have to pass the maximal length of the variable also, so range-checking can be performed.

2.2.3 Loading the init file

A call to the function `zzz_evaluate_file()` finally loads the init file, which may contain any valid Scheme code. The Scheme code has access to all variables defined above. For details about the init file, see the file '`startup.scm`' in the '`examples`' directory.

```
/* Evaluate startup file.  */
if ((return_val = zzz_evaluate_file (zzz_toplevel_env, "startup.scm")) != 0)
  fprintf (stderr, "example: error while reading init file\n");
```

The return value of `zzz_evaluate_file()` is zero when all went right and a non-zero value otherwise.

The variable `kill-lines` was bound to a generic variable of type `zzz_scm_t`. That means that any Scheme value can be stored in the variable and that no type checking is performed as for the other variables which have explicitly been bound to variable of type int, boolean or string. When using such variables, you must be careful when interpreting the values stored there. The following example shows how to safely handle a variable which was supposed to hold a list of strings.

```
/* Interpretation of bound variables of type zzz_scm_t must be done
   before the library is finalized, otherwise dangling pointers will
   make your life hard.  */

/* Iterate over list.  */
while (cons_p (kill_lines))
  {
    /* Make sure we are dealing with strings.  */
    if (string_p (car (kill_lines)))
      {
        printf ("Kill line: %s\n", string_val (car (kill_lines)));
      }
    kill_lines = cdr (kill_lines);
  }
```

2.2.4 Shutting down the library

After parsing the init file, you will normally want to free all memory used by the interpreter. This is done by a call to `zzz_finalize()`.

```
/* Free all memory used by the interpreter.  */
zzz_finalize ();
```

Note that all Scheme values, even those which might be stored into C variables using `zzz_bind_scm_variable()`, are no longer valid when you have shut down the interpreter.

That's it! Not to hard, methinks.

3 Embedding API

This section documents the C API for embedding Sizzle into C applications.

The calling conventions of most API functions follow the same concept. The return value of those functions is an error code, values computed by the functions are normally returned in reference parameters. This is a little bit unconvenient when calling these functions, because a lot of temporary variables for holding the intermediate results must be declared, but provides a very powerful interface for error handling, since not only the fact that an error occured can be signalled, but also additional information about the error condition can be returned in the result variable.

That is the reason why those return codes and parameters are also used for implementing exceptions. Whenever a call to an API function does not return `RESULT_SUCCESS`, the error is immediately propagated up the call chain. So not only errors are passed up, but also exceptions. The exception objects passed up together with the exception code can be examined and particular exceptions can be caught using this technique.

3.1 Initializing and Finalization

When using the Sizzle library, you habe the choice of using one of two methods. See Section 2.2 [Parsing initialization files], page 2, where the first is shown in detail. It consists of calling the functions `zzz_set_top_of_stack()`, `zzz_initialize()` and `zzz_finalize()`; the other method is used by the Sizzle interpreter '`sizzle`' and uses only the call `zzz_run()`. The former is more easily to incorporate into existing programs whereas the latter is much simpler and less error-prone.

`void` **zzz_set_top_of_stack** (`zzz_scm_t *` *cell*) Function

Tell the Sizzle library that the pointer *cell* points to a location on the C stack which marks the end of the stack as far as conservative garbage collection is concerned. Use this function before any other call to the Sizzle library, or use the `zzz_run()` function.

`void` **zzz_initialize** (`void`) Function

Initialize the library.

`void` **zzz_finalize** (`void`) Function

Shut down the library and free all memory allocated by the interpreter.

`void` **zzz_set_arguments** (`int` *argc*, `char *` *argv0*, `char *` *argv*[]) Function

Set the command line arguments which will be visible to running Scheme code. *argc* is the count of elements in the vector *argv*, *argv0* is the program name and *argv* is a vector of the remaining command line arguments.

`int` **zzz_run** (`int (*` *main*)(`int argc, char * argv`[]), `int` *argc*, `char *` *argv*[]) Function

Intitialize the Sizzle library and call the function *main*, passing *argc* and *argv* as arguments. This is the recommended function for using the Sizzle library and is used by the '`sizzle`' program itself. Returns the exit code of *main*. You should not call any function from '`libsizzle`' before calling this function and you should not call any after calling it. `zzz_run()` does all necessary initialization and finalization.

3.2 Scheme Variable Handling

The function in this section all modify objects in the Scheme name space. They either create or modify Scheme variables, constants or functions, but they can also be used to query those objects.

All functions below create their bindings in the outermost scope, even below the toplevel environments. The bindings are thus equivalent in scope to the builtin variables and procedures (in fact, builtin bindings are created using these functions).

result_t **zzz_get_variable** (zzz_scm_t *sym*, zzz_scm_t * *result*) Function

Return the value of the variable *sym* in the global environment. Returned is the value in the location pointed to by *result* and RESULT_SUCCESS if no error occurs, otherwise an error code is returned and an error object is stored in the location pointed to by *result*.

result_t **zzz_set_variable** (zzz_scm_t *sym*, zzz_scm_t value, zzz_scm_t * *result*) Function

Set the variable *sym* in the global environment to the value *value*. RESULT_SUCCESS is returned if no error occurs, otherwise an error code is returned and an error object is stored in the location pointed to by *result*.

zzz_scm_t **zzz_define_variable** (char * *name*, zzz_scm_t *value*) Function

Define the variable *name*, which will be bound in the toplevel environment to *value*. Returns the symbol created for the variable.

zzz_scm_t **zzz_define_constant** (char * *name*, zzz_scm_t *value*) Function

Define the read-only variable *name*, which will be bound in the toplevel environment to *value*. Returns the symbol created for the variable. The difference between zzz_define_constant() and zzz_define_variable() is that the variables defined with the former can not be modified by set! operations. Also, variables defined using zzz_define_constant() are more efficient because references to these constant variables are replaced by their values during runtime.

zzz_scm_t **zzz_define_function** (char * *name*, result_t (* *func*)(), int *param_format*) Function

Define a function called *name* which refers to the builtin function *func* and takes parameters as specified by *param_format*.

zzz_scm_t **zzz_define_form** (char * *name*, zzz_prim_t *func*, int preprocessing) Function

Define the special form *name*, which will be calling the C function *func*. Returns the symbol created for the function. Note that when functions for handling special forms are called, they are passed the entire form, including the special form name as the first element of the parameter list. If *preprocessing* is true, the function *func* will not evaluate the form, but only error check and replace it with an immediate form for faster execution.

3.3 Hash table handling

Hash tables are represented as vectors which hold association lists. When storing a key/value pair, or looking up the value for a given pair, a hash value is calculated for the key and used as an index into the vector. Then the association list found at the index is scanned for the key.

unsigned long **zzz_hash_function** (zzz_scm_t *object*) Function

Return a hash value for the Scheme object *object*.

zzz_scm_t **zzz_hashq_ref** (zzz_scm_t *tab*, zzz_scm_t *key*, zzz_scm_t *def*) Function

zzz_scm_t **zzz_hashv_ref** (zzz_scm_t *tab*, zzz_scm_t *key*, zzz_scm_t *def*) Function

zzz_scm_t **zzz_hash_ref** (zzz_scm_t *tab*, zzz_scm_t *key*, zzz_scm_t *def*) Function

Fetch the element associated with *key* from the hash table *tab*. If an error occurs or the *key* is not found, *def* is returned. zzz_hashq_ref uses eq? as the equality predicate for searching for *key*, zzz_hashv_ref uses eqv? and zzz_hash_ref uses *equal?*.

`zzz_scm_t` **zzz_hashq_set** (`zzz_scm_t` *tab*, `zzz_scm_t` *key*, `zzz_scm_t` *value*) Function

`zzz_scm_t` **zzz_hashv_set** (`zzz_scm_t` *tab*, `zzz_scm_t` *key*, `zzz_scm_t` *value*) Function

`zzz_scm_t` **zzz_hash_set** (`zzz_scm_t` *tab*, `zzz_scm_t` *key*, `zzz_scm_t` *value*) Function

Add the association *key* => *value* to the hash table *tab*. If an association with *key* already exists, it is overwritten with *value*. If an error occurs or the *key* is not found, *NULL* is returned, if everything went right, *value* is returned. `zzz_hashq_ref` uses `eq?` as the equality predicate for searching for *key*, `zzz_hashv_ref` uses `eqv?` and `zzz_hash_ref` uses *equal?*.

3.4 C Variable Handling

Another method for interfacing Scheme and C code is declare variables in your C source and then bind the locations of these variables to Scheme variable names. The advantage is that you can then reference the values stored into the variables by simply using the variables like normal C variables. Variables created with these functions are totally transparent to the Scheme code, there is not even a method how the Scheme code can find out whether a given variable is bound to a C variable or not.

`result_t` **zzz_protect_global** (`zzz_scm_t *` *scm*) Function

Protect the variable pointed to by *scm* globally from being garbage collected. You should call this functions whenever you create a variable of type `zzz_scm_t`, but do not wish to bind that variable to a Scheme variable using `zzz_bind_scm_variable()`. Returns `RESULT_SUCCESS` on success and an error code otherwise.

`void` **zzz_bind_int_variable** (`char *` *name*, `int *` *address*, `int` *read_only*) Function

Create a toplevel Scheme variable named *name* of type integer which is bound to the C variable pointed to by *address*. The variable will be read-only if *read_only* is true.

`void` **zzz_bind_bool_variable** (`char *` *name*, `int *` *address*, `int` *read_only*) Function

Create a toplevel Scheme variable named *name* of type boolean which is bound to the C variable pointed to by *address*. The variable will be read-only if *read_only* is true.

`void` **zzz_bind_string_variable** (`char *` *name*, `char *` *address*, `int` *len*, `int` *read_only*) Function

Create a toplevel Scheme variable named *name* of type string which is bound to the C variable pointed to by *address*. The variable will be read-only if *read_only* is true.

`void` **zzz_bind_scm_variable** (`char *` *name*, `zzz_scm_t *` *address*, `int` *read_only*) Function

Create a toplevel Scheme variable named *name* of type zzz_scm_t which is bound to the C variable pointed to by *address*. The variable will be read-only if *read_only* is true. This functions automatically takes care of protecting the passed location from garbage collection, thus variables bound using this functions are safe.

3.5 Evaluation Functions

Use one of these functions if you like to evaluate Scheme code. The functions which take an *environment* as a parameter evaluate the given expression, string or file in that environment, that means that all bindings which may be created are effectively created in the given environment. You can pass the variable *zzz_toplevel_env* for the *env* parameter to these functions if you do not have any special requirements.

`int` **zzz_evaluate_file** (`zzz_scm_t` *env*, `const char *` *filename*) Function

Evaluate all Scheme expressions from the file called *filename* in the environment *env*. Returns 0 if no error occurs, -1 if the file was not found, -2 if an error occured when evaluating the file or another exit code if the file contained a call to `exit`. All expressions in the file are evaluated in the environment *env*. Evaluation is aborted as soon as an error is encountered or evaluation is aborted using `exit`.

`int` **zzz_evaluate_string** (`zzz_scm_t` *env*, `const char *` *str*) Function

Evaluate all Scheme expressions from the string *str* in the environment *env*. Returns 0 if no error occurs, -2 if an evaluation error is encountered or another exit code if the string contained a call to `exit`. All expressions in the string are evaluated in the environment *env*. Evaluation is aborted as soon as an error is encountered or evaluation is aborted using `exit`.

