Inside PLT MzScheme

Matthew Flatt
mflatt@cs.utah.edu

Version 103
August 2000

Rice University
University of Utah

Copyright notice

Send us your Web links

If you use any parts or all of the DrScheme package (software, lecture notes) for one of your courses, for your research, or for your work, we would like to know about it. Furthermore, if you use it and publicize the fact on some Web page, we would like to link to that page. Please drop us a line at *scheme@cs.rice.edu*. Evidence of interest helps the DrScheme Project to maintain the necessary intellectual and financial support. We appreciate your help.

Thanks

Some typesetting macros were originally taken from Julian Smart's *Reference Manual for wxWindows 1.60: a portable C++ GUI toolkit*.

Contents

1. Overview

This manual describes MzScheme's C interface, which allows the interpreter to be extended by a dynamically-loaded library, or embedded within an abitrary C/C++ program. The manual assumes familiarity with MzScheme, as described in *PLT MzScheme: Language Manual*.

1.1 Writing MzScheme Extensions

To write a C/C++-based extension for MzScheme, follow these steps:

- For each C/C++ file that uses MzScheme library functions, `#include` the file **escheme.h**.

  This file is distributed with the PLT software in **plt/collects/mzscheme/include**, but if **mzc** is used to compile, this path is found automatically.

- Define the C function `scheme_initialize`, which takes a `Scheme_Env *` namespace (see §2.3) and returns a `Scheme_Object *` Scheme value.

  This initialization function can install new global primitive procedures or other values into the namespace, or it can simply return a Scheme value. The initialization function is called when the extension is loaded with `load-extension` (the first time); the return value from `scheme_initialize` is used as the return value for `load-extension`. The namespace provided to `scheme_initialize` is the current namespace when `load-extension` is called.

- Define the C function `scheme_reload`, which has the same arguments and return type as `scheme_initialize`.

  This function is called if `load-extension` is called a second time (or more times) for an extension. Like `scheme_initialize`, the return value from this function is the return value for `load-extension`.

- Compile the extension C/C++ files to create platform-specific object files.

  The **mzc** compiler, distributed with MzScheme, compiles plain C files when the `--cc` flag is specified. Actually, **mzc** does not compile the files itself, but it locates a C compiler on the system and launches it with the appropriate compilation flags. If the platform is a relatively standard Unix system, a Windows system with either Microsoft's C compiler or **gcc** in the path, or a MacOS system with Metrowerks CodeWarrior installed, then using **mzc** is typically easier than working with the C compiler directly.

- Link the extension C/C++ files with **mzdyn.o** (Unix) or **mzdyn.obj** (Windows) to create a shared object.

  The **mzdyn** object file is distributed in a platform-specific directory in **plt/collects/mzscheme/lib** for Unix or Windows, but it is not distributed for MacOS.

  The **mzc** compiler links object files into an extension when the `--ld` flag is specified, automatically locating **mzdyn**. Under MacOS, **mzc** generates the **mzdyn** object file as necessary.

- Load the shared object within Scheme using `(load-extension` *path*`)`, where *path* is the name of the extension file generated in the previous step.

IMPORTANT: Scheme values are garbage collected using a conservative garbage collector, so pointers to MzScheme objects can be kept in registers, stack variables, or structures allocated with `scheme_malloc`. However, static variables that contain pointers to collectable memory must be registered using `scheme_register_extension_global` (see §2.2).

As an example, the following C code defines an extension that returns `"hello world"` when it is loaded:

```
#include "escheme.h"
Scheme_Object *scheme_initialize(Scheme_Env *env) {
  return scheme_make_string("hello world");
}
Scheme_Object *scheme_reload(Scheme_Env *env) {
  return scheme_initialize(env); /* Nothing special for reload */
}
```

Assuming that this code is in the file **hw.c**, the extension is compiled under Unix with the following two commands:

mzc `--cc` **hw.c**
mzc `--ld` **hw.so hw.o**

(Note that the `--cc` and `--ld` flags are each prefixed by two dashes, not one.)

The **plt/collects/mzscheme/examples** directory in the PLT distribution contains additional examples.

1.2 Embedding MzScheme into a Program

To embed MzScheme in a program, first download the MzScheme source code. Then, follow these steps:

- Compile the MzScheme libraries.

  Under Unix, the libraries are **libmzscheme.a** and **libgc.a**. After compiling MzScheme and running **make install**, the libraries are in a platform-specific directory under **plt/collects/mzscheme/lib/**. Under Windows and MacOS, consult the compilation instructions for information on compiling the libraries.

- For each C/C++ file that uses MzScheme library functions, `#include` the file **scheme.h**.[1]

  This file is distributed with the PLT software in **plt/collects/mzscheme/include**.

- In your main program, obtain a global MzScheme environment `Scheme_Env *` by calling `scheme_basic_env`. This function must be called before any other function in the MzScheme library (except `scheme_make_param`).

- Access MzScheme though `scheme_load`, `scheme_rep`, `scheme_eval`, and/or other top-level MzScheme functions described in this manual.

- Compile the program and link it with the MzScheme libraries.

Scheme values are garbage collected using a conservative garbage collector, so pointers to MzScheme objects can be kept in registers, stack variables, or structures allocated with `scheme_malloc`. In an embedding application, static variables are also automatically registered as roots for garbage collection (but see the Windows-specific note below).

[1] The C preprocessor symbol `SCHEME_DIRECT_EMBEDDED` is defined as `1` when **scheme.h** is `#included`, or as `0` when **escheme.h** is `#included`.

For example, the following is a simple embedding program which evaluates all expressions provided on the command line and displays the results:

```
#include "scheme.h"
int main(int argc, char *argv[])
{
  Scheme_Env *e = scheme_basic_env();
  Scheme_Object *curout = scheme_get_param(scheme_config, MZCONFIG_OUTPUT_PORT);
  int i;
  for (i = 1; i < argc; i++) {
    if (scheme_setjmp(scheme_error_buf)) {
      return -1; /* There was an error */
    } else {
      Scheme_Object *v = scheme_eval_string(argv[i], e);
      scheme_display(v, curout);
      scheme_display(scheme_make_character('\n'), curout);
    }
  }
  return 0;
}
```

Under Windows, the garbage collector finds static variables in an embeddeding program by examining all memory pages. This strategy fails if a program contains multiple Windows threads; a page may get unmapped by a thread while the collector is examining the page, causing the collector to crash. To avoid this problem, set `GC_use_registered_statics` to 1 before calling any `scheme_` function, and register all globals with `GC_use_registered_statics` to 1 before calling any GC or MzScheme function, and register all globals with `scheme_register_static`.

1.3 MzScheme and Threads

In its normal configuration, MzScheme implements threads for Scheme programs without aid from the operating system. On a few platforms, including Windows, Solaris, and Linux, MzScheme can be compiled to map each Scheme thread to a separate operating system thread.[2] The advantage of the OS-thread configuration is that different Scheme threads can take advantage of different processors on a multi-processor machine.

In either configuration, MzScheme can co-exist with additional OS threads that are created by an extension or an embedding program. However, the additional OS threads must not call any `scheme_` function. In the normal configuration, only the OS thread that originally calls `scheme_basic_env` can call `scheme_` functions.[3] In the OS-thread configuration, only the `scheme_basic_env` thread and other OS threads created by MzScheme (via the `thread` Scheme function or the `scheme_thread` C function) should call or `scheme_` functions.

In the normal configuration, when `scheme_basic_env` is called a second time to reset the interpreter, it can be called in an OS thread that is different from the original call to `scheme_basic_env`. Thereafter, all calls to `scheme_` functions must originate from the new thread.

See §2.7 for more information about threads, including the possible effects of MzScheme's thread implementation on extension and embedding C code.

[2]MrEd requires the normal configuration on all platforms.

[3]This restriction is stronger than saying all calls must be serialized across threads. MzScheme relies on properties of specific threads to avoid stack overflow and garbage collection.

2. MzScheme Architecture

2.1 Scheme Values and Types

A Scheme value is represented by a pointer-size value. The low bit is a mark bit: a 1 in the low bit indicates an immediate integer, a 0 indicates a (word-aligned) pointer.

A pointer-based Scheme value references a structure that begins with a type tag. This type tag has the C type `Scheme_Type`. The rest of the structure (following the type tag) is type-dependent. Examples of `Scheme_Type` values include `scheme_pair_type`, `scheme_symbol_type`, and `scheme_compiled_closure_type`.

MzScheme's C interface gives Scheme values the type `Scheme_Object *`. (The "object" here *does not* refer to objects in the sense of object-oriented programming.) The `struct` type `Scheme_Object` is defined in **scheme.h**, but never access this structure directly. Instead, use macros (such as `SCHEME_CAR`) that provide access to the data of common Scheme types. A `Scheme_Object` structure is actually only allocated for certain types (a few built-in types that contain two words of data in addition to the type tag), but `Scheme_Object *` is nevertheless used as the type of a generic Scheme value (for historical reasons).

For all standard Scheme types, constructors are provided for creating Scheme values. For example, `scheme_make_pair` takes two `Scheme_Object *` values and returns the `cons` of the values.

The macro `SCHEME_TYPE` takes a `Scheme_Object *` and returns the type of the object. This macro performs the tag-bit check, and returns `scheme_integer_type` when the value is an immediate integer; otherwise, `SCHEME_TYPE` follows the pointer to get the type tag. Macros are provided to test for common Scheme types; for example, `SCHEME_PAIRP` returns `1` if the value is a Scheme cons cell, `0` otherwise.

In addition to the standard Scheme data types, there are six global constant Scheme values: `scheme_true`, `scheme_false`, `scheme_null`, `scheme_eof`, `scheme_void`, and `scheme_undefined`. Each of these has a unique type tag, but they are normally recognized via their constant addresses rather than via their type tags.

An extension or application can create new a primitive data type by calling `scheme_make_type`, which returns a fresh `Scheme_Type` value. To create a collectable instance of this type, allocate memory for the instance with `scheme_malloc`. From MzScheme's perspective, the only constraint on the data format of such an instance is that the first `sizeof(Scheme_Type)` bytes must contain the value returned by `scheme_make_type`.

Scheme values should never be allocated on the stack, or contain pointers to values on the stack. Besides the problem of restricting the value's lifetime to that of the stack frame, allocating values on the stack creates problems for continuations and threads, both of which copy into and out of the stack.

2.1.1 Standard Types

The following are the `Scheme_Type` values for the standard types:

- `scheme_char_type` — `SCHEME_CHAR_VAL` extracts the character; test for this type with `SCHEME_CHARP`

- `scheme_integer_type` — fixnum integers, which are identified via the tag bit rather than following a pointer to this `Scheme_Type` value; `SCHEME_INT_VAL` extracts the integer; test for this type with `SCHEME_INTP`
- `scheme_double_type` — flonum inexact numbers; `SCHEME_FLOAT_VAL` or `SCHEME_DBL_VAL` extracts the floating-point value; test for this type with `SCHEME_DBLP`
- `scheme_float_type` — single-precision flonum inexact numbers, when specifically enabled when compiling MzScheme; `SCHEME_FLOAT_VAL` or `SCHEME_FLT_VAL` extracts the floating-point value; test for this type with `SCHEME_FLTP`
- `scheme_bignum_type` — test for this type with `SCHEME_BIGNUMP`
- `scheme_rational_type` — test for this type with `SCHEME_RATIONALP`
- `scheme_complex_type` — test for this type or `scheme_complex_izi_type` with `SCHEME_COMPLEXP`
- `scheme_complex_izi_type` — complex number with an inexact zero imaginary part (so it counts as a real number); test for this type specifically with `SCHEME_COMPLEX_IZIP`
- `scheme_string_type` — `SCHEME_STR_VAL` extracts the string (which is always null-terminated, but may also contain embedded nulls; the Scheme string is modified if this string is modified) and `SCHEME_STRLEN_VAL` extracts the string length (not counting the null terminator); test for this type with `SCHEME_STRINGP`
- `scheme_symbol_type` — `SCHEME_SYM_VAL` extracts the string (do not modify this string); test for this type with `SCHEME_SYMBOLP`
- `scheme_box_type` — `SCHEME_BOX_VAL` extracts/sets the boxed value; test for this type with `SCHEME_BOXP`
- `scheme_pair_type` — `SCHEME_CAR` extracts/sets the `car` and `SCHEME_CDR` extracts/sets the `cdr`; test for this type with `SCHEME_PAIRP`
- `scheme_vector_type` — `SCHEME_VEC_SIZE` extracts the length and `SCHEME_VEC_ELS` extracts the array of Scheme values (the Scheme vector is modified when this array is modified); test for this type with `SCHEME_VECTORP`
- `scheme_type_symbol_type` — `SCHEME_TSYM_VAL` extracts the symbol; test for this type with `SCHEME_TSYMBOLP`
- `scheme_object_type` — `SCHEME_OBJ_CLASS` extracts the class, `SCHEME_OBJ_DATA` extracts/sets the user pointer, and `SCHEME_OBJ_FLAG` extracts/sets the flag; test for this type with `SCHEME_OBJP`
- `scheme_class_type` — test for this type with `SCHEME_CLASSP`
- `scheme_interface_type` — test for this type with `SCHEME_INTERFACEP`
- `scheme_structure_type` — structure instances; test for this type with `SCHEME_STRUCTP`
- `scheme_struct_type_type` — structure types; test for this type with `SCHEME_STRUCT_TYPEP`
- `scheme_unit_type` — test for this type with `SCHEME_UNITP`
- `scheme_input_port_type` — `SCHEME_INPORT_VAL` extracts/sets the user data pointer; test for this type with `SCHEME_INPORTP`
- `scheme_output_port_type` — `SCHEME_OUTPORT_VAL` extracts/sets the user data pointer; test for this type with `SCHEME_OUTPORTP`
- `scheme_promise_type` — test for this type with `SCHEME_PROMP`

- `scheme_process_type` — thread descriptors; test for this type with `SCHEME_PROCESSP`
- `scheme_sema_type` — semaphores; test for this type with `SCHEME_SEMAP`
- `scheme_hash_table_type` — test for this type with `SCHEME_HASHTP`
- `scheme_weak_box_type` — test for this type with `SCHEME_WEAKP`; `SCHEME_WEAK_PTR` extracts the contained object, or `NULL` after the content is collected; do not set the content of a weak box
- `scheme_generic_data_type` — data analogous to a generic procedure created with `make-generic`; test for this type with `SCHEME_GENDATAP`
- `scheme_namespace_type` — namespaces; test for this type with `SCHEME_NAMESPACEP`
- `scheme_config_type` — parameterizations; test for this type with `SCHEME_CONFIGP`

The following are the procedure types:

- `scheme_prim_type` — a primitive procedure
- `scheme_closed_prim_type` — a primitive procedure with a data pointer
- `scheme_compiled_closure_type` — a Scheme procedure
- `scheme_cont_type` — a continuation
- `scheme_escaping_cont_type` — an escape continuation
- `scheme_case_closure_type` — a `case-lambda` procedure

The predicate `SCHEME_PROCP` returns 1 for all procedure types and 0 for anything else.