`int` **zzz_read_eval_print** (`void`) Function

Execute Sizzle's read-eval-print loop. Expressions will be read from standard input, evaluated and the results will be written to standard output. Error messages will be sent to standard error. Returns 0 if terminated normally or another exitcode if the user enters the `exit` command with an exit code.

`result_t` **zzz_apply** (`zzz_scm_t` *env*, `zzz_scm_t` *proc*, `zzz_scm_t` *param_list*, `zzz_scm_t *` *result*) Function

Apply the procedure *proc* (which may be a primitive procedure or a closure) to the parameter list *param_list*. Evaluation takes place in the environment *env*. The result is returned in the location pointed to by *result*. On success, `RESULT_SUCCESS` is returned, otherwise an error code and an error object is stored in *result*.

`result_t` **zzz_evaluate** (`zzz_scm_t` *env*, `zzz_scm_t` *expr*, `zzz_scm_t *` *result*) Function

Evaluate the Scheme expression *expr* in the environment *env*. The result is returned in the location pointed to by *result*. On success, `RESULT_SUCCESS` is returned, otherwise an error code and an error object is stored in *result*.

3.6 Object constructors

When creating Scheme values, the functions defined in this section should be used. They guarantee correct interfacing to the memory management system and the garbage collector.

`zzz_scm_t` **zzz_cons** (`zzz_scm_t` *car*, `zzz_scm_t` *cdr*) Function

Creates a cons cell with car and cdr initialized to the values of *car* and *cdr*, respectively.

`zzz_scm_t` **zzz_make_list** (`zzz_scm_t` *first*, `...`) Function

Create a Scheme list of all parameters. *first* is the first element of the resulting list, followed by the optional parameters. The parameter list must be terminated with a NULL pointer.

`zzz_scm_t` **zzz_make_n_list** (`int` *n*, `zzz_scm_t` *first*, `...`) Function

Create a Scheme list of exactly *n* parameters. *first* is the first element of the resulting list, followed by the optional parameters.

`zzz_scm_t` **zzz_make_fixnum** (`long` *number*) Function

Create a fixnum object with integer value *number*. If *number* does not fit into the range of fixnums, its high bits are silently truncated. The fixnum range is between -268435456 and 268435455, inclusive.

You should not use this function unless you are sure that the given value fits into a fixnum. Use `zzz_make_integer()` instead.

`zzz_scm_t` **zzz_make_integer** (`long` *value*) Function
: Create an integer object with value *value*. A fixnum is returned if *value* fits into the range defined for fixnums, otherwise a long object is returned which holds all 32 bits of information from *value*.

`zzz_scm_t` **zzz_make_string** (`const char *` *s*, `int` *len*) Function
: Create a string of length *len* and initialize with the data pointed to by *s*. If *len* is less than zero, *s* is considered a null-terminated string and the length is calculated by using `strlen()`.

`zzz_scm_t` **zzz_make_nstring** (`int` *len*) Function
: Create a string of length *len*. The contents of the returned string is unspecified.

`zzz_scm_t` **zzz_make_ro_string** (`const char *` *s*, `int` *len*) Function
: Create a constant string of length *len* and initialize with the data pointed to by *s*. If *len* is less than zero, *s* is considered a null-terminated string and the length is calculated by using `strlen()`. The returned string is read-only and cannot be modified with Scheme primitives like `string-set!`.

`zzz_scm_t` **zzz_make_ro_nstring** (`int` *len*) Function
: Create a constant string of length *len*. The contents of the returned string is unspecified. The returned string is read-only and cannot be modified with Scheme primitives like `string-set!`.

`zzz_scm_t` **zzz_make_bool** (`long` *b*) Function
: Create a boolean object. If *b* is zero, the false Scheme object `#f` is returned, otherwise the canonic true Scheme object `#t` is returned.

`zzz_scm_t` **zzz_make_char** (`long` *b*) Function
: Create a character object with character value *b*. Only the lower 8 bits of *b* are actually used.

`zzz_scm_t` **zzz_make_float** (`double` *d*) Function
: Create a real number object initialized with the value of *d*.

`zzz_scm_t` **zzz_make_symbol** (`const char *` *s*, `int` *len*) Function
: Create a Scheme symbol with the string representation *s* of length *len*. If *len* is less than zero, `strlen()` is used to determine the string length. This function will return the same object for all strings with the same length and the same character contents.

`zzz_scm_t` **zzz_make_func** (`zzz_prim_t` *func*, `const char *` *name*, `int` *form*, `int` *param_format*) Function
: Create a primitive function object. *func* is the address of the C function that handles the primitive function, *name* is the Scheme name under which the function will be available, *form* should be 0 for primitive functions and non-zero of syntactic forms and *param_format* is a description of the parameter format the function expects. Use one of the `TAGGED_INFO_ARG_*_*_*` constants for this parameters.

`zzz_scm_t` **zzz_make_lambda** (`zzz_scm_t` *expr*) Function
: Create a lambda object. *expr* must be a list where the car is the formal parameter list and the cdr must be the procedure body.

`zzz_scm_t` **zzz_make_closure** (`zzz_scm_t` *proc*, `zzz_scm_t` *env*) Function
: Create a closure object which closes the procedure *proc* under the environment *env*.

`zzz_scm_t` **zzz_make_error** (`zzz_scm_t` *msg*, `zzz_scm_t` *reference*) Function
: Create an error object. *msg* is the error message, *reference* is a list containing the file name, line number and column number where the error occurred; or NULL if that information is not available.

zzz_scm_t **zzz_make_exception** (zzz_scm_t *exception*, zzz_scm_t *msg*) Function

Create an exception object. *exception* is the exception tag (any atom) and *msg* is the exception message which may hold additional information about the exception.

zzz_scm_t **zzz_make_environment** (unsigned *size*, zzz_scm_t *static_link*, int *final*, int *read_only*) Function

Create an environment object. *static_link* is the link to the including lexical environment in which variable binding will be searched. *size* denotes the size the hash table for the new environment will have; it should be a prime number.

When *final* is true, variable lookups will not descend the static chain further than to the created environment.

When *read_only* is true, variables defined in environments further down the static link can only be read, but not modified.

zzz_scm_t **zzz_make_location** (zzz_scm_t * *location*, int *read_only*) Function

Create a location object. A location object is an indirect object which refers to a Scheme object variable address. *location* points to the location to redirect to and *read_only* specified whether this location may be written to.

zzz_scm_t **zzz_make_lloc** (zzz_scm_t *symbol*, int *env_ofs*, unsigned *hash_val*) Function

Create a lloc object. A lloc object is substituted for all local variables which have been looked up once and which are not special variables. They are used to cache the location of variables and to speed up references to local variables. *symbol* is the variable name this lloc stands for, *env_ofs* is the number of environments to traverse down the static link and *hash_val* is the hash index into the environment where we have to search for *symbol*.

zzz_scm_t **zzz_make_gloc** (zzz_scm_t *name*, zzz_scm_t * *address*) Function

Create a gloc object. A gloc is similar to a lloc, but only applies to global variables not in an environment, but in the symbol table. *name* is the variable name this gloc stands for, and *address* is the location where the value of the variable can be found.

zzz_scm_t **zzz_make_constant** (zzz_scm_t *value*) Function

Create a constant object with value *value*. A constant object is special, since it value may be substituted whenever it is referenced, and therefore references are faster than variable references. Also variables containing constant objects can not be modified.

zzz_scm_t **zzz_make_keyword** (const char * *s*, int *len*) Function

Create a Scheme keyword object with the string representation *s* of length *len*. If *len* is less than zero, `strlen()` is used to determine the string length. This function will return the same object for all strings with the same length and the same character contents.

zzz_scm_t **zzz_make_vector** (int *len*) Function

Create a vector object which holds *len* elements. The vector elements are initialized to the empty list `'()`.

zzz_scm_t **zzz_make_ro_vector** (int *len*) Function

Create a constant vector object which holds *len* elements. The vector elements are initialized to the empty list `'()`. The returned vector is read-only and can not be modified using Scheme primitives like `vector-set!`.

zzz_scm_t **zzz_make_continuation** (void) Function

Make a continuation object. Note that the returned object is not initialized, you have to capture the current continuation explicitly using `zzz_capture_continuation()`.

zzz_scm_t **zzz_create_tagged_cell** (unsigned long *tag*, const void * *data0*, const void * *data1*, const void * *data2*); Function

This is the generic object creation function for tagged types. You have to pass a type tag (maybe combined with one or more of the type_info_* constants) in *tag*, and additional information in the parameters *data0*, *data1* and *data2*. The format of the data parameters depends on what the internal constructor function for the tagged type expects.

zzz_scm_t **zzz_make_values** (zzz_scm_t *values*) Function

Creates a multiple value object with the values in the list *values*.

zzz_scm_t **zzz_make_regexp** (const char * *str*, int *len*, int *flags*) Function

Creates a regular expression object which matches the string *str* of length *len*. *flags* are used for compiling the regular expression and may be any of the compile time flags specified for Posix regular expressions.

zzz_scm_t **zzz_make_promise** (zzz_scm_t *env*, zzz_scm_t *expression*) Function

Creates a promise object. The new object is closed over the environment *env*, that means that the promise will be evaluated in that environment once it is forced. *expr* is the expression to be delayed.

zzz_scm_t **zzz_make_macro** (zzz_scm_t *code*) Function

Creates a macro object with macro code *code*. The car of *code* must be the formal parameter list, the cdr must be the macro body.

zzz_scm_t **zzz_make_syntax** (zzz_scm_t *rules*) Function

Creates a syantx object with syntax rules *rules*. The rules must have the same syntax as provided to a call to the Scheme form syntax-rules. body.

zzz_scm_t **zzz_make_fport** (FILE * *file*, int *read*, int *write*) Function

Creates a standard IO port for the file referenced by *file*. *read* and *write* tell the input-output mode of the resulting port object.

zzz_scm_t **zzz_make_fdport** (int *fd*, int *read*, int *write*) Function

Creates a file descriptor port for the file descriptor *fd*. *read* and *write* tell the input-output mode of the resulting port object.

zzz_scm_t **zzz_make_sport** (int *start_len*) Function

Creates a string port with a preallocated character buffer of size *start_len*.

zzz_scm_t **zzz_make_sport_str** (const char * *str*) Function

Creates a string port with the initial contents of the null-terminated string *str*. The file pointer is initially set to zero.

zzz_scm_t **zzz_make_sport_str_n** (const char * *str*, unsigned *len*) Function

Creates a string port with the initial contents of the string *str* with length *len*. The file pointer is initially set to zero.

zzz_scm_t **zzz_make_sport_str_n_no_copy** (char * *str*, unsigned *len*) Function

Creates a string port with the initial contents of the string *str* with length *len*. The file pointer is initially set to zero. This function does not create a copy of the passed string, so you have to make sure that the contents remains valid as long as the string port does exists. Use it for cnstant strings etc.

`zzz_scm_t` **zzz_make_*TAG*vector** (`int` *len*) Function

Create a homogenous numeric vector object which holds *len* elements. The vector elements are initialized to zero.

TAG in the constructor name may be replaced by `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` or `f64`, depending on the needed datatype. So there are actually ten of these constructor functions.

`zzz_scm_t` **zzz_make_ro_*TAG*vector** (`int` *len*) Function

Create a constant homogenous numeric vector object which holds *len* elements. The vector elements are initialized to zero. The returned vector is read-only and can not be modified using Scheme primitives like *TAG*`vector-set!`.

TAG in the constructor name may be replaced by `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` or `f64`, depending on the needed datatype. So there are actually ten of these constructor functions.

`zzz_scm_t` **zzz_make_pointer** (`void *` *address*) Function

Create a pointer object which referencing *address*.

`zzz_scm_t` **zzz_make_pointer_n** (`void *` *address*, `unsigned` *n*) Function

Create a pointer object which referencing a memory area starting at *address* and being *n* bytes long.

`zzz_scm_t` **zzz_copy_list** (`zzz_scm_t` *list*) Function

Make a copy of the list *list*. Note that *list* must be a proper list. This function only copies the spine of the list, that means, that the list elements are shared, and only the cons cells construcing the list are newly allocated. If you need to make a deep copy of a list, use `zzz_copy_tree()` instead.

`zzz_scm_t` **zzz_copy_tree** (`zzz_scm_t` *tree*) Function

Make a deep copy of the object *tree*. That means, that vectors and pairs are cloned recursively and all other objects are simply copied to the returned object.

`zzz_scm_t` **zzz_make_not_available_exception** (`zzz_scm_t` *args*) Function

Create an exception object with the tag `not-available`, and the additional exception data *args*. This exception is thrown whenever a primitive is called which is not supported under the currently running version of Sizzle.

`zzz_scm_t` **zzz_make_port** (`unsigned` *port_type*, `void *` *info*) Function

Create a port of the specified port type and install *info* as the port's info field.

3.7 Type predicates

These type predicates are safe in the sense that you do not risk a segmentation fault when applying any of them to an arbitraty value of type `zzz_scm_t`. They simply return 0 if the condition they test for is not satisfied. Use them before using any of the accessor functions (see Section 3.8 [Accessor functions], page 15).

Note that the return values of these predicates are boolean values in the C sense: zero means false and any other value means true.

null_p (*c*) Macro

`null_p(c)` returns a true value if *c* is the empty list.

fixnum_p (*c*) Macro

`fixnum_p(c)` returns a true value if *c* is a fixnum object.

imm_p (*c*) Macro

`imm_p(c)` returns a true value if *c* is an immediate value, Immediate values are all values which are encoded into a pointer of type `zzz_scm_t`, and which do not occupy any cells on the heap. The empty list, all fixnum values and immediate form objects are immediate values.

immediate_p (*c*) Macro

`immediate_p(c)` returns a true value if *c* is an immediate form. Immediate forms are substituted for form applications to speed up the evaluation of expressions.

cons_p (*c*) Macro

`cons_p(c)` returns a true value if *c* is a cons pair.

list_p (*c*) Macro

Returns a true value if *c* is a list. A list is either a cons pair or the empty list.

integer_p (*c*) Macro

`integer_p(c)` returns a true value if *c* is an integer object. Both fixnum and long objects are integer objects.

procedure_p (*c*) Macro

`procedure_p(c)` returns a true value if *c* is a procedure object. Procedure objects are primitive procedures, syntax forms and lambda expressions.

number_p (*c*) Macro

`number_p(c)` returns a true value if *c* is a number object. Fixnum, long and real objects are numbers.

tagged_p (*c*) Macro

`tagged_p(c)` returns a true value if *c* is a tagged object. This macro is true for all values which are neither immediate in the sense of `immediate_p()` nor cons cells in the sense of `cons_p()`.

string_p (*c*) Macro

`string_p(c)` returns returns a true value if *c* is a string object.

rostring_p (*c*) Macro

`rostring_p` returns returns a true value if *c* is a constant string object.

bool_p (*c*) Macro

`bool_p(c)` returns a true value if *c* is a boolean object.

char_p (*c*) Macro

`char_p(c)` returns a true value if *c* is a character object.

float_p (*c*) Macro

`float_p(c)` returns a true value if *c* is a real number object.

symbol_p (*c*) Macro

`symbol_p(c)` returns a true value if *c* is a symbol.

func_p (*c*) Macro

`func_p(c)` returns a true value if *c* is a primitive procedure.

form_p (*c*) Macro

`form_p(c)` returns a true value if *c* is a syntactic form.

lambda_p (*c*) Macro

`lambda_p(c)` returns a true value if *c* is a lambda closure.

error_p (*c*) Macro
`error_p(c)` returns true if *c* is an error object.

except_p (*c*) Macro
`except_p(c)` returns true if *c* is an exception object.

long_p (*c*) Macro
`long_p(c)` returns true if *c* is a long integer object.

env_p (*c*) Macro
`env_p(c)` returns true if *c* is an environment.

location_p (*c*) Macro
`location_p(c)` returns true if *c* is a location object.

rolocation_p (*c*) Macro
`rolocation_p(c)` returns true if *c* is a constant location object.

lloc_p (*c*) Macro
`lloc_p(c)` returns true if *c* is an lloc object.

gloc_p (*c*) Macro
`gloc_p(c)` returns true if *c* is a gloc object.

constant_p (*c*) Macro
`constant_p(c)` returns true if *c* is a constant value object.

keyword_p (*c*) Macro
`keyword_p(c)` returns true if *c* is a keyword.

vector_p (*c*) Macro
`vector_p(c)` returns a true value if *c* is a vector object.

rovector_p (*c*) Macro
`rovector_p(c)` returns a true value if *c* is a constant vector object.

values_p (*c*) Macro
`values_p(c)` returns true if *c* is a multiple value object.

int_var_p (*c*) Macro
ro_int_var_p (*c*) Macro
`int_var_p(c)` returns true if *c* is an integer variable wrapper object, `ro_int_var_p(c)` returns true if *c* is a read-only integer variable wrapper object.

bool_var_p (*c*) Macro
ro_bool_var_p (*c*) Macro
`bool_var_p(c)` returns true if *c* is a boolean variable wrapper object, `ro_bool_var_p(c)` returns true if *c* is a read-only boolean variable wrapper object.

str_var_p (*c*) Macro
ro_str_var_p (*c*) Macro
`str_var_p(c)` returns true if *c* is a string variable wrapper object, `ro_str_var_p(c)` returns true if *c* is a read-only string variable wrapper object.

regexp_p (*c*) Macro
`regexp_p(c)` returns true if *c* is a regular expression object.

promise_p (*c*) Macro
`promise_p(c)` returns true if *c* is a promise object.

macro_p (*c*) Macro
: `macro_p(c)` returns true if *c* is a macro code object.

syntax_p (*c*) Macro
: `syntax_p(c)` returns true if *c* is a syntax object.

port_p (*c*) Macro
: `port_p(c)` returns true if *c* is a port object.

fport_p (*c*) Macro
: `fport_p(c)` returns true if *c* is a standard IO port object.

fdport_p (*c*) Macro
: `fdport_p(c)` returns true if *c* is a file desccriptor port object.

sport_p (*c*) Macro
: `sport_p(c)` returns true if *c* is a string port object.

*TAG***vector_p** (*c*) Macro
: *TAG*`vector_p(c)` returns a true value if *c* is a homogenous numeric vector object of the type indicated by *TAG*.

TAG in the predicate name may be replaced by `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` or `f64`, depending on the needed datatype. So there are actually ten of these predicate macros.

ro_*TAG***vector_p** (*c*) Macro
: `ro_`*TAG*`vector_p(c)` returns a true value if *c* is a homogenous numeric vector object of the type indicated by *TAG*.

TAG in the predicate name may be replaced by `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` or `f64`, depending on the needed datatype. So there are actually ten of these predicate macros.

3.8 Accessor functions

The macros and functions in this section are used to query the properties of Scheme objects. Before applying them to any object, you have to check whether the object is of the correct type, because the accessor macros and functions will not check for validity.

car (*c*) Macro
cdr (*c*) Macro
: Return the car or cdr of *c*, which must be a pair.

set_car (*c, value*) Macro
set_cdr (*c, value*) Macro
: Set the car or cdr of *c* to *value*.

car_addr (*c*) Macro
cdr_addr (*c*) Macro
: Return the address of the car or cdr of *c*.

tagged_type (*c*) Macro
: Return the type of the tagged object *c*.

tagged_info (*c*) Macro
: Return the tagged info field of the tagged object *c*. The result will be one of the `tagged_info_*` constants.

tagged_data0 (*c*) Macro
tagged_data1 (*c*) Macro
tagged_data2 (*c*) Macro
Returns the values of the data slots of the tagged object *c*. The results are of type `void *` and must be casted before being used.

immediate_val (*c*) Macro
Returns the value of the immediate form *c*.

fixnum_val (*c*) Macro
Returns the value of the fixnum *c*.

integer_val (*c*) Macro
Returns the integer value of *c*, which must be either a fixnum or a long object.

string_val (*c*) Macro
Returns a pointer to the string contents of *c*.

string_len (*c*) Macro
Returns the length of the string *c*.

bool_val (*c*) Macro
Returns the value of the boolean object *c*.

char_val (*c*) Macro
Returns the value of the character object *c*.

float_val (*c*) Macro
Returns the value of the real number object *c*.

symbol_name (*c*) Macro
Returns the name of the symbol *c*.

symbol_value (*c*) Macro
Returns the global value of the symbol *c*.

symbol_name_addr (*c*) Macro
Returns the address of the name field of the symbol *c*.

symbol_value_addr (*c*) Macro
Returns the address of the value field of the symbol *c*.

set_symbol_name (*c*, *s*) Macro
Set the name of the symbol *c* to *s*.

set_symbol_value (*c*, *s*) Macro
Set the global value of the symbol *c* to *s*.

func_ptr (*c*) Macro
Returns the address of the C function which handles the primitive procedure or syntactic form *c*.

func_name (*c*) Macro
Returns the name of the primitive procedure or syntactic form *c*.

func_form_p (*c*) Macro
Returns true if the primitive procedure or syntactic form *c* is actually a syntactic form.

func_param (*c*) Macro
Returns the parameter specification of the primitive procedure or syntactic form *c*.

lambda_name (*c*) Macro
Returns the name of the lambda expression *c*, or `NULL` if the closure is anonymous.

lambda_args (*c*) Macro
Returns the formal argument list of the closure *c*.

lambda_body (*c*) Macro
Returns the closure body of *c*

lambda_env (*c*) Macro
Returns the environment the lambda expression *c* is closed over.

lambda_file (*c*) Macro
The file name in which the closure was defined, or NULL if not available.

lambda_line (*c*) Macro
Returns the line number in the file where the closure was defined, or NULL if not available.

long_val (*c*) Macro
Returns the value of the long integer object *c*.

env_val (*c*) Macro
Returns the environment vector of the environment *c*.

location_address (*c*) Macro
Returns the address the location object *c* refers to.

lloc_val (*c*) Macro
Returns the lloc list containing access data for memoized lloc objects.

gloc_name (*c*) Macro
Returns the name of the gloc *c*.

gloc_val (*c*) Macro
Returns the address the gloc *c* refers to.

constant_val (*c*) Macro
Returns the value of the constant value object *c*.

keyword_name (*c*) Macro
Returns the name of the keyword *c*.

vector_val (*c*) Macro
Returns a pointer to the array of vector elements of vector *c*, which are all of type `zzz_scm_t`.

vector_len (*c*) Macro
Returns the length of the vector *c*.

zzz_vector_put (*vector, index, value*) Macro
Store the object *value* at index *index* into *vector*. Make sure that *index* is valid before calling this macro.

zzz_vector_get (*vector, index*) Macro
Fetch the object at index *index* from *vector*. Make sure that *index* is valid before calling this macro.

values_val (*c*) Macro
Returns the value list of the muliple value object *c*.

int_var_addr (*c*) Macro
Returns the address of the integer variable the integer variable wrapper *c* refers to.

bool_var_addr (*c*) Macro
Returns the address of the boolean variable the boolean variable wrapper *c* refers to.