The following are additional number predicates:

- `SCHEME_NUMBERP` — all numerical types
- `SCHEME_REALP` — all non-complex numerical types, plus `scheme_complex_izi_type`
- `SCHEME_EXACT_INTEGERP` — fixnums and bignums
- `SCHEME_EXACT_REALP` — fixnums, bignums, and rationals
- `SCHEME_FLOATP` — both single-precision (when enabled) and double-precision flonums

2.1.2 Global Constants

There are six global constants:

- `scheme_null` — test for this value with `SCHEME_NULLP`
- `scheme_eof` — test for this value with `SCHEME_EOFP`
- `scheme_true`
- `scheme_false` — test for this value with `SCHEME_FALSEP`; test *against* it with `SCHEME_TRUEP`
- `scheme_void` — test for this value with `SCHEME_VOIDP`
- `scheme_undefined`

2.1.3 Library Functions

`Scheme_Object *scheme_make_char(char` *ch*`)`

Returns the character value.

`Scheme_Object *scheme_make_character(char` *ch*`)`

Returns the character value. (This is a macro.)

`Scheme_Object *scheme_make_integer(long` *i*`)`

Returns the integer value; *i* must fit in a fixnum. (This is a macro.)

`Scheme_Object *scheme_make_integer_value(long` *i*`)`

Returns the integer value. If *i* does not fit in a fixnum, a bignum is returned.

`Scheme_Object *scheme_make_integer_value_from_unsigned(unsigned long` *i*`)`

Like `scheme_make_integer_value`, but for unsigned integers.

`int scheme_get_int_val(Scheme_Object *`*o*, `long *`*i*`)`

Extracts the integer value. Unlike the `SCHEME_INT_VAL` macro, this procedure will extract an integer that fits in a `long` from a Scheme bignum. If *o* fits in a `long`, the extracted integer is placed in **i* and 1 is returned; otherwise, 0 is returned and **i* is unmodified.

`int scheme_get_unsigned_int_val(Scheme_Object *`*o*, `unsigned long *`*i*`)`

Like `scheme_get_int_val`, but for unsigned integers.

`Scheme_Object *scheme_make_double(double` *d*`)`

Creates a new floating-point value.

`Scheme_Object *scheme_make_float(float` *d*`)`

Creates a new single-precision floating-point value. The procedure is nly available when MzScheme is compiled with single-precision numbers enabled.

`double scheme_real_to_double(Scheme_Object *`*o*`)`

Converts a Scheme real number to a double-precision floating-point value.

`Scheme_Object *scheme_make_pair(Scheme_Object *`*carv*, `Scheme_Object *`*cdrv*`)`

Makes a `cons` pair.

`Scheme_Object *scheme_make_string(char *`*chars*`)`

Makes a Scheme string from a null-terminated C string. The *chars* string is copied.

`Scheme_Object *scheme_make_string_without_copying(char *`*chars*`)`

Like *scheme_make_string*, but the string is not copied.

`Scheme_Object *scheme_make_sized_string(char *`*chars*`, long `*len*`, int `*copy*`)`

Makes a string value with size *len*. A copy of *chars* is made if *copy* is not 0. The string *chars* should contain *len* characters; *chars* can contain the null character at any position, and need not be null-terminated. However, if *len* is negative, then the null-terminated length of *chars* is used for the length.

`Scheme_Object *scheme_make_sized_offset_string(char *`*chars*`, long `*d*`, long `*len*`, int `*copy*`)`

Like `scheme_make_sized_string`, except the *len* characters start from position *d* in *chars*.

`Scheme_Object *scheme_alloc_string(int `*size*`, char `*fill*`)`

Allocates a new Scheme string.

`Scheme_Object *scheme_append_string(Scheme_Object *`*a*`, Scheme_Object *`*b*`)`

Creates a new string by appending the two given strings.

`Scheme_Object *scheme_intern_symbol(char *`*name*`)`

Finds (or creates) the symbol matching the given null-terminated string. The case of *name* is (non-destructively) normalized before interning if `scheme_case_sensitive` is 0.

`Scheme_Object *scheme_intern_exact_symbol(char *`*name*`, int `*len*`)`

Creates or finds a symbol given the symbol's length. The the case of *name* is not normalized.

`Scheme_Object *scheme_make_symbol(char *`*name*`)`

Creates an uninterned symbol from a null-terminated string.

`Scheme_Object *scheme_make_exact_symbol(char *`*name*`, int `*len*`)`

Creates an uninterned symbol given the symbol's length.

`Scheme_Object *scheme_intern_type_symbol(Scheme_Object *`*sym*`)`

Creates or finds a type symbol from a symbolic name.

`Scheme_Object *scheme_make_type_symbol(Scheme_Object *`*sym*`)`

Creates an uninterned type symbol.

`Scheme_Object *scheme_make_vector(int `*size*`, Scheme_Object *`*fill*`)`

Allocates a new vector.

`Scheme_Object *scheme_make_promise(Scheme_Object *`*expr*`, Scheme_Env *`*env*`)`

Creates a promise that can be evaluated with the Scheme function `force`. The *expr* argument is an uncompiled S-expression.

`Scheme_Object *scheme_box(Scheme_Object *`*v*`)`

Creates a new box containing the value *v*.

`Scheme_Object *scheme_make_weak_box(Scheme_Object *`*v*`)`

Creates a new weak box containing the value *v*.

`Scheme_Type scheme_make_type(char *`*name*`)`

Creates a new type (not a Scheme value).

2.2 Memory Allocation

MzScheme uses both `malloc` and allocation functions provided the conservative garbage collector. Embedding/extension C/C++ code may use either allocation method, keeping in mind that pointers to garbage-collectable blocks in `malloc`ed memory are invisible (i.e., such pointers will not prevent the block from being garbage-collected).

The garbage collector normally only recognizes pointers to the beginning of allocated objects. Thus, a pointer into the middle of a GC-allocated string will normally not keep the string from being collected. The exception to this rule is that pointers saved on the stack or in registers may point to the middle of a collectable object. Thus, it is safe to loop over an array by incrementing a local pointer variable.

The collector allocation functions are:

- `scheme_malloc` — Allocates collectable memory that may contain pointers to collectable objects.
- `scheme_malloc_atomic` — Allocates collectable memory that does not contain pointers to collectable objects. If the memory does contain pointers, they are invisible to the collector and will not prevent an object from being collected.

  Atomic memory is used for strings or other blocks of memory which do not contain pointers. Atomic memory can also be used to store intentionally-hidden pointers.
- `scheme_malloc_stubborn` — Allocates collectable memory that may contain pointers to collectable objects, but also has special properties to support generational collection. Once the content of the allocated memory is set, call `scheme_end_stubborn_change`; this function call serves as a promise that the memory's contents will never be changed again (until after it is garbage-collected).
- `scheme_malloc_uncollectable` — Allocates uncollectable memory that may contain pointers to collectable objects. There is no way to free the memory.

If a MzScheme extension stores Scheme pointers in a global variable, then that variable must be registered with `scheme_register_extension_global`; this makes the pointer visible to the garbage collector. Registered variables need not contain a collectable pointer at all times. No registration is needed for the global variables of an embedding program.

Collectable memory can be temporarily locked from collection by using the reference-counting function `scheme_dont_gc_ptr`.

2.2.1 Library Functions

`void *scheme_malloc(size_t `*n*`)`

Allocates n bytes of collectable memory.

`void *scheme_malloc_atomic(size_t` *n*`)`

Allocates n bytes of collectable memory containing no pointers visible to the garbage collector.

`void *scheme_malloc_stubborn(size_t` *n*`)`

Allocates n bytes of collectable memory that is intended for use with `scheme_end_stubborn_change`.

`void *scheme_malloc_uncollectable(size_t` *n*`)`

Allocates n bytes of uncollectable memory.

`void *scheme_malloc_eternal(size_t` *n*`)`

Allocates uncollectable atomic memory. This function is equivalent to `malloc` except that it the memory cannot be freed.

`void scheme_end_stubborn_change(void *`*p*`)`

Promises that the contents of p will never be changed.

`void *scheme_calloc(size_t` *num*, `size_t` *size*`)`

Allocates *num* * *size* bytes of memory.

`char *scheme_strdup(char *`*str*`)`

Copies the null-terminated string *str*; the copy is collectable.

`char *scheme_strdup_eternal(char *`*str*`)`

Copies the null-terminated string *str*; the copy will never be freed.

`void *scheme_malloc_fail_ok(void *(*`*mallocf*`)(size_t` *size*`), size_t` *size*`)`

Attempts to allocate *size* bytes using *mallocf*. If the allocation fails, the `exn:misc:out-of-memory` exception is raised.

`void scheme_register_extension_global(void *`*ptr*`, long` *size*`)`

Registers an extension's global variable that can contain Scheme pointers. The address of the global is given in *ptr*, and its size in bytes in *size*. This function can actually be used to register any permanent memory that the collector would otherwise treat as atomic.

`void scheme_register_static(void *`*ptr*`, long` *size*`)`

Like `scheme_register_extension_global`, but for use only on static variables, and for use in embedding applications in situations where the collector does not automatically find static variables.

The macro `MZ_REGISTER_STATIC` can be used directly on a static variable. It expands to a comment if statics need not be registered, and a call to `scheme_register_static` (with the address of the static variable) otherwise.

`void scheme_weak_reference(void **`*p*`)`

Registers the pointer **p* as a weak pointer; when no other (non-weak) pointers reference the same memory as **p* references, then **p* will be set to `NULL` by the garbage collector. The value in **p* may change, but the pointer remains weak with respect to the value of **p* at the time *p* was registered.

`void scheme_weak_reference_indirect(void **`*p*`, void *`*v*`)`

Like `scheme_weak_reference`, but **p* is cleared (regardless of its value) when there are no references to *v*.

`void scheme_register_finalizer(void *`*p*`, void` *(*f)(void *p, void *data)*`, void *`*data*`,`
`void` *(**oldf)(void *p, void *data)*`, void **`*olddata*`)`

Registers a callback function to be invoked when the memory *p* would otherwise be garbage-collected. The *f* argument is the callback function; when it is called, it will be passed the value *p* and the data pointer *data*; *data* can be anything — it is only passed on to the callback function. If *oldf* and *olddata* are not `NULL`, then **oldf* and **olddata* are filled with with old callback information (*f* and *data* will override ths old callback).

Note: registering a callback not only keeps *p* from collection until the callback is invoked, but it also keeps *data* from collection.

`void scheme_add_finalizer(void *`*p*`, void` *(*f)(void *p, void *data)*`, void *`*data*`)`

Adds a finalizer to a chain of primitive finalizers. This chain is separate from the single finalizer installed with `scheme_register_finalizer`; all finalizers in the chain are called immediately after a finalizer that is installed with `scheme_register_finalizer`.

See `scheme_register_finalizer`, above, for information about the arguments.

`void scheme_add_scheme_finalizer(void *`*p*`, void` *(*f)(void *p, void *data)*`, void *`*data*`)`

Installs a "will"-like finalizer, similar to `will-register`. Scheme finalizers are called one at a time, requiring the collector to prove that a value has become inaccesibile again before calling the next Scheme finalizer.

See `scheme_register_finalizer`, above, for information about the arguments.

`void scheme_dont_gc_ptr(void *`*p*`)`

Keeps the collectable block *p* from garbage collection. Use this procedure when a reference to *p* is be stored somewhere inaccessible to the collector. Once the reference is no longer used from the inaccessible region, de-register the lock with `scheme_gc_ptr_ok`.

This function keeps a reference count on the pointers it registers, so two calls to `scheme_dont_gc_ptr` for the same *p* should be balanced with two calls to `scheme_gc_ptr_ok`.

`void scheme_gc_ptr_ok(void *`*p*`)`

See `scheme_dont_gc_ptr`.

`void scheme_collect_garbage()`

Forces an immediate garbage-collection.