str_var_val (*c*) Macro
Returns the address of the string variable the string variable wrapper *c* refers to.

str_var_len (*c*) Macro
Returns the length of the string variable the string variable wrapper *c* refers to.

str_var_size (*c*) Macro
Returns the maximum size of the string variable the string variable wrapper *c* refers to.

regexp_valid (*c*) Macro
Returns true if the regular expression object *c* was compiled successfully.

regexp_string (*c*) Macro
Returns the string pattern which was compiled into the regular expression object *c* as a Scheme string object.

regexp_regex (*c*) Macro
Returns the regular expression structure of type `regex_t`, into which the regular expression was compiled.

promise_env (*c*) Macro
Returns the environment in which the promise *c* will be evaluated.

promise_result (*c*) Macro
Returns the result of the promise *c*. Only a valid field if the promise was already forced.

promise_expr (*c*) Macro
Returns the expression which was delayed in the promise *c*.

promise_thawed (*c*) Macro
Returns a true value if the promise *c* was already forced, false otherwise.

macro_code (*c*) Macro
Returns the macro code of the macro object *c*.

syntax_rules (*c*) Macro
Returns the syntax rules of the syntax object *c*.

port_ptype (*c*) Macro
Returns the port type of the port object *c*.

port_open_p (*c*) Macro
Returns true if the port object *c* is open.

port_write_p (*c*) Macro
Returns true if the port object *c* is for output.

port_read_p (*c*) Macro
Returns true if the port object *c* is for input.

port_line_number (*c*) Macro
Returns the current line number of the port object *c*.

port_col_number (*c*) Macro
Returns the current column number of the port object *c*.

port_saved_col (*c*) Macro
Returns the last column number of the port object *c*. This is remembered to restore column numbers after a `port_ungetc()` operation.

fport_file (*c*) Macro
Returns the file pointer of the standard IO file port *c*.

fdport_fd (*c*) Macro
Returns the file descriptor of the file descriptor port *c*.

fdport_has_unget (*c*) Macro
Returns true if the file descriptor port *c* has pushed back data.

fdport_unget (*c*) Macro
Returns the pushed back character of the file descriptor port *c*.

sport_val (*c*) Macro
Returns a pointer to the character buffer of the string port *c*.

sport_len (*c*) Macro
Returns the current length of valid data in the character buffer of the string port *c*.

sport_size (*c*) Macro
Returns the allocated size of the character buffer of the string port *c*.

sport_pos (*c*) Macro
Returns the current file pointer of the string port *c*.

*TAG***vector_val** (*c*) Macro
Returns a pointer to the array of vector elements of the homogenous numeric vector *c*, which are all of the type indicated by *TAG*.

TAG in the constructor name may be replaced by `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` or `f64`, depending on the needed datatype. So there are actually ten of these accessor macros.

*TAG***vector_len** (*c*) Macro
Returns the length of the homogenous numeric vector *c*.

TAG in the constructor name may be replaced by `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` or `f64`, depending on the needed datatype. So there are actually ten of these accessor macros.

3.9 Port Functions

`unsigned` **zzz_define_port_type** (`port_getc_t` *port_getc*, `port_ungetc_t` *port_ungetc*, `port_putc_t` *port_putc*, `port_puts_t` *port_puts*, `port_flush_t` *port_flush*, `port_close_t` *port_close*, `port_seek_t` *port_seek*, `port_tell_t` *port_tell*, `port_char_ready_p_t` *port_char_ready_p*, `port_free_t` *port_free*, `port_mark_t` *port_mark*) Function
Define a new port type. Must be given functions for all supported port operations.

result_t **zzz_port_puts** (zzz_scm_t *port*, char * *buf*, int *len*, zzz_scm_t * *result*); Function

Put the string *buf* of length *len* onto the port *port*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_putc** (int *ch*, zzz_scm_t *port*, zzz_scm_t * *result*) Function

Put the character *ch* to port *port*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_print** (zzz_scm_t *port*, zzz_scm_t *cell*, zzz_print_state_t *state*, zzz_scm_t * *result*) Function

Print the Scheme object *cell* to the port *port*. Formatting is specified via the passed *state* structure. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_getc** (zzz_scm_t *port*, int * *ch*, zzz_scm_t * *result*) Function

Read a character from the port *port* and return it in the location pointed to by *ch*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_ungetc** (int *ch*, zzz_scm_t *port*, zzz_scm_t * *result*) Function

Return the character *ch* back to the stream of characters from port *port*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_close** (zzz_scm_t *port*, zzz_scm_t * *result*) Function

Close the port *port*. After using this function, no more data can be written to or read from *port*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_seek** (zzz_scm_t *port*, long *ofs*, int *whence*, long * *res_ofs*, zzz_scm_t * *result*) Function

Move the file pointer of port *port* to position *ofs*. Interpretation of *ofs* depends on the parameter *whence*, which is the same as in th C library function fseek(). The new offset after repositioning is returned in the location pointed to by *res_ofs*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_tell** (zzz_scm_t *port*, long * *res_ofs*, zzz_scm_t * *result*) Function

Return the position of *port*'s file pointer in the location pointed to by *res_ofs*. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_flush** (zzz_scm_t *port*, zzz_scm_t * *result*); Function

Flushes all pending output of the port *port* which has not yet been written to the underlying file. Returns RESULT_SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

result_t **zzz_port_char_ready_p** (zzz_scm_t *port*, int * *ready*, zzz_scm_t * *result*); Function

Checks whether data is available for reading on the input port *port*. A flag indicating the availability of data is returned in the location pointed to by *ready*. Returns RESULT_

SUCCESS on success or an error code and an error object in the location pointed to by *result* if an error occurs.

3.10 Defining Scheme Types

unsigned long **zzz_define_tagged_type** (void * (* *constructor_func*) (zzz_tagged_t tagged, const void * data0, const void * data1, const void * data2), void (* *print_func*)(zzz_scm_t port, zzz_tagged_t tagged, zzz_print_state_t state), void (* *free_func*) (zzz_tagged_t tagged), void (* *mark_func*) (zzz_tagged_t tagged), int (* *equal_func*) (zzz_tagged_t tagged0, zzz_tagged_t tagged1)) — Function

This function defines a new Scheme type. The functions *constructor_func*, *print_func*, *free_func*, *mark_func* and *equal_func* are stored internally and are used for all objects of the new type. `zzz_define_tagged_type()` returns a type tag for the new type.

The following two functions can be given as arguments to `zzz_define_tagged_type()`, if the data fields of the tagged type contain variables of type `zzz_scm_t`.

void * **zzz_simple_constructor** (zzz_tagged_t *tagged*, const void * *data0*, const void * *data1*, const void * *data2*) — Function

Constructor for simple types which store complete value state in the data fields of the tagged cell. Returns its argument *tagged*.

void **zzz_simple_mark** (zzz_tagged_t *tagged*); — Function

Mark function for types which hold zzz_scm_t values in their data slots which should be protected from being GC'ed.

void **zzz_mark_cell** (zzz_scm_t *cell*); — Function

Use this functions from the marking function of your tagged type if you have to mark a variable of type `zzz_scm_t`.

3.11 Misc C API Functions

The following functions do not seem to fit into any of the previous sections, but they may come in handy from time to time.

void **zzz_garbage_collect** (void) — Function

Force immediate garbage collection.

zzz_eq (*first*, *second*) — Macro

Returns 1 if *first* and *second* are equal in the sense of `eq?`. Returns 0 otherwise.

This macro can be undefined in the file '`node.h`', which will cause a function with the same functionality, but more error checking, to be compiled in.

int **zzz_eqv** (zzz_scm_t *first*, zzz_scm_t *second*) — Function

Returns 1 if *first* and *second* are equal in the sense of `eqv?`. Returns 0 otherwise.

int **zzz_equal** (zzz_scm_t *first*, zzz_scm_t *second*) — Function

Returns 1 if *first* and *second* are equal in the sense of `equal?`. Returns 0 otherwise.

int **zzz_list_length** (zzz_scm_t *list*); — Function

Returns the length of list *list*. Returns -1 if *list* is not a proper list and returns -2 if *list* contains circular references. A proper list is a list terminated by the empty list `'()` and a circular list is a list which points to one of its own elements somewhere.

4 Memory Management

In this section, we will see how Sizzle represents Scheme objects internally and how the memory needed for storing these objects is managed.

4.1 Cell Representation

The Sizzle interpreter makes a difference between four types of Scheme objects: immediate values, fixnums, cons cells and tagged cells. These types are stored in memory differently, and we will look at the representation of them in detail in the following section.

Most functions use the opaque data type `zzz_scm_t`, which is defined as a pointer to `struct cell`. A value of type `zzz_scm_t` can represent four different kinds of data. They are differentiated by their two least significant bits.

If these bits are zero, it is a pointer to a cons cell which in turn holds two values of type `zzz_scm_t`. Cons cells are used to build the normal Scheme lists from. Environments, procedures, hash tables etc. are all build using lists, so objects of this type are quite common.

Should the two least significant bits be 1, it represents an 29-bit, two-complement integer value, a so-called fixnum; in order to get the real value, the value must be right-shifted by three bits.

The third kind, encoded by a 2 in the two least significant bits is a so-called tagged cell where the first word of the object pointed to is the a combination of the *tagged data type* (a 16 bit unsigned int) and additional 13 information bits. The usage of the info bits depends on the particular object type. Function objects, for example, encode the expected parameter formet in these bits; and read-only strings and vectors have a bit set in the info field. The next three words of the tagged cell object (as you can calculate, a tagged object is at least 4 words long) are data words and the values stored there depend on the data type. Some types store a pointer to additional storage there, others use the three words to hold values larger than one word (floating point values, for example, store their 2-word value in the last two words of the object).

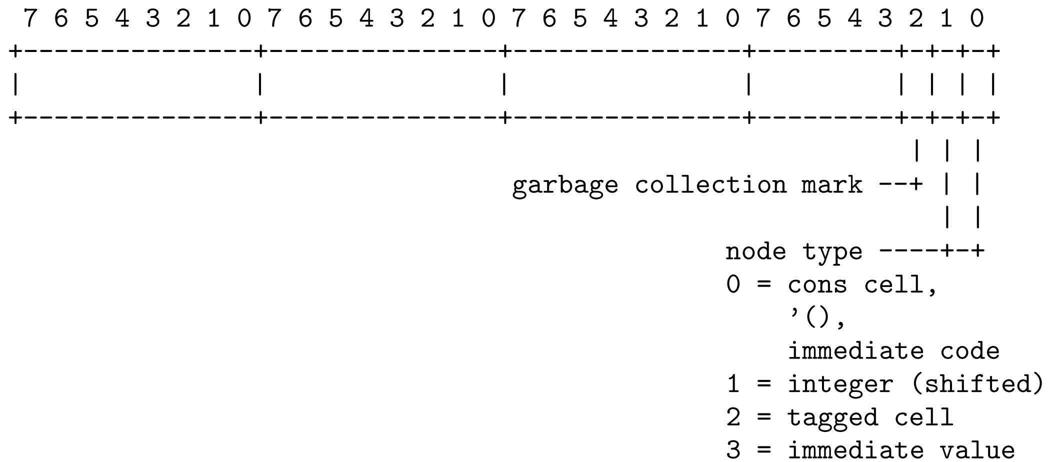
A value of 3 in the two least significant bits of the *car* of a cell means that the cell represents a so-called immediate value. Immediate values do not point to any heap-allocated object, but stand for themselves. The immediate forms, into which most syntactic forms are transformed on evaluation, are examples of this type of object.