2.3 Scheme Namespaces (Top-Level Environments)

A Scheme namespace (a top-level environment) is represented by a value of type `Scheme_Env *` (although it is also a Scheme value). Calling `scheme_basic_env` returns a namespace that includes all of MzScheme's standard global procedures and syntax.

The `scheme_basic_env` function must be called once by an embedding program, before any other MzScheme function is called (except `scheme_make_param`). The returned namespace is the initial current namespace for the main MzScheme thread. MzScheme extensions cannot call `scheme_basic_env`.

The current thread's current namespace is available from `scheme_get_env`, given the current parameterization (see §2.8): `scheme_get_env(scheme_config)`.

New values can be added as globals in a namespace using `scheme_add_global`. The `scheme_lookup_global` function takes a Scheme symbol and returns the global value for that name, or `NULL` if the symbol is undefined.

2.3.1 Library Functions

`void scheme_add_global(char *`*name*`, Scheme_Object *`*val*`, Scheme_Env *`*env*`)`

Adds a value to the table of globals for the namespace *env*, where *name* is a null-terminated string. (The string's case will be normalized in the same way as for interning a symbol.)

`void scheme_add_global_symbol(Scheme_Object *`*name*`, Scheme_Object *`*val*`, Scheme_Env *`*env*`)`

Adds a value to the table of globals by symbol name instead of string name.

`void scheme_add_global_constant(char *`*name*`, Scheme_Object *`*v*`, Scheme_Env *`*env*`)`

Like `scheme_add_global`, but the global variable name is also made constant if built-in constants are enabled, and `#%`*name* is also defined as a constant.

`void scheme_add_global_keyword(char *`*name*`, Scheme_Object *`*v*`, Scheme_Env *`*env*`)`

Like `scheme_add_global`, but the global variable name is also made constant and a keyword (unless keywords are disabled).

`void scheme_remove_global(char *`*name*`, Scheme_Env *`*env*`)`

Removes the variable binding from the table of globals for the namespace *env*. Constant globals cannot be removed.

`void scheme_remove_global_symbol(Scheme_Object *`*name*`, Scheme_Env *`*env*`)`

Removes a variable binding from the table of globals by symbol instead of by name.

`void scheme_remove_global_constant(char *`*name*`, Scheme_Env *`*env*`)`

Undefines *name* and also `#%`*name*. Both are undefined despite their potential constantness.

`void scheme_constant(Scheme_Object *`*sym*`, Scheme_Env *`*env*`)`

Declares the given global variable name (given as a symbol) to be constant in the table of globals for the namespace *env*.

`void scheme_set_keyword(Scheme_Object *`*sym*`, Scheme_Env *`*env*`)`

Declares the given symbol to be a keyword in the namespace *env*.

`Scheme_Object *scheme_lookup_global(Scheme_Object *`*symbol*`, Scheme_Env *`*env*`)`

Given a global variable name (as a symbol) in *sym*, returns the current value.

`Scheme_Bucket *scheme_global_bucket(Scheme_Object *`*symbol*`, Scheme_Env *`*env*`)`

Given a global variable name (as a symbol) in *sym*, returns the bucket where the value is stored. When the value in this bucket is `NULL`, then the global variable is undefined.

The `Scheme_Bucket` structure is defined as:

```
typedef struct Scheme_Bucket {
  Scheme_Type type; /* = scheme_variable_type */
  void *key;
  void *val;
} Scheme_Bucket;
```

`void scheme_set_global_bucket(char *`*procname*`, Scheme_Bucket *`*var*`, Scheme_Object *`*val*`,`
`int` *set_undef*`)`

Changes the value of a global variable. The *procname* argument is used to report errors (in case the global variable is constant, not yet bound, or a keyword). If *set_undef* is not 1, then the global variable must already have a binding. (For example, `set!` cannot set unbound variables, while `define` can.)

`Scheme_Env *scheme_get_env(Scheme_Config *`*config*`)`

Returns the current namespace for the given parameterization. See §2.8 for more information. The current thread's current parameterization is available as `scheme_config`.

2.4 Procedures

A **primitive procedure** is a Scheme-callable procedure that is implemented in C. Primitive procedures are created in MzScheme with the function `scheme_make_prim_w_arity`, which takes a C function pointer, the name of the primitive, and information about the number of Scheme arguments that it takes; it returns a Scheme procedure value.

The C function implementing the procedure must take two arguments: an integer that specifies the number of arguments passed to the procedure, and an array of `Scheme_Object *` arguments. The number of arguments passed to the function will be checked using the arity information. (The arity information provided to `scheme_make_prim_w_arity` is also used for the Scheme `arity` procedure.) The procedure implementation is not allowed to mutate the input array of arguments, although it may mutate the arguments themselves when appropriate (e.g., a fill in a vector argument).

The function `scheme_make_closed_prim_w_arity` is similar to `scheme_make_prim_w_arity`, but it takes an additional `void *` argument; this argument is passed back to the C function when the closure is invoked. In this way, closure-like data from the C world can be associated with the primitive procedure.

2.4.1 Library Functions

`Scheme_Object *scheme_make_prim_w_arity(Scheme_Prim *`*prim*`, char *`*name*`,`
`short` *mina*`, short` *maxa*`)`

Creates a primitive procedure value, given the C function pointer *prim*. The form of *prim* is defined by:

```
typedef Scheme_Object *(*Scheme_Prim)(int argc, Scheme_Object **argv);
```

The value *mina* should be the minimum number of arguments that must be supplied to the procedure. The value *maxa* should be the maximum number of arguments that can be suplied to the procedure, or -1 if the procedure can take arbitrarily many arguments. The *mina* and *maxa* values are used for automatically checking the argument count before the primitive is invoked, and also for the Scheme `arity` procedure. The *name* argument is used to report application arity errors at run-time.

`Scheme_Object *scheme_make_folding_prim(Scheme_Prim *`*prim*`, char *`*name*`,`
`short` *mina*`, short` *maxa*`, short` *folding*`)`

Like `scheme_make_prim_w_arity`, but if *folding* is non-zero, the compiler assumes that an application of the procedure to constant values can be folded to a constant. For example, `+`, `zero?`, and `string-length` are folding primitives, but `display`, `cons`, and `string-ref` are not. (Constant strings are currently mutable in MzScheme.)

`Scheme_Object *scheme_make_prim(Scheme_Prim *`*prim*`)`

Same as `scheme_make_prim_w_arity`, but the arity (0, -1) and the name "UNKNOWN" is assumed. This function is provided for backward compatibility only.

`Scheme_Object *scheme_make_noneternal_prim_w_arity(Scheme_Prim *`*prim*`,`
`char *`*name*`, short` *mina*`, short` *maxa*`)`

Same as `scheme_make_prim_w_arity`. This function is provided for backward compatibility only.

`Scheme_Object *scheme_make_noneternal_prim(Scheme_Prim *`*prim*`)`

Same as `scheme_make_prim`. This function is provided for backward compatibility only.

`Scheme_Object *scheme_make_closed_prim_w_arity(Scheme_Closed_Prim *`*prim*`, void *`*data*`,`
`char *`*name*`, short` *mina*`, short` *maxa*`)`

Creates a primitive procedure value; when the C function *prim* is invoked, *data* is passed as the first parameter. The form of *prim* is defined by:

```
typedef Scheme_Object *(*Scheme_Closed_Prim)(void *data, int argc, Scheme_Object **argv);
```

`Scheme_Object *scheme_make_closed_prim(Scheme_Closed_Prim *`*prim*`, void *`*data*`)`

Creates a closed primitive procedure value. This function is provided for backward compatibility only.

2.5 Evaluation

A Scheme S-expression is evaluated by calling `scheme_eval`. This function takes an S-expression (as a `Scheme_Object *`) and a namespace and returns the value of the expression in that namespace.

The function `scheme_apply` takes a `Scheme_Object *` that is a procedure, the number of arguments to pass to the procedure, and an array of `Scheme_Object *` arguments. The return value is the result of the application. There is also a function `scheme_apply_to_list`, which takes a procedure and a list (constructed with `scheme_make_pair`) and performs the Scheme `apply` operation.

The `scheme_eval` function actually calls `scheme_compile` followed by `scheme_eval_compiled`.

2.5.1 Top-level Evaluation Functions

The functions `scheme_eval`, `scheme_apply`, etc., are **top-level evaluation functions**. Continuation invocations are confined to jumps within a top-level evaluation.

The functions `_scheme_eval_compiled`, `_scheme_apply`, etc. provide the same functionality without starting a new top-level evaluation; these functions should only be used within new primitive procedures. Since these functions allow full continuation hops, calls to non-top-level evaluation functions can return zero or multiple times.

Currently, escape continuations and primitive error escapes can jump out of all evaluation and application functions. For more information, see §2.6.

2.5.2 Tail Evaluation

All of MzScheme's built-in functions and syntax support proper tail-recursion. When a new primitive procedure or syntax is added to MzScheme, special care must be taken to ensure that tail recursion is handled properly. Specifically, when the final return value of a function is the result of an application, then `scheme_tail_apply` should be used instead of `scheme_apply`. When `scheme_tail_apply` is called, it postpones the procedure application until control returns to the Scheme evaluation loop.

For example, consider the following implementation of a `thunk-or` primitive, which takes any number of thunks and performs `or` on the results of the thunks, evaluating only as many thunks as necessary.

```
static Scheme_Object *
thunk_or (int argc, Scheme_Object **argv)
{
  int i;
  Scheme_Object *v;

  if (!argc)
    return scheme_false;

  for (i = 0; i < argc - 1; i++)
    if (SCHEME_FALSEP((v = _scheme_apply(argv[i], 0, NULL))))
      return v;

  return scheme_tail_apply(argv[argc - 1], 0, NULL);
}
```

This `thunk-or` properly implements tail-recursion: if the final thunk is applied, then the result of `thunk-or` is the result of that application, so `scheme_tail_apply` is used for the final application.

2.5.3 Multiple Values

A primitive procedure can return multiple values by returning the result of calling `scheme_values`. The functions `scheme_eval_compiled_multi`, `scheme_apply_multi`, `_scheme_eval_compiled_multi`, and `_scheme_apply_multi` potentially return multiple values; all other evaluation and applications procedures return a single value or raise an exception.

Multiple return values are repsented by the `scheme_multiple_values` "value". This quasi-value has the type `Scheme_Object *`, but it is not a pointer or a fixnum. When the result of an evaluation or application is `scheme_multiple_values`, the number of actual values can be obtained as `scheme_multiple_count` and the array of `Scheme_Object *` values as `scheme_multiple_array`. If any application or evaluation procedure is called, the `scheme_multiple_count` and `scheme_multiple_array` variables may be modified, but the array previously referenced by `scheme_multiple_array` is never re-used and should never be modified.

The `scheme_multiple_count` and `scheme_multiple_array` variables only contain meaningful values when `scheme_multiple_values` is returned.

2.5.4 Library Functions

`Scheme_Object *scheme_eval(Scheme_Object *`*expr*`, Scheme_Env *`*env*`)`

Evaluates the (uncompiled) S-expression *expr* in the namespace *env*.

`Scheme_Object *scheme_eval_compiled(Scheme_Object *`*obj*`)`

Evaluates the compiled expression *obj*, which was previously returned from `scheme_compile`.

`Scheme_Object *scheme_eval_compiled_multi(Scheme_Object *`*obj*`)`

Evaluates the compiled expression *obj*, possibly returning multiple values (see §2.5.3).

`Scheme_Object *_scheme_eval_compiled(Scheme_Object *`*obj*`)`

Non-top-level version of `scheme_eval_compiled`. (See §2.5.1.)

`Scheme_Object *_scheme_eval_compiled_multi(Scheme_Object *`*obj*`)`

Non-top-level version of `scheme_eval_compiled_multi`. (See §2.5.1.)

`Scheme_Env *scheme_basic_env()`

Creates the main namespace for an embedded MzScheme. This procedure must be called before other MzScheme library function (except `scheme_make_param`). Extensions to MzScheme cannot call this function.

If it is called more than once, this function resets all threads (replacing the main thread), parameters, ports, namespaces, and finalizations.

`Scheme_Object *scheme_make_namespace(int `*argc*`, Scheme_Object **`*argv*`)`

Creates and returns a new namespace. This values can be cast to `Scheme_Env *`. It can also be installed in a parameterization using `scheme_set_param` with `MZCONFIG_ENV`.

When MzScheme is embedded in an application, create the initial namespace with `scheme_basic_env` before calling this procedure to create new namespaces.

`Scheme_Object *scheme_apply(Scheme_Object *`*f*`, int `*c*`, Scheme_Object **`*args*`)`

Applies the procedure *f* to the given arguments.

`Scheme_Object *scheme_apply_multi(Scheme_Object *`*f*`, int` *c*`, Scheme_Object **`*args*`)`

Applies the procedure *f* to the given arguments, possibly returning multiple values (see §2.5.3).

`Scheme_Object *_scheme_apply(Scheme_Object *`*f*`, int` *c*`, Scheme_Object **`*args*`)`

Non-top-level version of `scheme_apply`. (See §2.5.1.)

`Scheme_Object *_scheme_apply_multi(Scheme_Object *`*f*`, int` *c*`, Scheme_Object **`*args*`)`

Non-top-level version of `scheme_apply_multi`. (See §2.5.1.)

`Scheme_Object *scheme_apply_to_list(Scheme_Object *`*f*`, Scheme_Object *`*args*`)`

Applies the procedure *f* to the list of arguments in *args*.

`Scheme_Object *scheme_eval_string(char *`*str*`, Scheme_Env *`*env*`)`

Reads an S-expression from *str* and evaluates it in the given namespace (raising an exception if the expression returns multiple values).

`Scheme_Object *scheme_eval_string_multi(char *`*str*`, Scheme_Env *`*env*`)`

Like `scheme_eval_string`, but returns `scheme_multiple_values` when the expression returns multiple values.

`Scheme_Object *scheme_eval_string_all(char *`*str*`, Scheme_Env *`*env*`, int` *all*`)`

Like `scheme_eval_string`, but if *all* is not `0`, then expressions are read and evaluated from *str* until the end of the string is reached.

`Scheme_Object *scheme_tail_apply(Scheme_Object *`*f*`, int` *n*`, Scheme_Object **`*args*`)`

Applies the procedure as a tail-call. Actually, this function just registers the given application to be invoked when control returns to the evaluation loop. (Hence, this function is only useful within a primitive procedure that is returning to its calle.)

`Scheme_Object *scheme_tail_apply_no_copy(Scheme_Object *`*f*`, int` *n*`, Scheme_Object **`*args*`)`

Like `scheme_tail_apply`, but the array *args* is not copied. Use this only when *args* has infinite extent and will not be used again, or when *args* will certainly not be used again until the called procedure has returned.

`Scheme_Object *scheme_tail_apply_to_list(Scheme_Object *`*f*`, Scheme_Object *`*l*`)`

Applies the procedure as a tail-call.

`Scheme_Object *scheme_compile(Scheme_Object *`*form*`, Scheme_Env *`*env*`)`

Compiles the S-expression *form* in the given namespace. The returned value can be used with `scheme_eval_compiled` et al.

`Scheme_Object *scheme_expand(Scheme_Object *`*form*`, Scheme_Env *`*env*`)`

Expands all macros in the S-expression *form* using the given namespace.

`Scheme_Object *scheme_values(int` *n*, `Scheme_Object **`*args*`)`

Returns the given values together as multiple return values. Unless *n* is `1`, the result will always be `scheme_multiple_values`.

`void scheme_rep()`

Executes a read-eval-print loop, reading from the current input port and writing to the current output port. The current thread's namespace is used for evaluation.

2.6 Exceptions and Escape Continuations

When MzScheme encounters an error, it raises an exception. The default exception handler invokes the error display handler and then the error escape handler. The default error escape handler escapes via a **primitive error escape**, which is implemented by calling `scheme_longjmp(scheme_error_buf)`. An embedding program should call `scheme_setjmp(scheme_error_buf)` before any top-level entry into MzScheme evaluation to catch primitive error escapes:

```
...
if (scheme_setjmp(scheme_error_buf)) {
  /* There was an error */
  ...
} else {
  v = scheme_eval_string(s, env);
}
...
```

New primitive procedures can raise a generic exception by calling `scheme_signal_error`. The arguments for `scheme_signal_error` are roughly the same as for the standard C function `printf`. A specific primitive exception can be raised by calling `scheme_raise_exn`.

Full continuations are implemented in MzScheme by copying the C stack and using `scheme_setjmp` and `scheme_longjmp`. As long a C/C++ application invokes MzScheme evaluation through the top-level evaluation functions (`scheme_eval`, `scheme_eval`, etc., as opposed to `_scheme_eval`, `_scheme_apply`, etc.), the code is protected against any unusual behavior from Scheme evaluations (such as returning twice from a function) because continuation invocations ae confined to jumps within a single top-level evaluation. However, escape continuation jumps are still allowed; as explained in the following sub-section, special care must be taken in extension that is sensitive to escapes.

2.6.1 Temporarily Catching Error Escapes

When implementing new primitive procedure, it is sometimes useful to catch and handle errors that occur in evaluating subexpressions. One way to do this is the following: first copy `scheme_error_buf` to a temporary variable, invoke `scheme_setjmp(scheme_error_buf)`, perform the function's work, and then restore `scheme_error_buf` before returning a value.

However, beware that the invocation of an escaping continuation looks like a primitive error escape, but the special indicator flag `scheme_jumping_to_continuation` is non-zero (instead of its normal zero value); this situation is only visible when implementing a new primitive procedure. Honor the escape request by chaining to the previously saved error buffer; otherwise, call `scheme_clear_escape`.

```
  mz_jmp_buf save;
  memcpy(&save, &scheme_error_buf, sizeof(mz_jmp_buf));
  if (scheme_setjmp(scheme_error_buf)) {
    /* There was an error or continuation invokcation */
    if (scheme_jumping_to_continuation) {
      /* It was a continuation jump */
      scheme_longjmp(save, 1);
      /* To block the jump, instead: scheme_clear_escape(); */
    } else {
      /* It was a primitive error escape */
    }
  } else {
    scheme_eval_string("x", scheme_env);
  }
  memcpy(&scheme_error_buf, &save, sizeof(mz_jmp_buf));
```

This solution works fine as long as the procedure implementation only calls top-level evaluation functions (`scheme_eval`, `scheme_eval`, etc., as opposed to `_scheme_eval`, `_scheme_apply`, etc.). Otherwise, use `scheme_dynamic_wind` to protect your code against full continuation jumps in the same way that `dynamic-wind` is used in Scheme.

The above solution simply traps the escape; it doesn't report the reason that the escape occurred. To catch exceptions and obtain information about the exception, the simplest route is to mix Scheme code with C-implemented thunks. The code below can be used to catch exceptions in a variety of situations. It implements the function `_apply_catch_exceptions`, which catches exceptions during the application of a thunk. (This code is in `plt/src/mzscheme/dynsrc/oe.c` in the source code distribution.)

```
static Scheme_Object *exn_catching_apply, *exn_p, *exn_message;

static void init_exn_catching_apply()
{
  if (!exn_catching_apply) {
    char *e =
      "(#%lambda (thunk) "
        "(#%with-handlers ([#%void (#%lambda (exn) (#%cons #f exn))]) "
          "(#%cons #t (thunk))))";
    /* make sure we have a namespace with the standard syntax: */
    Scheme_Env *env = (Scheme_Env *)scheme_make_namespace(0, NULL);

#if !SCHEME_DIRECT_EMBEDDED
    scheme_register_extension_global(&exn_catching_apply, sizeof(Scheme_Object *));
    scheme_register_extension_global(&exn_p, sizeof(Scheme_Object *));
    scheme_register_extension_global(&exn_message, sizeof(Scheme_Object *));
#endif

    exn_catching_apply = scheme_eval_string(e, env);
    exn_p = scheme_lookup_global(scheme_intern_symbol("exn?"), env);
    exn_message = scheme_lookup_global(scheme_intern_symbol("exn-message"), env);
  }
}

/* This function applies a thunk, returning the Scheme value if there's no exception,
```

```
   otherwise returning NULL and setting *exn to the raised value (usually an exn
   structure). */
Scheme_Object *_apply_thunk_catch_exceptions(Scheme_Object *f, Scheme_Object **exn)
{
  Scheme_Object *v;

  init_exn_catching_apply();

  v = _scheme_apply(exn_catching_apply, 1, &f);
  /* v is a pair: (cons #t value) or (cons #f exn) */

  if (SCHEME_TRUEP(SCHEME_CAR(v)))
    return SCHEME_CDR(v);
  else {
    *exn = SCHEME_CDR(v);
    return NULL;
  }
}

Scheme_Object *extract_exn_message(Scheme_Object *v)
{
  init_exn_catching_apply();

  if (SCHEME_TRUEP(_scheme_apply(exn_p, 1, &v)))
    return _scheme_apply(exn_message, 1, &v);
  else
    return NULL; /* Not an exn structure */
}
```

In the following example, the above code is used to catch exceptions that occur during while evaluating source code from a string.

```
static Scheme_Object *do_eval(void *s, int noargc, Scheme_Object **noargv)
{
  return scheme_eval_string((char *)s, scheme_get_env(scheme_config));
}

static Scheme_Object *eval_string_or_get_exn_message(char *s)
{
  Scheme_Object *v, *exn;

  v = _apply_thunk_catch_exceptions(scheme_make_closed_prim(do_eval, s), &exn);
  /* Got a value? */
  if (v)
    return v;

  v = extract_exn_message(exn);
  /* Got an exn? */
  if (v)
    return v;

  /* ‘raise’ was called on some arbitrary value */
  return exn;
```

}

2.6.2 Library Functions

`void scheme_signal_error(char *`*msg*`, ...)`

Raises a generic primitive exception. The parameters are roughly as for `printf`, but restricted to the following format directives:

- `%c` — a character
- `%d` — an integer
- `%ld` — a `long` integer
- `%f` — a floating-point `double`
- `%s` — a nul-terminated string
- `%S` — a MzScheme symbol (a `Scheme_Object*`)
- `%t` — a string with a `long` size (two arguments), possibly containing a non-terminating nul character, and possibly without a nul-terminator
- `%T` — a MzScheme string (a `Scheme_Object*`)
- `%q` — a string, truncated to 253 characters, with ellipses printed if the string is truncated
- `%Q` — a MzScheme string (a `Scheme_Object*`), truncated to 253 characters, with ellipses printed if the string is truncated
- `%V` — a MzScheme value (a `Scheme_Object*`), truncated according to the current error print width.
- `%e` — an `errno` value, to be printed as a text message.
- `%E` — a platform-specific value, to be printed as a text message.
- `%%` — a percent sign

The arguments following the format string must include no more than 10 strings, 10 MzScheme values, 10 integers, and 10 floating-point numbers. (This restriction simplifies the implementation with precise garbage collection.)

`void scheme_raise_exn(int` *exnid*`, ...)`

Raises a specific primitive exception. The *exnid* argument specifies the exception to be raised. If an instance of that exception has n fields, then the next $n-2$ arguments are values for those fields (skipping the `message` and `debug-info` fields). The remaining arguments start with an error string and proceed roughly as for `printf`; see `scheme_signal_error` above for more details.

Exception ids are `#define`d using the same names as in Scheme, but prefixed with "MZ", all letters are capitalized, and all ":'s', "-"s, and "/"s are replaced with underscores. For example, `MZEXN_I_O_FILESYSTEM_DIRECTORY` is the exception id for the bad directory pathname exception.

`void scheme_warning(char *`*msg*`, ...)`

Signals a warning. The parameters are roughly as for `printf`; see `scheme_signal_error` above for more details.

`void scheme_wrong_count(char *`*name*`, int `*minc*`, int `*maxc*`, int `*argc*`, Scheme_Object **`*argv*`)`

This function is automatically invoked when the wrong number of arguments are given to a primitive procedure. It signals that the wrong number of parameters was received and escapes (like `scheme_signal_error`). The *name* argument is the name of the procedure that was given the wrong number of arguments; *minc* is the minimum number of expected arguments; *maxc* is the maximum number of expected arguments, or -1 if there is no maximum; *argc* and *argv* contain all of the received arguments.

`void scheme_wrong_type(char *`*name*`, char *`*expected*`, int `*which*`, int `*argc*`, Scheme_Object **`*argv*`)`

Signals that an argument of the wrong type was received, and escapes (like `scheme_signal_error`). *name* is the name of the procedure that was given the wrong type of argument; *expected* is the name of the expected type; *which* is the offending argument in the *argv* array; *argc* and *argv* contain all of the received arguments. If the original *argc* and *argv* are not available, provide -1 for *which* and a pointer to the bad value in *argv*; *argc* is ignored in this case.

`void scheme_wrong_return_arity(char *`*name*`, int `*expected*`, int `*got*`, Scheme_Object **`*argv*`,`
`const char *`*detail*`, ...)`

Signals that the wrong number of values were returned to a multiple-values context. The *expected* argument indicates how many values were expected, *got* indicates the number received, and *argv* are the received values. The *detail* string can be `NULL` or it can contain a `printf`-style string (with additional arguments) to describe the context of the error; see `scheme_signal_error` above for more details about the `printf`-style string.

`void scheme_unbound_global(char *`*name*`)`

Signals an unbound-variable error, where *name* is the name of the variable.

`char *scheme_make_provided_string(Scheme_Object *`*o*`, int `*count*`, int *`*len*`)`

Converts a Scheme value into a string for the purposes of reporting an error message. The *count* argument specifies how many Scheme values total will appear in the error message (so the string for this value can be scaled appropriately). If *len* is not `NULL`, it is filled with the length of the returned string.

`char *scheme_make_args_string(char *`*s*`, int `*which*`, int `*argc*`, Scheme_Object **`*argv*`, long *`*len*`)`

Converts an array of Scheme values into a string, skipping the array element indicated by *which*. This function is used to specify the "other" arguments to a function when one argument is bad (thus giving the user more information about the state of the program when the error occurred). If *len* is not `NULL`, it is filled with the length of the returned string.

`void scheme_check_proc_arity(char *`*where*`, int `*a*`, int `*which*`, int `*argc*`, Scheme_Object **`*argv*`)`

Checks the *which*th argument in *argv* to make sure it is a procedure that can take *a* arguments. If there is an error, the *where*, *which*, *argc*, and *argv* arguments are passed on to `scheme_wrong_type`. As in `scheme_wrong_type`, *which* can be -1, in which case **argv* is checked.

```
Scheme_Object *scheme_dynamic_wind(
               void (*pre)(void *data),
               Scheme_Object *(*action)(void *data),
```

```
void (*post)(void *data),
Scheme_Object *(*jmp_handler)(void *data),
void *data)
```

Evaluates calls the function *action* to get a value for the `scheme_dynamic_wind` call. The functions *pre* and *post* are invoked when jumping into and out of *action*, repsectively.