Bit two is used to implement a marking garbage collector. Cells which have been seen during the scan phase of collection have this bit set in the *car*.

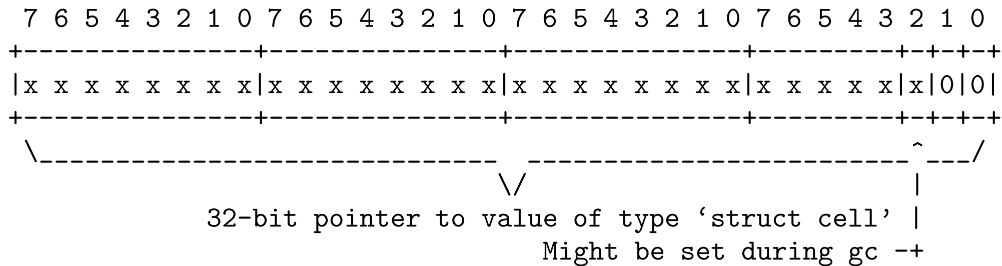
Cons cells and tagged cells are allocated on two different heaps, because they have a different size.

Pointer Layout

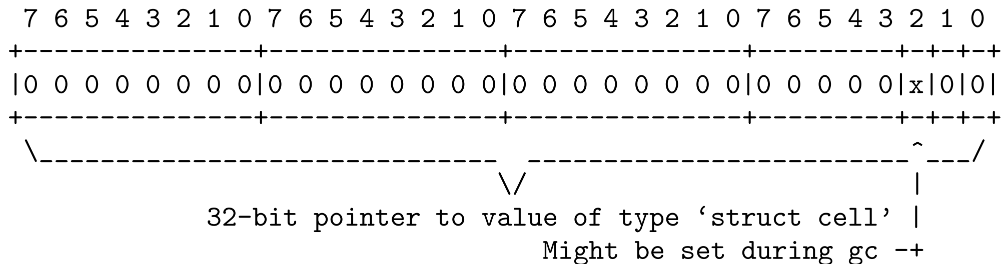
General layout of a variable of type `zzz_scm_t`:



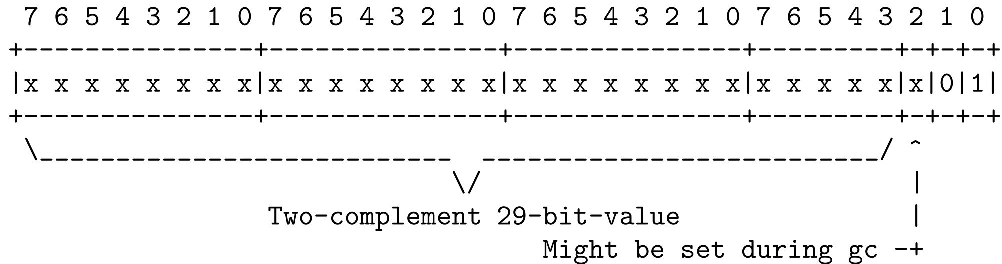
Cons cell pointers have zeros in their least significant bits. They can be used as pointers to structures of type `struct cell` without further modification. Note: when dealing with values of this type while garbage collection is running is dangerous, because you have to mask out bit #2 before dereferencing the pointer.



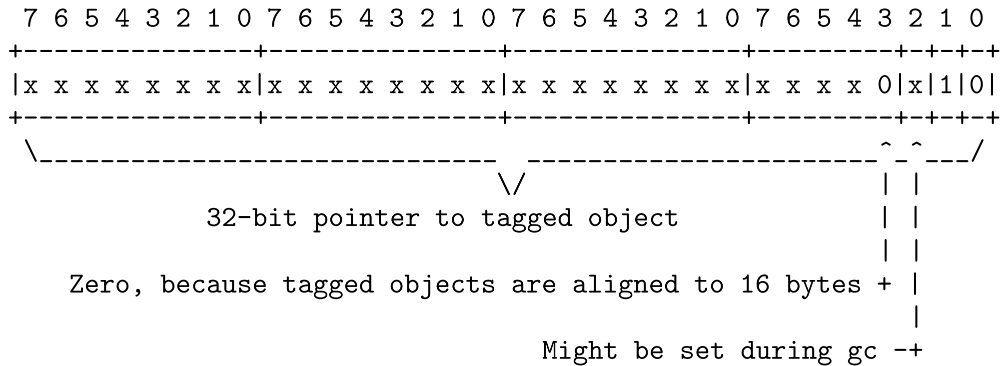
The null pointer (which is the representation of the empty list) is a word of only zero bits. Only during garbage collection the third bit may be set and must be masked off.



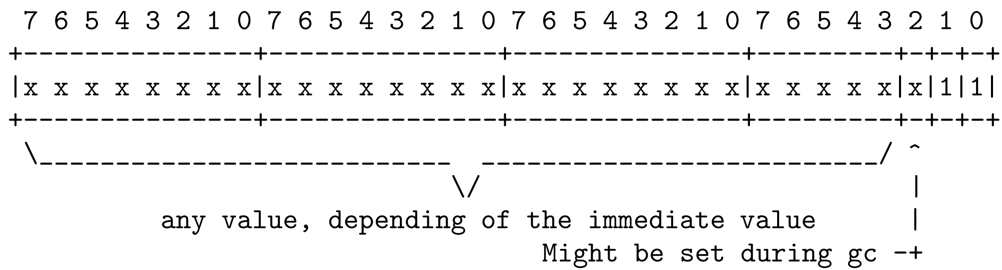
Integer values, aka fixnums, have the following layout. The value of any fixnum value can be obtained by arithmetically right-shifting the value by three bits.



Tagged cells hold the value 2 in their two least significant bits and a pointer to a tagged cell structure can be obtained by masking the lower 4 bits off.



Immediate codes have their least significant two bits set and their value encoded in the upper 29 bits.

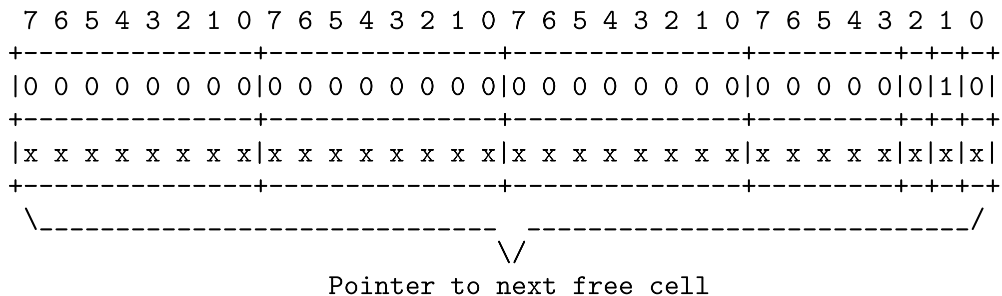


Heap Cell Layout

When looking at any cell on one of the heaps, the values defined in the following paragraphs are valid.

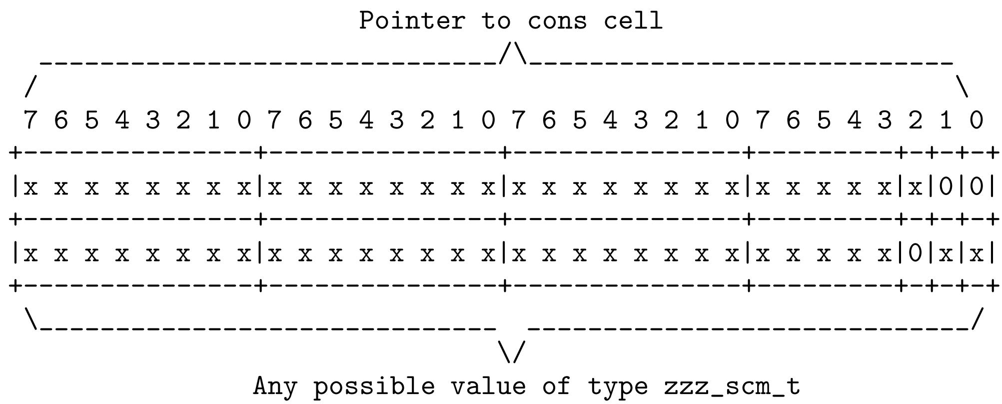
On the cons heap, any cell can be either free or in use.

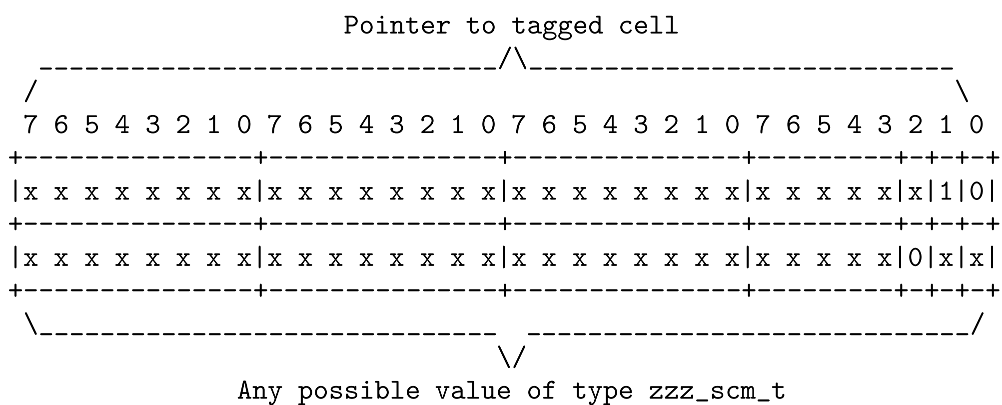
Free cells carry a pointer to the next entry in the free list in the cdr and the value '2' in the car.

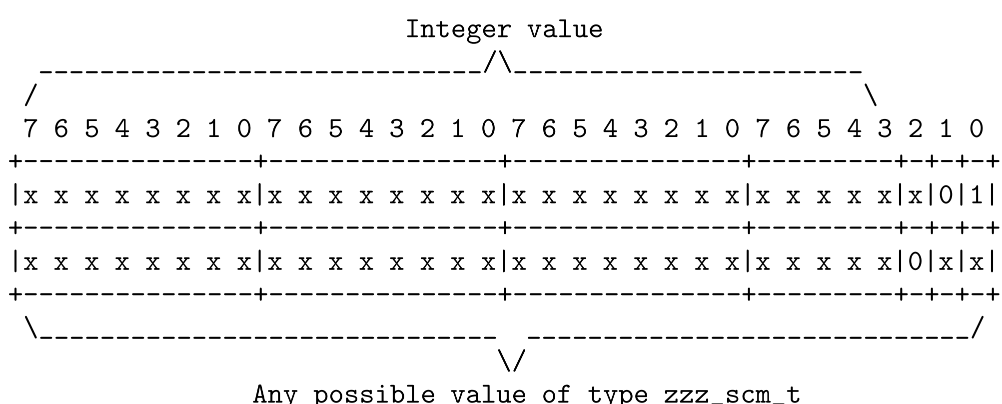


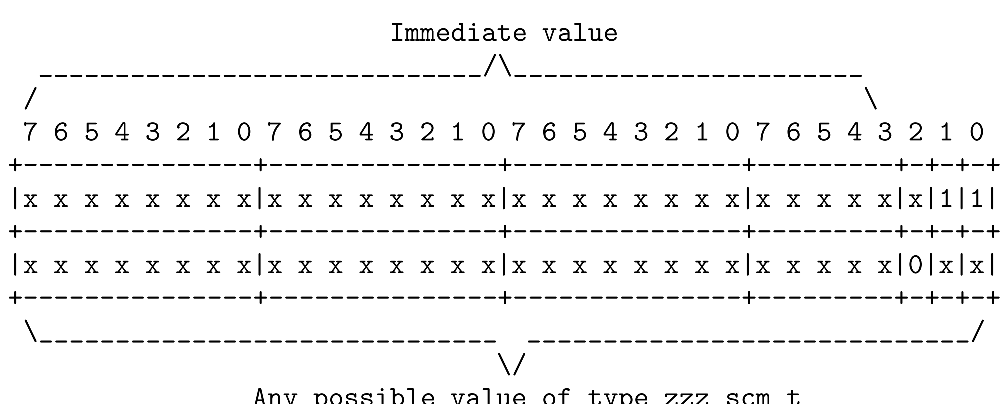
Used cons cells can be in one of the following four states.

The car of a cons cell either contains a cons cell pointer (or NULL), a tagged cell pointer, a fixnum or an immediate value. During garbage collection, bit #2 of the car may be set.


Pointer to cons cell
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|0|0|
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|0|x|x|
Any possible value of type zzz_scm_t


Pointer to tagged cell
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|1|0|
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|0|x|x|
Any possible value of type zzz_scm_t


Integer value
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|0|1|
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|0|x|x|
Any possible value of type zzz_scm_t


Immediate value
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|1|1|
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|0|x|x|
Any possible value of type zzz_scm_t

The first word of a tagged cell always containes a type tag (and maybe information in the upper bits); the tag 0 stands for an unused cell. The following three words may hold any value, they are not interpreted by the memory manager.

Free tagged cell:

```
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------+-+-+-+
|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0|0 0 0 0 0|0|1|0|
+---------------+---------------+---------------+---------+-+-+-+
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|x|x|
.\_____________________________  _____________________________/.
.                              \/                              .
.                    Pointer to next free cell                 .
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|x|x|
+---------------+---------------+---------------+---------+-+-+-+
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|x|x|
+---------------+---------------+---------------+---------+-+-+-+
 \_____________________________  _____________________________/
                               \/
                             Garbage
```

A use tagged cell looks like this. Note that the third bit of the first word may be set during garbage collection, just like for cons cells.

```
      Type information               Type tag
  ___________/\__________   _____________/\_____________
 /                      \ /                            \
 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------+-+-+-+
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|0|0|
+---------------+---------------+---------------+---------+-+-+-+
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|x|x|
+---------------+---------------+---------------+---------+-+-+-+
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|x|x|
+---------------+---------------+---------------+---------+-+-+-+
|x x x x x x x x|x x x x x x x x|x x x x x x x x|x x x x x|x|x|x|
+---------------+---------------+---------------+---------+-+-+-+
 \_____________________________  _____________________________/
                               \/
              Pointer(s) to cell data / immediate data
```

4.2 Garbage Collection

In order to free the programmer from the burden of memory management, the Scheme runtime environment must reclaim storage when it is no longer accessible by the running program. This process is called *Garbage Collection*, or GC for short.

Memory allocation in Sizzle works as follows: functions creating data objects call either the function `zzz_alloc_tagged_cell()` or `zzz_cons()` (which in turn calls `zzz_alloc_cons_cell()`). After a certain number of calls to one of these functions, the garbage collection is invoked. When not enough free memory is reserved for fulfilling the allocation request, more memory is requested from the operating system and the space needed is reserved. Note that Sizzle does not invoke the garbage collector when the freelists are empty, because it tries to avoid garbage collection when memory is nearly full.

The garbage collector uses a simple mark–and–sweep algorithm. That means, that garbage collection starts by marking all directly and indirectly reachable objects as referenced. In the second step, all objects which are not marked are returned to the freelists of their respective types.

For marking all referenced objects it is necessary to find the *root pointers* through which all used objects can be reached. Sizzle performs *conservative marking*, that is, it marks all data objects which are pointed to by values on the stack of the C program it runs in and by marking all objects in locations which have been explicitly made known to the interpreter. This mechanism requires that the interpreter knows how to find the beginning and the end of the stack.

Two methods are provided to make these addresses available: The first is to make the embedding application calling a library function which calls back to the procedure doing the applications work. With this method, the library can calculate the end of the stack automatically, but it takes over the main control function of the program. The Sizzle command line interpreter uses this method.

The other method is to make the client code call explicitly a library function and tell it where the end of the stack is by passing the address of a local variable. The tutorial in this manual is an example for this procedure (see Section 2.2 [Parsing initialization files], page 2).

Object Marking

The marking phase is performed by traversing all objects pointed by the root pointers. Immediate objects do not need to get marked, because they do not use heap memory at all. What happens to other objects, depends on their type. Cons cells are directly known to the mark function, and the funtion tries to reduce the needed stack space by avoiding deep recursion. Therefore, the spine of lists is marked in a loop, marking recursively only the list elements. Because lists in Scheme are normally long and shallow, this speeds up marking a lot.

Tagged objects are marked by calling the mark function for the type of the object. The mark function is responsible for calling the mark function recursively for all Scheme objects referenced by the object.

5 Creating Data Types

Sizzle has been designed in a way that it is easy to add additional data types. For now, this section only crudely documents the method for defining data types, so for details, please refer to the source code to find out how to do it. Start by reading the files '`node.h`' and '`node.c`' where all builtin data types are defined.

The internal representation of all data is documented in Section 4.1 [Cell Representation], page 22.

The API functions for handling tagged types are documented briefly in Section 3.10 [Defining Scheme Types], page 21.

New data types in Sizzle are introduced by defining so-called *tagged types*. Objects of tagged types occupy at least one tagged cell on the heap, but they may allocate additional memory and store pointers to those allocated areas into the heap cells. For every tagged type, a bunch of function must be defined which tell the memory manager to handle objects of these types without actually knowing the memory layout of the types.

5.1 Type Functions

This section documents what type functions are, and how to write them. First, I will summarize what functions are required, then I will describe each function type in detail, giving examples as I go.

The following type functions are defined:

Constructor function
: This function gets called when an object is created. May allocate memory which may be needed to store an object, and must initialize the object from data passed to the constructor.

Print function
: This gets called by primitives like `display` or `write`, and is responsible for printing a textual representation of the object to a port.

Free function
: Destructor function, which must free all memory which has been allocated by the constructor.

Mark function
: Called by the garbage collector during the mark phase for traversing all reachable objects.

Equal function
: Equality predicate called by `equal?` to determine whether two objects of the same type have the same structure.

5.1.1 Constructor Function

The constructor function is responsible for initializing a tagged object. When it is called, a tagged cell is already allocated on the heap and has been initialized with the tagged type tag for the object. The constructor has three formal arguments: The address of the allocated cell, and three void pointer, which are used to pass arguments to the constructor function. The actual type and value of these arguments depend on the object type the cosntructor function was defined for.

The prototype for constructor functions looks like this:

```
typedef void * (* constructor_func_t) (zzz_tagged_t tagged, void * data0,
                                       void * data1, void * data2);
```

Often, types only need to store their three pointer arguments in the three data slots of a tagged cell. These types can use the library function `zzz_simple_constructor()`, which does this cell initialization. User types are free to use this pre-defined constructor functions for their types, if its specification suits their needs.

The constructor function for the string datatype is given below, to serve as a non-trivial example.

```
static void *
string_constructor (zzz_tagged_t tagged, void * data0,
                    void * data1, void * data2)
{
  char * p = (char *) data0;
  int len =  (int) ((long) data1);
  void * ret;

  ret = zzz_malloc (len + 1);   /* +1 for terminating '\0'.  */
  if (p)
    memmove ((char *) ret, p, len);
  ((char *) ret)[len] = '\0';
  tagged->fill[1] = (unsigned long) ret;
  tagged->fill[2] = (unsigned long) len;
  return tagged;
}
```

5.1.2 Print Function

Print functions must print a textual representation of an object to a Scheme port. They must match the following protoype:

```
typedef void (* print_func_t) (zzz_scm_t port, zzz_tagged_t tagged,
                               zzz_print_state_t state);
```

The argument *port* specifies the Scheme port to print to, *tagged* is the object, and *state* specifies the current state for printing. `zzz_print_state_t` is defined as a pointer to the following structure, which currently includes only one field. This field is a bitset for which the `PRINT_*` macros have been defined.

```
struct zzz_print_state
{
  int flags;
};

#define PRINT_NEWLINE 0x01
#define PRINT_QUOTE   0x02
#define PRINT_ESCAPE  0x04
```

`PRINT_NEWLINE` is currently unused. `PRINT_QUOTE` means, that the print function should do all necessary quoting for making the textual representation suitable for reading the object back in using `read`. `PRINT_ESCAPE` means that special characters must be escaped in a way compatible with `read`. `write` for example, calls the print functions with both `PRINT_QUOTE` and `PRINT_ESCAPE` set, `display` calls the function without these flags.

The print function for strings is rather complicated, but because it interprets the print flags, it makes a good example.

```
static void
string_print (zzz_scm_t port, zzz_tagged_t tagged, zzz_print_state_t state)
{
  zzz_scm_t result;
  char * p = tagged_data0 (tagged);
  int len = (long) tagged_data1 (tagged);

  assert (len >= 0);
  if (!(state->flags & (PRINT_ESCAPE | PRINT_QUOTE)))
    zzz_port_puts (port, p, len, &result); /* Fast path for ‘display’.  */
  else
    {
      if (state->flags & PRINT_QUOTE)
        zzz_port_putc ('"', port, &result);
      while (len-- > 0)
        {
          if ((state->flags & PRINT_ESCAPE) && (*p == '\\' || *p == '"'))
            zzz_port_putc ('\\', port, &result);
          zzz_port_putc (*p, port, &result);
          p++;
        }
      if (state->flags & PRINT_QUOTE)
        zzz_port_putc ('"', port, &result);
    }
}
```

5.1.3 Free Function

The free function must free all memory which might have been allocated in the constructor. As an example, here is the string type free function.

```
static void
string_free (zzz_tagged_t tagged)
{
  int len =  (long) tagged_data1 (tagged);
  zzz_free (tagged_data0 (tagged), len + 1); /* + 1 for terminating '\0' */
}
```

5.1.4 Equal Function

The equal function must determine whether two objects of the same type have the same structure in the sense of `equal?`. For strings, that means that the strings must be tested whether they have the same length and the same character contents.

```
static int
string_equal (zzz_tagged_t tagged0, zzz_tagged_t tagged1)
{
  char * p1 = tagged_data0 (tagged0);
  char * p2 = tagged_data0 (tagged1);
  int len1 = (long) tagged_data1 (tagged0);
  int len2 = (long) tagged_data1 (tagged1);

  if (len1 != len2)
    return 0;
  if (len1 == 0)
    return 1;
  len1 = memcmp (p1, p2, len1);
  if (len1)
    return 0;
  else
    return 1;
}
```

5.1.5 Mark Function

The mark function for strings is empty, because no other Scheme objects are referenced. It would be possible to pass a null pointer to the type definition function also, if no mark function is required.

```
static void
string_mark (zzz_tagged_t tagged)
{
}
```

Here is a second example, the mark function for vector objects. It calls the library mark function for every vector element.

```
static int
vector_equal (zzz_tagged_t tagged0, zzz_tagged_t tagged1)
{
  zzz_scm_t * vec1   = vector_val (tagged0);
  unsigned long len1 = vector_len (tagged0);
  zzz_scm_t * vec2   = vector_val (tagged1);
  unsigned long len2 = vector_len (tagged1);
  unsigned long x;

  if (len1 != len2)
    return 0;
  for (x = 0; x < len1; x++)
    {
      if (!zzz_equal (vec1[x], vec2[x]))
        return 0;
    }
  return 1;
}
```

Like for the constructor function, a library function exists to be used as a mark function when your tagged type has a certain structure. Whenever a tagged type is to be created which should hold

Scheme values in the data slots of the tagged cells, you can use the function `zzz_simple_mark()`. This function simply calls the system's mark function for all three data slots.