The function *jmp_handler* is called when an error is signaled (or an escaping continuation is invoked) duirng the call to *action*; if *jmp_handler* returns `NULL`, then the error is passed on to the next error handler, otherwise the return value is used as the return value for the `scheme_dynamic_wind` call.

The pointer *data* can be anything; it is passed along in calls to *action*, *pre*, *post*, and *jmp_handler*.

`void scheme_clear_escape()`

Clears the "jumping to escape continuation" flag associated with a thread. Call this function when blocking escape continuation hops (see the first example in §2.6.1).

2.7 Threads

The intializer function `scheme_basic_env` creates the main Scheme thread; all other threads are created through calls to `scheme_thread`.

Information about each internal MzScheme thread is kept in a `Scheme_Process` structure. A pointer to the current thread's structure is available as `scheme_current_process`. A `Scheme_Process` structure includes the following fields:

- `error_buf` — This is the `mz_jmp_buf` value used to escape from errors. The `error_buf` value of the current thread is available as `scheme_error_buf`.
- `cjs.jumping_to_continuation` — This flag distinguishes escaping-continuation invocations from error escapes. The `cjs.jumping_to_continuation` value of the current thread is available as `scheme_jumping_to_continuation`.
- `config` — The thread's current parameterization. See also §2.8.
- `next` — The next thread in the linked list of threads; this is `NULL` for the main thread.

The list of all threads is kept in a linked list; `scheme_first_process` points to the first thread in the list. The last thread in the list is always the main thread.

2.7.1 Integration with Threads

MzScheme's threads can break external C code under two circumstances:

- *Pointers to stack-based values can be communicated between threads.* For example, if thread A stores a pointer to a stack-based variable in a global variable, if thread B uses the pointer in the global variable, it may point to data that is not currently on the stack.
- *C functions that can invoke MzScheme (and also be invoked by MzScheme) depend on strict function-call nesting.* For example, suppose a function F uses an internal stack, pushing items on to the stack on entry and popping the same items on exit. Suppose also that F invokes MzScheme to evaluate an expression. If the evaluate on this expression invoked F again in a new thread, but then returns to the first thread before completeing the second F, then F's internal stack will be corrupted.

If either of these circumstances occurs, MzScheme will probably crash.

2.7.2 Blocking the Current Thread

Embedding or extension code sometimes needs to block, but blocking should allow other MzScheme threads to execute. To allow other threads to run, block using `scheme_block_until`. This procedure takes two functions: a polling function that tests whether the blocking operation can be completed, and a prepare-to-sleep function that sets bits in `fd_sets` when MzScheme decides to sleep (because all MzScheme threads are blocked). Under Windows and BeOS, an "`fd_set`" can also accomodate OS-level semaphores or other handles via `scheme_add_fd_handle`.

Since the functions passed to `scheme_block_until` are called by the Scheme thread scheduler, they must never raise exceptions, call `scheme_apply`, or trigger the evaluation of Scheme code in any way. The `scheme_block_until` function itself may call the current exception handler, however, in reaction to a break (if breaks are enabled).

2.7.3 Threads in Embedded MzScheme with Event Loops

When MzScheme is embedded in an application with an event-based model (i.e., the execution of Scheme code in the main thread is repeatedly triggered by external events until the application exits) special hooks must be set to ensure that non-main threads execute correctly. For example, during the execution in the main thread, a new thread may be created; the new thread may still be running when the main thread returns to the event loop, and it may be arbitrarily long before the main thread continues from the event loop. Under such circumstances, the embedding program must explicitly allow MzScheme to execute the non-main threads; this can be done by periodically calling the function `scheme_check_threads`.

Thread-checking only needs to be performed when non-main threads exist (or when there are active callback triggers). The embedding application can set the global function pointer `scheme_notify_multithread` to a function that takes an integer parameter and returns `void`. This function is be called with 1 when thread-checking becomes necessary, and then with 0 when thread checking is no longer necessary. An embedding program can use this information to prevent unnecessary `scheme_check_threads` polling.

The below code illustrates how MrEd formerly set up `scheme_check_threads` polling using the wxWindows `wxTimer` class. (Any regular event-loop-based callback is appropriate.) The `scheme_notify_multithread` pointer is set to `MrEdInstallThreadTimer`. (MrEd no longer work this way, however.)

```
class MrEdThreadTimer : public wxTimer
{
 public:
  void Notify(void); /* callback when timer expires */
};

static int threads_go;
static MrEdThreadTimer *theThreadTimer;
#define THREAD_WAIT_TIME 40

void MrEdThreadTimer::Notify()
{
  if (threads_go)
    Start(THREAD_WAIT_TIME, TRUE);

  scheme_check_threads();
}
```

```
static void MrEdInstallThreadTimer(int on)
{
  if (!theThreadTimer)
    theThreadTimer = new MrEdThreadTimer;

  if (on)
    theThreadTimer->Start(THREAD_WAIT_TIME, TRUE);
  else
    theThreadTimer->Stop();

  threads_go = on;
  if (on)
    do_this_time = 1;
}
```

An alternate architecture, which MrEd now uses, is to send the main thread into a loop, which blocks until an event is ready to handle. MzScheme automatically takes care of running all threads, and it does so efficiently because the main thread blocks on a file descriptor, as explained in §2.7.2.

2.7.3.1 Callbacks for Blocked Threads

Scheme threads are sometimes blocked on file descriptors, such as an input file or the X event socket. Blocked non-main threads do not block the main thread, and therefore do not affect the event loop, so `scheme_check_threads` is sufficient to implement this case correctly. However, it is wasteful to poll these descriptors with `scheme_check_threads` when nothing else is happening in the application and when a lower-level poll on the file descriptors can be installed. If the global function pointer `scheme_wakeup_on_input` is set, then this case is handled more efficiently by turning off thread checking and issuing a "wakeup" request on the blocking file descriptors through `scheme_wakeup_on_input`. (The `scheme_wakeup_on_input` function is only used on platforms with file descriptions.)

A `scheme_wakeup_on_input` procedure takes a pointer to an array of three `fd_sets` (sortof[1]) and returns `void`. The `scheme_wakeup_on_input` does not sleep; it just sets up callbacks on the specified file descriptors. When input is ready on any of those file descriptors, the callbacks are be removed and `scheme_wake_up` is called.

For example, the X Windows version of MrEd formerly set `scheme_wakeup_on_input` to this `MrEdNeedWakeup`:

```
static XtInputId *scheme_cb_ids = NULL;
static int num_cbs;

static void MrEdNeedWakeup(void *fds)
{
  int limit, count, i, p;
  fd_set *rd, *wr, *ex;

  rd = (fd_set *)fds;
  wr = ((fd_set *)fds) + 1;
  ex = ((fd_set *)fds) + 2;

  limit = getdtablesize();
```

[1]To ensure maximum portability, use `MZ_FD_XXX` instead of `FD_XXX`.

```
  /* See if we need to do any work, really: */
  count = 0;
  for (i = 0; i < limit; i++) {
    if (MZ_FD_ISSET(i, rd))
      count++;
    if (MZ_FD_ISSET(i, wr))
      count++;
    if (MZ_FD_ISSET(i, ex))
      count++;
  }

  if (!count)
    return;

  /* Remove old callbacks: */
  if (scheme_cb_ids)
    for (i = 0; i < num_cbs; i++)
      notify_set_input_func((Notify_client)NULL, (Notify_func)NULL,
                            scheme_cb_ids[i]);

  num_cbs = count;
  scheme_cb_ids = new int[num_cbs];

  /* Install callbacks */
  p = 0;
  for (i = 0; i < limit; i++) {
    if (MZ_FD_ISSET(i, rd))
      scheme_cb_ids[p++] = XtAppAddInput(wxAPP_CONTEXT, i,
                                         (XtPointer *)XtInputReadMask,
                                         (XtInputCallbackProc)MrEdWakeUp, NULL);
    if (MZ_FD_ISSET(i, wr))
      scheme_cb_ids[p++] = XtAppAddInput(wxAPP_CONTEXT, i,
                                         (XtPointer *)XtInputWriteMask,
                                         (XtInputCallbackProc)MrEdWakeUp, NULL);
    if (MZ_FD_ISSET(i, ex))
      scheme_cb_ids[p++] = XtAppAddInput(wxAPP_CONTEXT, i,
                                         (XtPointer *)XtInputExceptMask,
                                         (XtInputCallbackProc)MrEdWakeUp,
                                         NULL);
  }
}

/* callback function when input/exception is detected: */
Bool MrEdWakeUp(XtPointer, int *, XtInputId *)
{
  int i;

  if (scheme_cb_ids) {
    /* Remove all callbacks: */
    for (i = 0; i < num_cbs; i++)
     XtRemoveInput(scheme_cb_ids[i]);

    scheme_cb_ids = NULL;
```

```
    /* ''wake up'' */
    scheme_wake_up();
  }

  return FALSE;
}
```

2.7.4 Sleeping by Embedded MzScheme

When all MzScheme threads are blocked, MzScheme must "sleep" for a certain number of seconds or until external input appears on some file descriptor. Generally, sleeping should block the main event loop of the entire application. However, the way in which sleeping is performed may depend on the embedding application. The global function pointer `scheme_sleep` can be set by an embedding application to implement a blocking sleep, although MzScheme implements this function for you.

A `scheme_sleep` function takes two arguments: a `float` and a `void *`. The latter is really points to an array of three "`fd_set`" records (one for read, one for write, and one for exceptions); these records are described further below. If the `float` argument is non-zero, then the `scheme_sleep` function blocks for the specified number of seconds, at most. The `scheme_sleep` function should block until there is input one of the file descriptors specified in the "`fd_set`," indefinitely if the `float` argument is zero.

The second argument to `scheme_sleep` is conceptually an array of three `fd_set` records, but always use `scheme_get_fdset` to get anything other than the zeroth element of this array, and manipulate each "`fd_set`" with `MZ_FD_XXX` instead of `FD_XXX`.

The following function `mzsleep` is an appropriate `scheme_sleep` function for most any Unix or Windows application. (This is approximately the built-in sleep used by MzScheme.)

```
void mzsleep(float v, void *fds)
{
  if (v) {
    sleep(v);
  } else {
    int limit;
    fd_set *rd, *wr, *ex;

# ifdef WIN32
    limit = 0;
# else
    limit = getdtablesize();
# endif

    rd = (fd_set *)fds;
    wr = (fd_set *)scheme_get_fdset(fds, 1);
    ex = (fd_set *)scheme_get_fdset(fds, 2);

    select(limit, rd, wr, ex, NULL);
  }
}
```

2.7.5 Library Functions

`Scheme_Object *scheme_thread(Scheme_Object *`*thunk*, `Scheme_Config *`*config*`)`

Creates a new thread, using the given parameterization for the new thread. If *config* is `NULL`, a new parameterization is created using the current thread's parameterization's current base parameterization. The new thread begins evaluating the application of the procedure *thunk* (with no arguments).

`Scheme_Object *scheme_make_sema(long` *v*`)`

Creates a new semaphore.

`void scheme_post_sema(Scheme_Object *`*sema*`)`

Posts to *sema*.

`int scheme_wait_sema(Scheme_Object *`*sema*, `int` *try*`)`

Waits on *sema*. If *try* is not 0, the wait can fail and 0 is returned for failure, otherwise 1 is returned.

`void scheme_process_block(float` *sleep_time*`)`

Allows the current thread to be swapped out in favor of other threads. If *sleep_time* positive, then the current thread will sleep for at least *sleep_time* seconds.

`void scheme_swap_process(Scheme_Process *`*process*`)`

Swaps out the current thread in favor of *process*.

`void scheme_break_thread(Scheme_Process *`*thread*`)`

Issues a user-break in the given thread.

`int scheme_break_waiting(Scheme_Process *`*thread*`)`

Returns `1` if a break from `break-thread` or `scheme_break_thread` has occured in the specified process but has not yet been handled.

`int scheme_block_until(int (*`*f*`)(Scheme_Object *`*data*`),`
`void (*`*fdf*`)(Scheme_Object *`*data*`, void *`*fds*`), void *`*data*`, float` *sleep*`)`

Blocks the current thread until *f* returns a true value. The *f* function is called periodically, and it may be called multiple times even after it returns a true value. (If *f* ever returns a true value, it must continue to return a true value.) The argument to *f* is the same *data* as provided to `scheme_block_until`, and *data* is ignored otherwise. (The type mismatch between `void*` and `Scheme_Object*` is an ugly artifact. The *data* argument is not intended to necessarily be a `Scheme_Object*` value.)

If MzScheme decides to sleep, then the *fdf* function is called to sets bits in *fds*, conceptually an array of three `fd_sets`: one or reading, one for writing, and one for exceptions. Use `scheme_get_fdset` to get elements of this array, and manipulate an "`fd_set`" with `MZ_FD_XXX` instead of `FD_XXX`. Under Windows and BeOS, an "`fd_set`" can also accomodate OS-level semaphores or other handles via `scheme_add_fd_handle`.

The *fdf* argument can be `NULL`, indicating that the thread becomes unblocked only through Scheme actions, and never through external processes (e.g., through a socket or OS-level semaphore).

If *sleep* is a positive number, then `scheme_block_until` polls *f* roughly every *sleep* seconds, but `scheme_block_until` does not return until *f* returns a true value.

The return value from `scheme_block_until` is the return value of its most recent call to *f*, which enables *f* to return some information to the `scheme_block_until` caller.

See §2.7.2 for information about restrictions on the *f* and *fdf* functions.

`void scheme_check_threads()`

This function is periodically called by the embedding program to give background processes time to execute. See §2.7.3 for more information.

`void scheme_wake_up()`

This function is called by the embedding program when there is input on an external file descriptor. See §2.7.4 for more information.

`void *scheme_get_fdset(void *`*fds*`)`

Extracts an "`fd_set`" from an array passed to `scheme_sleep`, a callback for `scheme_block_until`, or an input port callback for `scheme_make_input_port`.

`void scheme_add_fd_handle(void *`*h*, `void *`*fds*, `int` *repost*`)`

Adds an OS-level semaphore (Windows, BeOS) or other waitable handle (Windows) to the "`fd_set`" *fds*. When MzScheme performs a "`select`" to sleep on *fds*, it also waits on the given semaphore or handle. This feature makes it possible for MzScheme to sleep until it is awakened by an external process.

MzScheme does not attempt to deallocate the given semaphore or handle, and the "`select`" call using *fds* may be unblocked due to some other file descriptor or handle in *fds*. If *repost* is a true value, then *h* must be an OS-level semeaphore, and if the "`select`" unblocks due to a post on *h*, then *h* is reposted; this allows clients to treat *fds*-installed semaphores uniformly, whether or not a post on the semaphore was consumed by "`select`".

The `scheme_add_fd_handle` function is useful for implementing the second procedure passed to `scheme_wait_until`, or for implementing a custom input port.

Under Unix and MacOS, this function has no effect.

`void scheme_add_fd_eventmask(void *`*fds*, `int` *mask*`)`

Adds an OS-level event type (Windows) to the set of types in the "`fd_set`" *fds*. When MzScheme performs a "`select`" to sleep on *fds*, it also waits on events of them specified type. This feature makes it possible for MzScheme to sleep until it is awakened by an external process.

The event mask is only used when some handle is installed with `scheme_add_fd_handle`. This restriction is stupid, and it may force you to create a dummy semaphore that is never posted.

Under Unix, BeOS, and MacOS, this function has no effect.

`int scheme_tls_allocate()`

Allocates a thread local storage index to be used with `scheme_tls_set` and `scheme_tls_get`.

`void scheme_tls_set(int` *index*, `void *`*v*`)`

Stores a thread-specific value using an index allocated with `scheme_tls_allocate`.

`void *scheme_tls_get(int` *index*`)`

Retrieves a thread-specific value installed with `scheme_tls_set`. If no thread-specific value is available for the given index, `NULL` is returned.

2.8 Parameterizations

Parameterization information is stored in a `Scheme_Config` record. For the currently executing thread, `scheme_config` is the current parameterization. For any thread, the thread's `Scheme_Process` record's `config` field stores the parameterization pointer.

Parameter values for built-in parameters are obtained and modified using `scheme_get_param` and `scheme_set_param`. Each parameter is stored as a `Scheme_Object *` value, and the built-in parameters are accessed through the following indices:

- `MZCONFIG_ENV` — `current-namespace` (use `scheme_get_env`)
- `MZCONFIG_INPUT_PORT` — `current-input-port`
- `MZCONFIG_OUTPUT_PORT` — `current-output-port`
- `MZCONFIG_ERROR_PORT` — `current-error-port`
- `MZCONFIG_ENABLE_BREAK` — `break-enabled`
- `MZCONFIG_ENABLE_EXCEPTION_BREAK` — `exception-break-enabled`
- `MZCONFIG_ERROR_DISPLAY_HANDLER` — `error-display-handler`
- `MZCONFIG_ERROR_PRINT_VALUE_HANDLER` — `error-value->string-handler`
- `MZCONFIG_EXIT_HANDLER` — `exit-handler`
- `MZCONFIG_EXN_HANDLER` — `current-exception-handler`
- `MZCONFIG_DEBUG_INFO_HANDLER` — `debug-info-handler`
- `MZCONFIG_EVAL_HANDLER` — `current-eval`
- `MZCONFIG_LOAD_HANDLER` — `current-load`
- `MZCONFIG_PRINT_HANDLER` — `current-print`
- `MZCONFIG_PROMPT_READ_HANDLER` — `current-prompt-read`
- `MZCONFIG_CAN_READ_GRAPH` — `read-accept-graph`
- `MZCONFIG_CAN_READ_COMPILED` — `read-accept-compiled`
- `MZCONFIG_CAN_READ_BOX` — `read-accept-box`
- `MZCONFIG_CAN_READ_TYPE_SYMBOL` — `read-accept-type-symbol`
- `MZCONFIG_CAN_READ_PIPE_QUOTE` — `read-accept-bar-quote`
- `MZCONFIG_PRINT_GRAPH` — `print-graph`
- `MZCONFIG_PRINT_STRUCT` — `print-struct`
- `MZCONFIG_PRINT_BOX` — `print-box`
- `MZCONFIG_CASE_SENS` — `read-case-sensitive`
- `MZCONFIG_SQUARE_BRACKETS_ARE_PARENS` — `read-square-brackets-as-parens`
- `MZCONFIG_CURLY_BRACES_ARE_PARENS` — `read-curly-braces-as-parens`
- `MZCONFIG_ERROR_PRINT_WIDTH` — `error-print-width`
- `MZCONFIG_CONFIG_BRANCH_HANDLER` — `parameterization-branch-handler`
- `MZCONFIG_ALLOW_SET_UNDEFINED` — `allow-compile-set!-undefined`
- `MZCONFIG_ALLOW_COND_AUTO_ELSE` — `allow-compile-cond-fallthrough`
- `MZCONFIG_MANAGER` — `current-custodian`
- `MZCONFIG_REQ_LIB_USE_COMPILED` — `require-library-use-compiled`
- `MZCONFIG_LOAD_DIRECTORY` — `current-load-relative-directory`
- `MZCONFIG_COLLECTION_PATHS` — `current-library-collection-paths`
- `MZCONFIG_PORT_PRINT_HANDLER` — `global-port-print-handler`
- `MZCONFIG_REQUIRE_COLLECTION` — `current-require-relative-collection`
- `MZCONFIG_LOAD_EXTENSION_HANDLER` — `current-load-extension`

When installing a new parameter with `scheme_set_param`, no checking is performed on the supplied value to ensure that it is a legal value for the parameter; this is the responsibility of the caller of `scheme_set_param`. Note that Boolean parameters should only be set to the values `#t` and `#f`.

New primitive parameter indices are created with `scheme_new_param` and implemented with `scheme_make_parameter` and `scheme_param_config`.

2.8.1 Library Functions

`Scheme_Object *scheme_get_param(Scheme_Config *`*config*`, int `*param_id*`)`

Gets the current value of the parameter specified by *param_id*. (This is a macro.)

`Scheme_Object *scheme_get_param_or_null(Scheme_Config *`*config*`, int `*param_id*`)`

Gets the current value of the parameter specified by *param_id*. (This is a macro.)

`Scheme_Object *scheme_make_config(Scheme_Config *`*base*`)`

Creates and returns a new configuration, using *base* as the base configuration. If *base* is `NULL`, the current thread's parameterization's current base parameterization is used.

`int scheme_new_param()`

Allocates a new primitive parameter index. This function must be called *before* `scheme_basic_env`.

`Scheme_Object *scheme_make_parameter(Scheme_Prim *`*function*`, char *`*name*`)`

Use this function instead of the other primitive-constructing functions, like `scheme_make_prim`, to create a primitive parameter procedure. See also `scheme_param_config`, below.

`Scheme_Object *scheme_param_config(char *`*name*`, Scheme_Object *`*param*`, int `*argc*`, Scheme_Object **`*argv*`,`
`int `*arity*`, Scheme_Prim *`*check*`, char *`*expected*`, int `*isbool*`)`

Call this procedure in a primitive parameter procedure to implement the work of getting or setting the parameter. The *name* argument should be the parameter procedure name; it is used to report errors. The *param* argument is a fixnum corresponding to the primtive parameter index returned by `scheme_new_param`. The *argc* and *argv* arguments should be the un-touched and un-tested arguments that were passed to the primitive parameter. Argument-checking is performed within `scheme_param_config` using *arity*, *check*, *expected*, and *isbool*:

- If *arity* is non-negative, potential parameter values must be able to accept the specified number of arguments. The *check* and *expected* arguments should be `NULL`.
- If *check* is not `NULL`, it is called to check a potential parameter value. The arguments passed to *check* are always `1` and an array that contains the potential parameter value. If *isbool* is `0` and *check* returns `scheme_false`, then a type error is reported using *name* and *expected*. If *isbool* is `1`, then a type error is reported only when *check* returns `NULL` and any non-`NULL` return value is used as the actual value to be stored for the parameter.
- Otherwise, *isbool* should be 1. A potential procedure argument is then treated as a Boolean value.

2.9 Bignums, Rationals, and Complex Numbers

MzScheme supports integers of an arbitrary magnitude; when an integer cannot be represented as a fixnum (i.e., 30 or 62 bits plus a sign bit), then it is reprsented by the MzScheme type `scheme_bignum_type`. There is no overlap in integer values represented by fixnums and bignums.

Rationals are implemented by the type `scheme_rational_type`, composed of a numerator and a denominator. The numerator and denominator fixnums or bignums (possibly mixed).

Complex numbers are implemented by the types `scheme_complex_type` and `scheme_complex_izi_type`, composed of a real and imaginary part. The real and imaginary parts will either be both flonums, both exact numbers (fixnums, bignums, and rationals can be mixed in any way), or one part will be eacxt 0 and the other part will be a flonum. If the inexact part is inexact 0, the type is `scheme_complex_izi_type`, otherwise the type is `scheme_complex_type`; this distinction make it easy to test whether a complex number should be treated as a real number.

2.9.1 Library Functions

`int scheme_is_exact(Scheme_Object *`*n*`)`

Returns `1` if *n* is an exact number, `0` otherwise (*n* need not be a number).

`int scheme_is_inexact(Scheme_Object *`*n*`)`

Returns `1` if *n* is an inexact number, `0` otherwise (*n* need not be a number).

`Scheme_Object *scheme_make_bignum(long `*v*`)`

Creates a bignum representing the integer *v*. This can create a bignum that otherwise fits into a fixnum. This must only be used to create temporary values for use with the `bignum` functions. Final results can be normalized with `scheme_bignum_normalize`. Only normalized numbers can be used with procedures that are not specific to bignums.

`Scheme_Object *scheme_make_bignum_from_unsigned(unsigned long `*v*`)`

Like `scheme_make_bignum`, but works on unsigned integers.

`double scheme_bignum_to_double(Scheme_Object *`*n*`)`

Converts a bignum to a floating-point number, with reasonable but unspecified accuracy.

`float scheme_bignum_to_float(Scheme_Object *`*n*`)`

If MzScheme is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_bignum_to_double`.

`Scheme_Object *scheme_bignum_from_double(double `*d*`)`

Creates a bignum that is close in magnitude to the floating-point number *d*. The conversion accuracy is reasonable but unspecified.

`Scheme_Object *scheme_bignum_from_float(float `*f*`)`

If MzScheme is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_bignum_from_double`.

`char *scheme_bignum_to_string(Scheme_Object *`*n*`, int `*radix*`)`

Writes a bignum into a newly allocated string.

`Scheme_Object *scheme_read_bignum(char *`*str*`, int `*offset*`, int `*radix*`)`

Reads a bignum from a string, starting from position *offset* in *str*. If the string does not represent an integer, then `NULL` will be returned. If the string represents a number that fits in 31 bits, then a `scheme_integer_type` object will be returned.

`Scheme_Object *scheme_bignum_normalize(Scheme_Object *`*n*`)`

If *n* fits in 31 bits, then a `scheme_integer_type` object will be returned. Otherwise, *n* is returned.

`Scheme_Object *scheme_make_rational(Scheme_Object *`*r*`, Scheme_Object *`*d*`)`

Creates a rational from a numerator and denominator. The *n* and *d* parameters must be fixnums or bignums (possibly mixed). The resulting will be normalized (thus, an bignum or fixnum might be returned).

`double scheme_rational_to_double(Scheme_Object *`*n*`)`

Converts the rational *n* to a `double`.

`float scheme_rational_to_float(Scheme_Object *`*n*`)`

If MzScheme is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_rational_to_double`.

`Scheme_Object *scheme_rational_numerator(Scheme_Object *`*n*`)`

Returns the numerator of the rational *n*.

`Scheme_Object *scheme_rational_denominator(Scheme_Object *`*n*`)`

Returns the denominator of the rational *n*.

`Scheme_Object *scheme_rational_from_double(double `*d*`)`

Converts the given `double` into a maximally-precise rational.

`Scheme_Object *scheme_rational_from_float(float `*d*`)`

If MzScheme is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_rational_from_double`.

`Scheme_Object *scheme_make_complex(Scheme_Object *`*r*`, Scheme_Object *`*i*`)`

Creates a complex number from real and imaginary parts. The *r* and *i* arguments must be fixnums, bignums, flonums, or rationals (possibly mixed). The resulting number will be normalized (thus, a real number might be returned).

`Scheme_Object *scheme_complex_real_part(Scheme_Object *`*n*`)`

Returns the real part of the complex number *n*.

`Scheme_Object *scheme_complex_imaginary_part(Scheme_Object *`*n*`)`

Returns the imaginary part of the complex number *n*.

2.10 Ports and the Filesystem

Ports are represented as Scheme values with the types `scheme_input_port_type` and `scheme_output_port_type`. The function `scheme_read` takes an input port value and returns the next S-expression from the port. The function `scheme_write` takes an output port and a value and writes the value to the port. Other standard low-level port functions are also provided, such as `scheme_getc`.