5.2 Defining Tagged Types

After writing the type functions, you have to register the new type with the library. This is done by calling the type definition function `zzz_define_tagged_type()`, which has this protoype:

```
unsigned long zzz_define_tagged_type
          (constructor_func_t constructor_func,
           print_func_t print_func,
           free_func_t free_func,
           mark_func_t mark_func,
           equal_func_t equal_func);
```

The return value of this function is the type tagged, which can then be used to create objects of the given type. It is used in calls to the tagged typ creation function `zzz_create_tagged_type()`. It is called with the type tag obtained by the call to the register function and three opaque pointers, which will be passed to the type's constructor function.

```
zzz_scm_t zzz_create_tagged_cell (unsigned long tag, void * data0,
                                  void * data1, void * data2);
```

6 Creating Port Types

New port types can be created on the C as well as on the Scheme level. For now, refer to the files 'sport.c' and 'sport.h', which define string ports and which will be instructive if you want to build your own port types in C. The other port types are implemented in the files 'fport.[ch]', 'fdport.[ch]' and 'scmport.[ch]'.

On the Scheme level, the procedure make-soft-port can be used to create ports which handle in- and output in a special way. Refer to the Reference Manual for more information. Scheme ports are implemented in the files 'scmport.c' and 'scmport.h'.

7 Adding Primitives

It is easy to add primitive procedures to the interpreter. The process involves two steps. You have to write a C function which handles the primitive procedure calls and you must tell the interpreter under which name your function wants to be called.

7.1 The C Function

This is a small example for writing a C primitive. The code is actually taken from the code of the Sizzle library, it implements the primitive function `eval`.

```
/*:doc
  eval
  (eval expr [environment]) => value(s)
  Evaluate the expression expr in the environment specified by
  environment and return the resulting value(s).  environment
  defaults to the current top-level environment.
  doc:*/
#define FUNC_NAME "eval"
static result_t
eval_func (zzz_scm_t env, zzz_scm_t form, zzz_scm_t new_env,
           zzz_scm_t * result)
{
  static char * s_func_name = FUNC_NAME;
  zzz_scm_t res;

  if (zzz_eq (new_env, zzz_undefined))
    new_env = zzz_current_environment;
  else
    CHECK_TYPE (2, env_p (new_env), zzz_environment_type_name);
  return zzz_evaluate (new_env, form, result);
}
#undef FUNC_NAME
```

The first thing to notice is the definition of the CPP macro `FUNC_NAME`. This macro must always be defined to be the name of the Scheme primitive this function stands for, because it is used in error messages produced by several support macros. `s_func_name` must be declared always for the same reason, and in exactly the same way.

`eval` takes an optional second parameter. Therefore, the code must check whether any value was given for that parameter, if not the value of the parameter will be `zzz_undefined`. Depending on this check, `eval` either provides a default value or checks the given value's type. The macro `CHECK_TYPE` will produce an error message and return from the function if it is not. The first argument to `CHECK_TYPE` is the parameter number to include in error messages, the second is an expression which must evaluate to true in order to proceed, and the last parameter is a string telling which parameter type was expected. When the test for the second argument fails, `CHECK_TYPE` will generate an appropriate error message including the error message from the last parameter and return immediately from the current function.

When the parameters have been verified, the function `zzz_evaluate()` is called to evaluate the first parameter. The return value if that call is simply returned, an the result pointer is passed to the evaluation function, which will store the result value in the given location.

7.2 Adding the Function to the Core

After writing the C function for a primitive, we have to announce to the library what we have laborously achieved. This is done using the API function `zzz_define_function()`, like in the following example.

```
zzz_define_function ("eval", eval_func, argv_1_1_0);
```

Here, `"eval"` is the name of the Schme primitive we want to bind the function to, *eval_func* is the C function we have defined in the previous section and `argv_1_1_0` is a descriptor of the argument format our function expects. There are several of those `argv_*_*_*` macros defined in '`node.h`'. The first number stands for the number of fixed arguments the procedure expects, the second number for the number of optional arguments and the last number is 1 if a rest parameter will be accepted, to which a list of all remaining arguments will be bound; and 0 if no rest argument is acceptable.

8 sizzle-gen-if

Included in the Sizzle distribution is the script '`sizzle-gen-if`'. This script can be used to automatically generate glue code for C functions from an interface specification. It is possible (in some simple cases) to generate a primitive function for wrapping a C library function without writing a single line of code. An example for this is the procedure `fnmatch`, which was included into the Sizzle core by specifying its interface, and the rest (primitive procedure code, parameter checking, parameter unboxing, return value boxing, primitive definition, constant definition) was handled by '`sizzle-gen-if`'.

This section documents the use of the script. For further information, you can of course always refer to the source code and the file '`fnmatch.if`' in the directory '`libsizzle`' of the distribution, which demonstrates the use.

8.1 sizzle-gen-if invocation

'`sizzle-gen-if`' is used as follows.

```
$ sizzle-gen-if infile outfile
```

The script must be invoked with the input file as the first and the primary output file as the second argument. It will copy the input file to the output file and replace interface specification marked with a `$` character with the corresponfing generated glue code. Additionally, a *init file* will be created, with the name of the output file with and '`.x`' appended. This file will contain the necessary initialization code for registering primitives, creating constants and symbols and protecting global variables from garbage collection. The output file can then be `#include`'d into any C source file, and the init file should be `#include`'d into the init function of that source file, so that initialization can take place for the generated wrapper code.

Suppose you have an interface specification file called '`fnmatch.if`', which specifies how the C function `fnmatch` must be called, and which possibly defines some constants or symbols. To generate the glue code and the initialization statements, you have to call '`sizzle-gen-if`' with the name of the file and the name the output file should have.

```
mgrabmue@tortoise (~/cvs/sizzle/scripts): sizzle-gen-if fnmatch.if fnmatch.c
mgrabmue@tortoise (~/cvs/sizzle/scripts):
```

When no errors are found, '`sizzle-gen-if`' does not print any messages. You can have a look at the current directory to see which files have been created.

```
mgrabmue@tortoise (~/cvs/sizzle/scripts): ls -l fnmatch.*
-rw-r--r--   1 mgrabmue mgrabmue     3263 Aug 22 15:26 fnmatch.c
-rw-r--r--   1 mgrabmue mgrabmue      568 Aug 22 15:26 fnmatch.c.x
-rw-r--r--   1 mgrabmue mgrabmue      658 Aug 22 12:26 fnmatch.if
mgrabmue@tortoise (~/cvs/sizzle/scripts):
```

As you can see, the output file '`fnmatch.c`' exists now, and it is considerably bigger than the input file. This can give you a feeling how much typing you have saved by using '`sizzle-gen-if`' :-)

Additionally, the initialization file '`fnmatch.c.x`' was created. It holds the code needed to register the newly generated primitive procedure, so that is gets accessible to the outer world.

8.2 The Generation Process.

An interface specification file is processed by '`sizzle-gen-if`' under the following rules:

- All data is copied verbatim to the output file, except when escaped with the escape character `$`, which indicates the beginning of an interface specification.

- The escape character is `$`. After this character, a Scheme object is read using the standard procedure `read` and processed as an interface specification. All whitespace following the specification will be discarded. The result of applying the interface specification will then be written to the output file.
- Additionally, data may be written to the initialization file as needed. This can be code for registering primitives, protecting global variables from garbage collection and creation of symbols and constant values.

8.3 The Specification File

A specification file can contain arbitrary text. The use of interface specifications must be explicitly announced with the character `$`. All other data is simply copied to the output file. This is useful to include things like preprocessor statements, other variable declarations or even helper functions in the interface file.

When a `$` character is read, a Scheme expression representing an interface specification is expected. After reading this expression (which will discard any whitespace after the expression), character copying starts again, until a `$` is read, and so on.

This is the grammar for an interface specification.

```
<if-spec>         ::= <func-spec> | <cpp-const-spec> |
                      <symbol-spec> | <exception-spec>
<func-spec>       ::= "(" "function" <ret-type> <func-name>
                      "(" <param-list> ")" [glue-code] ")"
<func-name>       ::= identifier
<ret-type>        ::= <type> | "<void>"
<param-list>      ::= <empty> | <param> <param-list>
<param>           ::= "(" <param-name> <param-type> [<keyword> ...] ")"
<param-name>      ::= identifier
<param-type>      ::= <type>
<keyword>         ::= "invalidate" | "unchecked" | "ignored" |
                      "optional" | "rest"
<type>            ::= "<int>" | "<string>" | "<real>" | "<file>" |
                    "<pointer>" | "<object>"
<glue-code>       ::= string
<cpp-const-spec> ::= "(" "cpp-constant" <c-name> <type> ")"
<scheme-name>     ::= identifier
<symbol-spec>     ::= "(" "symbol" <scheme-name> <c-name> ")"
<exception-spec> ::= "(" "exception" <scheme-name> <c-name> ")"
<c-name>          ::= identifier
```

<*func-name*> must be the name of the C function to be wrapped. The wrapper function will be called `<func-name>_func`. <*ret-type*> is the expected return type of the function and may be `<void>` if it is to be ignored. The return type <ret-type> is used to construct the correct return value for the function's result. The parameter list <*param-list*> specifies the names of the parameters for the function and their types, so that type-checking code can be emitted.

The option `unchecked` says that no type checking is performed for this parameter, it will be passed through to the called function. Note that this seldom makes sense, except for debug printing or for wrapping a function aware of Sizzle's Scheme types.

The option `ignored` tells the processor not to emit type checking code for this parameter, and that it will not be passed to the wrapped function. It will be silently ignored.

The additional keyword `invalidate`, which can be passed as a third element in a parameter specification, is currently only implemented for pointer values (type `<pointer>`). It will cause

the emission of extra code which will invalidate the parameter pointer object with a call to `zzz_invalidate_pointer()`. This is useful if the wrapped function performed some action rendering the pointer in the pointer object useless. An invalidated pointer cannot be passed to any wrapped function expecting `<pointer>` arguments after that, because they check a pointer object's valid flag. Thus pointer object handling gets a little bit more safe.

When `optional` is passed as an option, then the passed parameter is allowed to be undefined. You should either specify `ignored` also, to avoid passing an undefined value to the wrapped function, or you must give your own <*glue-code*> which handles that case properly.

The option `rest` may be given on the last argument. When given, it means that this argument will be the rest argument to which a list of the remaining argument will be bound. The type of this variable is always a list, so no parameter type checking code is emitted. The type for this parameter should always be `<object>`.

The specifications for symbols and exception will create global variables called <*c-name*> and bind symbol objects with the contents <*scheme-name*> to them. Also, the variables will be protected from garbage collection.

The <*cpp-const-spec*> specification creates a Scheme constant which is bound to the value which gets substituted by the CPP macro <*c-name*>. The <*type*> argument is needed to create the constructor for the constant object.

8.4 A Simple Example

To start with, I want to give a feeling of what '`sizzle-gen-if`' can do. At first, it may seem extremely powerful, but sooner or later you will find out that the C functions you want to wrap are not as well suited as in this (contrived) example.