File ports are created with `scheme_make_file_input_port` and `scheme_make_file_output_port`; these functions take a `FILE *` file pointer and return a Scheme port. Strings are read or written with `scheme_make_string_input_port`, which takes a null-terminated string, and `scheme_make_string_output_port`, which takes no arguments. The contents of a string output port are obtained with `scheme_get_string_output`.

Custom ports, with arbitrary read/write handlers, are created with `scheme_make_input_port` and `scheme_make_output_port`.

2.10.1 Library Functions

`Scheme_Object *scheme_read(Scheme_Object *`*port*`)`

Reads the next S-expression from the given input port.

`void scheme_write(Scheme_Object *`*obj*`, Scheme_Object *`*port*`)`

`write`s the Scheme value *obj* to the given output port.

`void scheme_write_w_max(Scheme_Object *`*obj*`, Scheme_Object *`*port*`, int `*n*`)`

Like `scheme_write`, but the printing is truncated to *n* characters. (If printing is truncated, the last three characters are printed as “.”.)

`void scheme_display(Scheme_Object *`*obj*`, Scheme_Object *`*port*`)`

`display`s the Scheme value *obj* to the given output port.

`void scheme_display_w_max(Scheme_Object *`*obj*`, Scheme_Object *`*port*`, int `*n*`)`

Like `scheme_display`, but the printing is truncated to *n* characters. (If printing is truncated, the last three characters are printed as “.”.)

`void scheme_write_string(char *`*str*`, long `*d*`, long `*len*`, Scheme_Object *`*port*`)`

`display`s *len* characters of *str*, starting with the *d*th character, to the given output port.

`char *scheme_write_to_string(Scheme_Object *`*obj*`, long *`*len*`)` `write`s the Scheme value *obj* to a newly allocated string. If *len* is not `NULL`, `*`*len* is set to the length of the string.

`void scheme_write_to_string_w_max(Scheme_Object *`*obj*`, long *`*len*`, int `*n*`)`

Like `scheme_write_to_string`, but the string is truncated to *n* characters. (If the string is truncated, the last three characters are ".".)

`char *scheme_display_to_string(Scheme_Object *`*obj*`, long *`*len*`)`

`display`s the Scheme value *obj* to a newly allocated string. If *len* is not `NULL`, `*`*len* is set to the length of the string.

`void scheme_display_to_string_w_max(Scheme_Object *`*obj*`, long *`*len*`, int `*n*`)`

Like `scheme_display_to_string`, but the string is truncated to *n* characters. (If the string is truncated, the last three characters are ".".)

`void scheme_debug_print(Scheme_Object *`*obj*`)`

`write`s the Scheme value *obj* to the main thread's output port.

`void scheme_flush_output(Scheme_Object *`*port*`)`

If *port* is a file port, a buffered data is written to the file. Otherwise, there is no effect. *port* must be an output port.

`int scheme_getc(Scheme_Object *`*port*`)`

Get the next character from the given input port.

`int scheme_peekc(Scheme_Object *`*port*`)`

Peeks the next character from the given input port.

`long scheme_get_chars(Scheme_Object *`*port*`, long `*size*`, char *`*buffer*`, int `*offset*`)`

Gets multiple characters at once. The *size* argument indicates the number of requested characters, to be put into the *buffer* array starting at *offset*. The return value is the number of characters actually read, which can be less than the requested number of characters if an end-of-file is encountered. See also `scheme_are_all_chars_ready`.

If *size* is negative, then `scheme_get_chars` returns only as many characters as are immediately ready for reading, up to -*size* characters. If no characters are ready, `scheme_get_chars` blocks until at least one character is ready.

`int scheme_are_all_chars_ready(Scheme_Object *`*port*`)`

Returns 1 if `scheme_char_ready` will never return 0 for *port*. This function is useful for ensuring that `scheme_get_chars` will not block for multiple-character reads.

`void scheme_ungetc(int `*ch*`, Scheme_Object *`*port*`)`

Puts the character *ch* back as the next character to be read from the given input port. The character need not have been read from *port*, and `scheme_ungetc` can be called to insert any number of characters at the start of *port*.

Use `scheme_getc` followed by `scheme_unget` only when your program will certainly call `scheme_getc` again to consume the character. Otherwsie, use `scheme_peekc`, because some a port may implement peeking and getting differently.

`int scheme_char_ready(Scheme_Object *`*port*`)`

Returns 1 if a call to `scheme_getc` is guranteed not to block for the given input port.

`void scheme_need_wakeup(Scheme_Object *`*port*`, void *`*fds*`)`

Requests that appropriate bits are set in *fds* to specify which file descriptors(s) the given input port reads from. (*fds* is sortof a pointer to an `fd_set` struct; see §2.7.3.1.)

`long scheme_tell(Scheme_Object *`*port*`)`

Returns the current read position of the given input port.

`long scheme_tell_line(Scheme_Object *`*port*`)`

Returns the current read line of the given input port. If lines are not counted, -1 is returned.

`void scheme_count_lines(Scheme_Object *`*port*`)`

Turns on line-counting for the given input port. To get accurate line counts, call this function immediately after creating a port.

`void scheme_close_input_port(Scheme_Object *`*port*`)`

Closes the given input port.

`void scheme_close_output_port(Scheme_Object *`*port*`)`

Closes the given output port.

`Scheme_Object *scheme_make_port_type(char *`*name*`)`

Creates a new port subtype.

```
Scheme_Input_Port *scheme_make_input_port(Scheme_Object *subtype,
                    void *data,
                    int (*getc_fun)(Scheme_Input_Port*),
                    int (*peekc_fun)(Scheme_Input_Port*),
                    int (*char_ready_fun)(Scheme_Input_Port*),
                    void (*close_fun)(Scheme_Input_Port*),
                    void (*need_wakeup_fun)(Scheme_Input_Port*, void *),
                    int must_close)
```

Creates a new input port with arbitary control functions. The pointer *data* will be installed as the port's user data, which can be extracted/set with the `SCHEME_INPORT_VAL` macro. The C value `EOF` should be used by *getc_fun* to return an end-of-file. If *peekc_fun* is `NULL`, it is automatically implemented in terms of *getc_fun*.

The function *need_wakeup_fun* will be invoked when the port is blocked on a read; *need_wakeup_fun* should set appropriate bits in *fds* to specify which file decriptor(s) it is blocked on. The *fds* argument is conceptually an array of three `fd_set` structs (one for read, one for write, one for exceptions), but manipulate this array using `scheme_get_fdset` to get a particular element of the array, and use `MZ_FD_XXX` instead of `FD_XXX` to manipulate a single "`fd_set`". Under Windows and BeOS, each "`fd_set`" can also contain OS-level semaphores or other handles via `scheme_add_fd_handle`.

Although the return type of `scheme_make_input_port` is `Scheme_Input_Port *`, it can be cast into a `Scheme_Object *`.

If *must_close* is non-zero, the new port will be registered with the current custodian, and *close_fun* is guranteed to be called before the port is garbage-collected.

```
Scheme_Output_Port *scheme_make_output_port(Scheme_Object *subtype,
                    void *data,
                    void (*write_string_fun)(char *, long, Scheme_Output_Port*),
                    void (*close_fun)(Scheme_Output_Port*),
                    int must_close)
```

Creates a new output port with arbitary control functions. The pointer *data* will be installed as the port's user data, which can be extracted/set with the `SCHEME_OUTPORT_VAL` macro. When *write_string_fun* is called, the second parameter is the length of the string to be written.

Although the return type of `scheme_make_output_port` is `Scheme_Output_Port *`, it can be cast into a `Scheme_Object *`.

If *must_close* is non-zero, the new port will be registered with the current custodian, and *close_fun* is guranteed to be called before the port is garbage-collected.

`Scheme_Object *scheme_make_file_input_port(FILE *fp)`

Creates a Scheme input file port from an ANSI C file pointer.

`Scheme_Object *scheme_make_named_file_input_port(FILE *fp, char *filename)`

Creates a Scheme input file port from an ANSI C file pointer. The filename is used for error reporting.

`Scheme_Object *scheme_make_file_output_port(FILE *fp)`

Creates a Scheme output file port from an ANSI C file pointer.

`Scheme_Object *scheme_make_string_input_port(char *str)`

Creates a Scheme input port from a string; successive `read-char`s on the port return successive characters in the string.

`Scheme_Object *scheme_make_string_output_port()`

Creates a Scheme output port; all writes to the port are kept in a string, which can be obtained with `scheme_get_string_output`.

`char *scheme_get_string_output(Scheme_Object *port)`

Returns (in a newly allocated string) all data that has been written to the given string output port so far. (The returned string is null-terminated.)

`char *scheme_get_sized_string_output(Scheme_Object *`*port*`, int *`*len*`)`

Returns (in a newly allocated string) all data that has been written to the given string output port so far and fills in `*len` with the length of the string (not including the null terminator).

`void scheme_pipe(Scheme_Object **`*read*`, Scheme_Object **`*write*`)`

Creates a pair of ports, setting `*`*read* and `*`*write*; data written to `*`*write* can be read back out of `*`*read*. The pipe can store arbitrarily many unread characters,

`void scheme_pipe_with_limit(Scheme_Object **`*read*`, Scheme_Object **`*write*`, int` *limit*`)`

Like `scheme_pipe` is *limit* is `0`. If *limit* is positive, creates a pipe that stores at most *limit* unread characters, blocking writes when the pipe is full.

`int scheme_file_exists(char *`*name*`)`

Returns 1 if a file by the given name exists, 0 otherwise. If *name* specifies a directory, FALSE is returned. The *name* should be already expanded.

`int scheme_directory_exists(char *`*name*`)`

Returns 1 if a directory by the given name exists, 0 otherwise. The *name* should be already expanded.

`char *scheme_expand_filename(char *`*name*`, int` *len*`, char *`*where*`, int *`*expanded*`)`

Expands the pathname *name*, resolving relative paths with respect to the current directory parameter. Under Unix, this expands "~" into a user's home directory. On the Macintosh, aliases are resolved to real pathnames. The *len* argument is the length of the input string; if it is -1, the string is assumed to be null-terminated. The *where* argument is used if there is an error in the filename; if this is `NULL`, and error is not reported and `NULL` is returned instead. If *expanded* is not `NULL`, **expanded* is set to 1 if some expansion takes place, or 0 if the input name is simply returned.

`char *scheme_build_mac_filename(FSSpec *`*spec*`, int` *isdir*`)`

MacOS only: Converts an `FSSpec` record (defined by MacOS) into a pathname string. If *spec* contains only directory information (via the `vRefNum` and `parID` fields), *isdir* should be `1`, otherwise it should be `0`.

`int scheme_mac_path_to_spec(const char *`*filename*`, FSSpec *`*spec*`, long *`*type*`)`

MacOS only: Converts a pathname into an `FSSpec` record (defined by MacOS), returning `1` if successful and `0` otherwise. If *type* is not `NULL` and *filename* is a file that exists, *type* is filled with the file's four-character MacOS type. If *type* is not `NULL` and *filename* is not a file that exists, *type* is filled with `0`.

`char *scheme_os_getcwd(char *`*buf*`, int` *buflen*`, int *`*actlen*`, int` *noexn*`)`

Gets the current working directory according to the operating system. This is separate from MzScheme's current directory parameter.

The direcory path is written into *buf*, of length *buflen*, if it fits. Otherwise, a new (collectable) string is allocated for the directory path. If *actlen* is not `NULL`, **actlen* is set to the length of the current directory path. If *noexn* is no 0, then an exception is raised if the operation fails.

`int scheme_os_setcwd(char *`*buf*`, int `*noexn*`)`

Sets the current working directory according to the operating system. This is separate from MzScheme's current directory parameter.

If *noexn* is not 0, then an exception is raised if the operation fails.

`char *scheme_format(char *`*format*`, int `*flen*`, int `*argc*`, Scheme_Object **`*argv*`, int *`*rlen*`)`

Creates a string like MzScheme's `format` procedure, using the format string *format* (of length *flen*) and the extra arguments specified in *argc* and *argv*. If *rlen* is not `NULL`, `*`*rlen* is filled with the length of the resulting string.

`void scheme_printf(char *`*format*`, int `*flen*`, int `*argc*`, Scheme_Object **`*argv*`)`

Writes to the current output port like MzScheme's `printf` procedure, using the format string *format* (of length *flen*) and the extra arguments specified in *argc* and *argv*.

2.11 Structures

A new Scheme structure type is created with `scheme_make_struct_type`. This creates the structure type, but does not generate the constructor, etc. procedures. The `scheme_make_struct_values` function takes a structure type and creates these procedures. The `scheme_make_struct_names` function generates the standard structure procvedures names given the structure type's name. Instances of a structure type are created with `scheme_make_struct_instance` and the function `scheme_is_struct_instance` tests a structure's type. The `scheme_struct_ref` and `scheme_struct_set` functions access or modify a field of a structure.

The the structure procedure values and names generated by `scheme_make_struct_values` and `scheme_make_struct_names` can be restricted by passing any combination of these flags:

- `SCHEME_STRUCT_NO_TYPE` — the structure type value/name is not returned.
- `SCHEME_STRUCT_NO_CONSTR` — the constructor procedure value/name is not returned.
- `SCHEME_STRUCT_NO_PRED`— the predicate procedure value/name is not returned.
- `SCHEME_STRUCT_NO_GET` — the selector procedure values/names are not returned.
- `SCHEME_STRUCT_NO_SET` — the mutator procedure values/names are not returned.