Take this specification:

```
$(function <int> strlen ((string <string>)))
```

Here we declare a function

- With the name `strlen`.
- With the return value `<int>`.
- Which expects one argument of type `<string>` with the name `string`.

When passed through '`sizzle-gen-if`', this specification is transformed to this C function.

```
/*:doc
  strlen
  (strlen string) => integer
  Apply the C function strlen to the following parameters:
  string: string
  doc:*/
#define FUNC_NAME "strlen"
static result_t
strlen_func (zzz_scm_t env, zzz_scm_t string, zzz_scm_t * result)
{
  static char * s_func_name = FUNC_NAME;
  zzz_scm_t __retval;

  CHECK_TYPE (1, string_p (string), zzz_string_type_name);
  {
    int res = strlen (string_val (string));
    __retval = zzz_make_integer (res);
  }
```

```
  *result = __retval;
  return RESULT_SUCCESS;
}
#undef FUNC_NAME
```

Note how the type of the actual parameter is checked for correctnes. Then the string value is extracted from the string object before calling the wrapped function, and the result of the function call is stored into an integer object before returning it.

The specification and will also generate an init file containing the following code, which will register the primitive under the name `strlen` as a function taking exactly one argument.

```
  zzz_define_function ("strlen",
                       (zzz_prim_t) strlen_func,
                       argv_1_0_0);
```

8.5 Manual Glue Code

The automatically generated wrapper functions are nice, but they are not as flexible as often is needed. For example, it is not possible to specify `<number>` as an argument type and have the script automatically generate code which selects the value depending on the actual type of the parameter, which may be integer or real. Therefor, the additional parameter <*glue-code*> is provided for interface specifications. When this code is present, the body of the wrapper procedure will not be generated, but instead <*glue-code*> will be inserted. The glue code must assign a return value to the variable *__retval*, but the positive effect is that argument type checking is performed, so the glue code can rely on the fact that the parameters have the types specified in the interface specification.

Consider this specification for the primitive `sin`:

```
$(function <real> sin ((x <number>))
"
    double d;

    if (integer_p (x))
      d = (double) integer_val (x);
    else if (float_p (x))
      d = float_val (x);
    else
      abort ();
    __retval = zzz_make_float (sin (d));
")
```

The glue code tests which number type the parameter *x* has, and performs the correct action according to the type. When processed with '`sizzle-gen-if`', the resulting code looks as follows:

```
/*:doc
  sin
  (sin x) => real
  Apply the C function sin to the following parameters:
  x: <number>
  doc:*/
#define FUNC_NAME "sin"
static result_t
sin_func (zzz_scm_t env, zzz_scm_t x, zzz_scm_t * result)
{
  static char * s_func_name = FUNC_NAME;
```

```
  zzz_scm_t __retval;

  CHECK_TYPE (1, number_p (x), zzz_number_type_name);
  {
    double d;

    if (integer_p (x))
      d = (double) integer_val (x);
    else if (float_p (x))
      d = float_val (x);
    else
      abort ();
    __retval = zzz_make_float (sin (d));
  }
  *result = __retval;
  return RESULT_SUCCESS;
}
#undef FUNC_NAME
```

Additionally, the following initialization file will be created:

```
  zzz_define_function ("sin",
                       (zzz_prim_t) sin_func,
                       argv_1_0_0);
```

8.6 Pointer arguments

A lot of C library functions take pointers as their arguments. It is not possible for '`sizzle-gen-if`' to understand the meaning of all the possible weird argument passing techniques C programmers have invented. Thus, '`sizzle-gen-if`' decides that all data types it cannot understand are opaque pointers. This does not catch all cases, but you already knew that '`sizzle-gen-if`''s capabilities were limited :-)

This is an example specification for the C library function `free()`. It demonstrates how to use the option keyword `invalidate` with pointer arguments.

```
$(function <void> free ((p <pointer> invalidate)))
```

expands to

```
/*:doc
  free
  (free p) => *unspecified*
  Apply the C function free to the following parameters:
  p: pointer
  doc:*/
#define FUNC_NAME "free"
static result_t
free_func (zzz_scm_t env, zzz_scm_t p, zzz_scm_t * result)
{
  static char * s_func_name = FUNC_NAME;
  zzz_scm_t __retval;

  CHECK_TYPE (1, valid_pointer_p (p), "pointer");
  {
    free (pointer_val (p));
```

```
        __retval = zzz_unspecified;
        zzz_invalidate_pointer (p);
      }
      *result = __retval;
      return RESULT_SUCCESS;
    }
    #undef FUNC_NAME
```

When `free_func()` returns, the pointer *p* cannot be used with wrapper functions anymore, because it has been invalidated. This is what you want after you free a pointer, isn't it? Using this trick, '`sizzle-gen-if`' can even make programming with pointers safer.

Index

B

bool_p 13
bool_val 16
bool_var_addr 18
bool_var_p 14

C

car 15
car_addr 15
cdr 15
cdr_addr 15
char_p 13
char_val 16
cons_p 13
constant_p 14
constant_val 17

E

env_p 14
env_val 17
error_p 14
except_p 14

F

fdport_fd 19
fdport_has_unget 19
fdport_p 15
fdport_unget 19
fixnum_p 12
fixnum_val 16
float_p 13
float_val 16
form_p 13
fport_file 19
fport_p 15
func_form_p 16
func_name 16
func_p 13
func_param 17
func_ptr 16

G

gloc_name 17
gloc_p 14
gloc_val 17

I

imm_p 13
immediate_p 13
immediate_val 16
int_var_addr 18
int_var_p 14
integer_p 13
integer_val 16

K

keyword_name 17
keyword_p 14

L

lambda_args 17
lambda_body 17
lambda_env 17
lambda_file 17
lambda_line 17
lambda_name 17
lambda_p 13
list_p 13
lloc_p 14
lloc_val 17
location_address 17
location_p 14
long_p 14
long_val 17

M

macro_code 18
macro_p 15

N

null_p 12
number_p 13

P

port_col_number 19
port_line_number 19
port_open_p 18
port_p 15
port_ptype 18
port_read_p 18
port_saved_col 19
port_write_p 18
procedure_p 13
promise_env 18
promise_expr 18
promise_p 14
promise_result 18
promise_thawed 18

R

regexp_p 14
regexp_regex 18
regexp_string 18
regexp_valid 18
ro_bool_var_p 14
ro_int_var_p 14
ro_str_var_p 14
ro_*TAG*vector_p 15
rolocation_p 14
rostring_p 13
rovector_p 14

S

set_car 15
set_cdr 15
set_symbol_name 16
set_symbol_value 16
sport_len 19
sport_p 15
sport_pos 19
sport_size 19
sport_val 19
str_var_len 18
str_var_p 14
str_var_size 18
str_var_val 18
string_len 16
string_p 13
string_val 16
symbol_name 16
symbol_name_addr 16
symbol_p 13
symbol_value 16
symbol_value_addr 16
syntax_p 15
syntax_rules 18

T

tagged_data0 16
tagged_data1 16
tagged_data2 16
tagged_info 15
tagged_p 13
tagged_type 15
*TAG*vector_len 19
*TAG*vector_p 15
*TAG*vector_val 19

V

values_p 14
values_val 18
vector_len 17
vector_p 14
vector_val 17

Z

zzz_apply 8
zzz_bind_bool_variable 7
zzz_bind_int_variable 7
zzz_bind_scm_variable 7
zzz_bind_string_variable 7
zzz_cons 8
zzz_copy_list 12
zzz_copy_tree 12
zzz_create_tagged_cell 11
zzz_define_constant 6
zzz_define_form 6
zzz_define_function 6
zzz_define_port_type 19
zzz_define_tagged_type 21
zzz_define_variable 6
zzz_eq 21
zzz_equal 21
zzz_eqv 21
zzz_evaluate 8
zzz_evaluate_file 8
zzz_evaluate_string 8
zzz_finalize 5
zzz_garbage_collect 21
zzz_get_variable 6
zzz_hash_function 6
zzz_hash_ref 6
zzz_hash_set 7
zzz_hashq_ref 6
zzz_hashq_set 7
zzz_hashv_ref 6
zzz_hashv_set 7
zzz_initialize 5
zzz_list_length 21
zzz_make_bool 9
zzz_make_char 9
zzz_make_closure 9
zzz_make_constant 10
zzz_make_continuation 10
zzz_make_environment 10
zzz_make_error 9
zzz_make_exception 10
zzz_make_fdport 11
zzz_make_fixnum 8
zzz_make_float 9
zzz_make_fport 11
zzz_make_func 9
zzz_make_gloc 10
zzz_make_integer 9
zzz_make_keyword 10

zzz_make_lambda 9
zzz_make_list 8
zzz_make_lloc 10
zzz_make_location 10
zzz_make_macro 11
zzz_make_n_list 8
zzz_make_not_available_exception 12
zzz_make_nstring 9
zzz_make_pointer 12
zzz_make_pointer_n 12
zzz_make_port 12
zzz_make_promise 11
zzz_make_regexp 11
zzz_make_ro_nstring 9
zzz_make_ro_string 9
zzz_make_ro_*TAG*vector 12
zzz_make_ro_vector 10
zzz_make_sport 11
zzz_make_sport_str 11
zzz_make_sport_str_n 11
zzz_make_sport_str_n_no_copy 11
zzz_make_string 9
zzz_make_symbol 9
zzz_make_syntax 11
zzz_make_*TAG*vector 12
zzz_make_values 11
zzz_make_vector 10
zzz_mark_cell 21
zzz_port_char_ready_p 20
zzz_port_close 20
zzz_port_flush 20
zzz_port_getc 20
zzz_port_print 20
zzz_port_putc 20
zzz_port_puts 20
zzz_port_seek 20
zzz_port_tell 20
zzz_port_ungetc 20
zzz_protect_global 7
zzz_read_eval_print 8
zzz_run 5
zzz_set_arguments 5
zzz_set_top_of_stack 5
zzz_set_variable 6
zzz_simple_constructor 21
zzz_simple_mark 21
zzz_vector_get 17
zzz_vector_put 17

Table of Contents

1 Introduction **1**

2 Embedding Sizzle **2**

2.1 Compiling and Linking 2
2.2 Parsing Initialization files 2
2.2.1 Initializing the library 2
2.2.2 Declaring variables 3
2.2.3 Loading the init file 3
2.2.4 Shutting down the library 4

3 Embedding API **5**

3.1 Initializing and Finalization 5
3.2 Scheme Variable Handling 5
3.3 Hash table handling 6
3.4 C Variable Handling 7
3.5 Evaluation Functions 7
3.6 Object constructors 8
3.7 Type predicates 12
3.8 Accessor functions 15
3.9 Port Functions 19
3.10 Defining Scheme Types 21
3.11 Misc C API Functions 21

4 Memory Management **22**

4.1 Cell Representation 22
4.2 Garbage Collection 26

5 Creating Data Types **28**

5.1 Type Functions 28
5.1.1 Constructor Function 28
5.1.2 Print Function 29
5.1.3 Free Function 30
5.1.4 Equal Function 30
5.1.5 Mark Function 31
5.2 Defining Tagged Types 32

6 Creating Port Types **33**

7 Adding Primitives **34**

7.1 The C Function 34
7.2 Adding the Function to the Core 35

8 sizzle-gen-if 36

8.1 sizzle-gen-if invocation 36
8.2 The Generation Process 36
8.3 The Specification File 37
8.4 A Simple Example 38
8.5 Manual Glue Code 39
8.6 Pointer arguments 40

Index 42