When all values or names are returned, they are returned as an array with the following order: structure type, constructor, predicate, first selector, first mutator, second selector, etc. When particular values/names are omitted, the array is compressed accordingly.

2.11.1 Library Functions

`Scheme_Object *scheme_make_struct_type(Scheme_Object *`*base_name*`, Scheme_Object *`*super_type*`,`
`int `*num_fields*`)`

Creates and returns a new structure type. The *base_name* argument is used as the name of the new structure type; it must be a symbol. The *super_type* argument should be `NULL` or an existing structure type to use as the super-type. The *num_fields* argument specifies the number of fields for instances of this structure type. (If a super-type is used, this is the number of additional fields, rather than the total number.)

`Scheme_Object **scheme_make_struct_names(Scheme_Object *`*base_name*`, Scheme_Object *`*field_names*`, int `*flags*`, int *`*count_out*`)`

Creates and returns an array of standard structure value name symbols. The *base_name* argument is used as the name of the structure type; it should be the same symbol passed to the associated call to `scheme_make_struct_type`. The *field_names* argument is a (Scheme) list of field name symbols. The *flags* argument specifies which names should be generated, and if *count_out* is not `NULL`, *count_out* is filled with the number of names returned in the array.

`Scheme_Object **scheme_make_struct_values(Scheme_Object *`*struct_type*`, Scheme_Object **`*names*`, int `*count*`, int `*flags*`)`

Creates and returns an array of the standard structure value and procedure values for *struct_type*. The *struct_type* argument must be a structure type value created by `scheme_make_struct_type`. The *names* procedure must be an array of name symbols, generally the array returned by `scheme_make_struct_names`. The *count* argument specifies the length of the *names* array (and therefore the number of expected return values) and the *flags* argument specifies which values should be generated.

`Scheme_Object *scheme_make_struct_instance(Scheme_Object *`*struct_type*`, int `*argc*`, Scheme_Object **`*argv*`)`

Creates an instance of the structure type *struct_type*. The *argc* and *argv* arguments provide the field values for the new instance.

`int scheme_is_struct_instance(Scheme_Object *`*struct_type*`, Scheme_Object *`*v*`)`

Returns 1 if *v* is an instance of *struct_type* or 0 otherwise.

`Scheme_Object *scheme_struct_ref(Scheme_Object *`*s*`, int `*n*`)`

Returns the *n*th field (counting from 0) in the structure *s*.

`void scheme_struct_set(Scheme_Object *`*s*`, int `*n*`, Scheme_Object *`*v*`)`

Sets the *n*th field (counting from 0) in the structure *s* to *v*.

2.12 Units

Primitive units can be created by allocating an instance of the `Scheme_Unit` data type:

```
typedef struct Scheme_Unit {
  Scheme_Type type;  /* = scheme_unit_type */
  short num_imports;
  short num_exports;
  Scheme_Object **exports;
  Scheme_Object **export_debug_names; /* NULL */
  Scheme_Object *(*init_func)(Scheme_Object **boxes, Scheme_Object **anchors,
                              struct Scheme_Unit *m, void *debug_request);
  Scheme_Object *data;
} Scheme_Unit;
```

The fields are filled as follows:

- The `type` field is always `scheme_unit_type`.
- The `num_imports` field specifies the number of variables imported by the unit and the `num_exports` field specifies the number of variables exported.
- Exported variables are named; the `exports` field must point to an array of symbols for the variable names.
- The `export_debug_names` field is `NULL` for primitive units.
- The `init_func` field points to a function that is called when the unit is instantiated. (A single unit can be instantiated multiple times.) The first argument to this procedure is an array of boxes for import and export variables (import variables first); the value of an imported or exported variable is the value in the corresponding box, accessed or set with the `SCHEME_ENVBOX_VAL` macro. Boxes for imported variables should never be mutated. Boxes for exported variables will be initialized to `scheme_undefined` and should be properly initialized by the `init_func` function.

  The second argument to `init_func` is an array of anchor pointers associated with the boxes in the first argument. Whenever a box pointer is kept, the corresponding anchor pointer must also be kept to keep the box from being collected as garbage. Note that the anchor is for the box itself, *not* the value within the box.

  The final argument to `init_func` should be ignored.

  The return value of `init_func` corresponds to the value of the last expression in the body of a Scheme-based unit.
- The `data` field is not used directly by MzScheme; it is available to store unit-specific data needed by `init_func`.

2.12.1 Library Functions

`Scheme_Object *scheme_invoke_unit(Scheme_Object *`*unit*`, int` *num_ins*`,`
`Scheme_Object **`*ins*`, Scheme_Object **`*anchors*`, int` *tail*`, int` *multi*`)`

Invokes a unit. The *num_ins* argument specifies the number of variables to import into the unit. The *ins* array must be an array of variables boxes (`NULL` if no variables are imported). The *anchors* argument is is parallel to the *ins* array, providing a garbage-collecting anchor for each variable. The `scheme_invoke_unit` function will check that the correct number of variables are provided for importing into the unit.

A variable box can be any pointer. The pointer is deferenced as a `Scheme_Object **` to get the variable box's contents. Anchors are associated with variable boxes so that a box can point into the middle of an allocated array; in this case, the anchor would be the start of the array, so that the garbage collector sees a reference to the array.

If *tail* is non-zero, `scheme_invoke_unit` produces a tail-call to invoke the unit. If *tail* is zero and *multi* is non-zero, multiple values may be returned.

`Scheme_Object *scheme_make_envunbox(Scheme_Object *`*v*`)`

Creates a new variable box with *v* as the initial value. No anchor is needed (i.e., `NULL` can be used as an anchor) for boxes created this way.

`Scheme_Object *scheme_assemble_compound_unit(Scheme_Object *`*imports*`, Scheme_Object *`*links*`,`
`Scheme_Object *`*exports*`)`

"Compiles" a compound-unit expression, given the names for the compound unit's imports, exports, and sub-unit linking.

- The *imports* argument is a Scheme list of symbols for the imported variable names.
- The *links* argument is a list of sub-unit linking specifications, where each specification is a pair consisting of:
  - a single tag symbol, used to identify the unit for links and re-exports
  - a list of variable specifications, where each variable specification is either
    - a symbol that is present in the *imports* list, specifying a link to a variable imported into the compound unit, or
    - pair consisting of a tag symbol and a list of symbols, specifying the names of exported variables from another sub-unit in the compound unit
- the *exports* argument is a list of sub-unit export specifications, where each export specification is a pair consisting of
  - a tag symbol
  - a symbol naming one export from the corresponding sub-unit (used as both the name of the sub-unit export and the name of the compound unit's export), or a pair of symbols where the first one is the name of the sub-unit's export and the second one is the name as exported from the compound unit

The return value is an "assembled" compound unit. A compound unit is created from the assembly with `scheme_make_compound_unit`.

`Scheme_Object *scheme_make_compound_unit(Scheme_Object *`*assembly*`, Scheme_Object **`*subs*`)`

Returns a compound unit given an assembly created by `scheme_assemble_compound_unit` and an array of sub-units to be linked into the compound unit. The order of units in the sub-unit array should parallel the order of tags in the assembly's *links* specification.

2.13 Objects, Classes, and Interfaces

Primitive C++-like classes can be created with `scheme_make_class`. Methods are added to a primitive class with `scheme_add_method`; all methods must be added to a class before an object is created from the class. A C function that implements a class method is similar to a closed primitive function: it is passed a pointer to the object, a interger inidicating the number of arguments passed to the method, and an array of `Scheme_Object *` arguments.

More general classes are created in two phases. The `scheme_make_class_assembly` function creates a class assembly value that represents a "compiled" class expression. A class is created from an assembly with `scheme_create_class` with a creation-time determined superclass. An initialization procedure that is passed to `scheme_make_class_assembly` is called whenever an instance of the class is created.

Interfaces are also created in two phases: `scheme_make_interface_assembly` creates a "compiled" interface expression, and `scheme_create_interface` instantiates an actual interface from an assembly.

The function `scheme_make_object` creates a new object from a class and list of initialization arguments. Instance variables are retrieved with `scheme_find_ivar`, which takes an object and a symbol and returns the instance variable's value, or `NULL` if it is not found. Classes and objects can be compared with `scheme_is_a` and `scheme_is_subclass`.

An object value contains one pointer field that can be used by an implementation of a primitive class; this field is set or accessed with the `SCHEME_OBJ_DATA` macro. There is an additional flag field – set/accessed with `SCHEME_OBJ_FLAG` — that is initialized to 0; if this flag is set to a negative value, then the object will no longer be usable from Scheme. (This is useful, for example, for closing a Scheme object when a corresponding C++ object can no longer be used.)

Connecting an arbitrary C++ class library to MzScheme can be tricky, and may require a large amount of glue code. The **plt/collects/mzscheme/examples** directory in the PLT distribution contains a **tree.cxx** example.

2.13.1 Library Functions

`Scheme_Object *scheme_make_class(char *`*name*`, Scheme_Object *`*sup*`,`
`Scheme_Method_Prim *`*init*`, int `*num_methods*`)`

Creates a new primitive class. If an intializer method *init* is provided, then objects of this class can be created from Scheme. The class *sup* specifies a superclass for the primitive class; it can be `NULL` to indicate `object%`. The *num_methods* argument must be an upper-bound on the actual number of methods to be installed with `scheme_add_method_w_arity` or `scheme_add_method`. Once all of the methods are installed, `scheme_made_class` must be called.

`void scheme_add_method_w_arity(Scheme_Object *`*cl*`, char *`*name*`, Scheme_Method_Prim *`*f*`,`
`short `*mina*`, short `*maxa*`)`

Adds a primitive method to a primitive class. The form of the method *f* is defined by:

```
Scheme_Object *Scheme_Method_Prim(Scheme_Object *obj, int argc, Scheme_Object **argv);
```

`void scheme_add_method(Scheme_Object *`*cl*`, char *`*name*`, Scheme_Method_Prim *`*f*`)`

Like `scheme_add_method_w_arity`, but *mina* and *maxa* are defaulted to 0 and -1, respectively.

`void scheme_made_class(Scheme_Object *`*cl*`)`

Indicates that all of the methods have been added to the primitive class *cl*.

`Scheme_Object *scheme_make_object(Scheme_Object *`*sclass*`, int `*argc*`, Scheme_Object **`*argv*`)`

Creates an instance of the class *sclass*. The arguments to the object's intialization function are speicified by *argc* and *argv*.

`Scheme_Object *scheme_make_uninited_object(Scheme_Object *`*sclass*`)`

Creates a Scheme object instance of *sclass* without intitializing the object. This is useful for creating a Scheme representation of an existing primitive object.

`Scheme_Object *scheme_find_ivar(Scheme_Object *`*obj*`, Scheme_Object *`*sym*`, int `*force*`)`

Finds an instance variable by name (as a symbol). Returns `NULL` if the instance variable is not found. The *force* argument should be `1`.

`Scheme_Object *scheme_get_generic_data(Scheme_Object *`*class_or_intf*`, Scheme_Object *`*name*`)`

Creates a Scheme value that contains the essential information of a generic procedure. This information can be applied to an object using `scheme_apply_generic_data`. If the named field is not found in the specified class, then the `NULL` pointer is returned.

`Scheme_Object *scheme_apply_generic_data(Scheme_Object *`*gdata*`, Scheme_Object *`*sobj*`, int `*force*`)`

Given the result of a call to `scheme_get_generic_data`, extracts a value from the specified Scheme object. If the object is not in the appropriate class, and error is raised.

If *force* is 0 and the instance variable in the object is implemented as a primitive method, then NULL is returned.

`int scheme_is_subclass(Scheme_Object *`*sub*`, Scheme_Object *`*parent*`)`

Returns 1 if the class *sub* is derived from the class *parent*, 0 otherwise.

`int scheme_is_implementation(Scheme_Object *`*cl*`, Scheme_Object *`*intf*`)`

Returns 1 if the class *cl* implements the interface *intf*, 0 otherwise.

`int scheme_is_interface_extension(Scheme_Object *`*sub*`, Scheme_Object *`*intf*`)`

Returns 1 if the interface *sub* is an extension of the interface *intf*, 0 otherwise.

`int scheme_is_a(Scheme_Object *`*obj*`, Scheme_Object *`*sclass*`)`

Returns 1 if *obj* is an instance of the class *sclass* or of a class derived from *sclass*, 0 otherwise.

`char *scheme_get_class_name(Scheme_Object *`*sclass*`, int *`*len*`)`

Returns the name of the class *sclass* if it has one, or `NULL` otherwise. If the return value is not `NULL`, `*len` is set to the length of the string.

```
struct Scheme_Class_Assembly *scheme_make_class_assembly(
                              const char *name, int n_interface,
                              int n_public, Scheme_Object **publics,
                              int n_override, Scheme_Object **overrides,
                              int n_inh, Scheme_Object **inherits,
                              int n_ren, Scheme_Object **renames,
                              int mina, int maxa,
                              Scheme_Instance_Init_Proc *initproc)
```

"Compiles" a class expression, given a name for the class (or `NULL`), the number of interfaces that will be declared as implemented by the class in *n_interfaces*, and names for `public`, `override`, `inherit`, and `rename` instance variables as symbols. The *mina* and *maxa* arguments specify the arity of the initialization procedure (i.e., the implicit `lambda` in a class expression that accepts initialization arguments). The *initproc* function has the following prototype:

```
typedef void (*Scheme_Instance_Init_Proc)(Scheme_Object **init_boxes,
                                          Scheme_Object **extract_boxes,
                                          Scheme_Object *super_init,
                                          int argc,
                                          Scheme_Object **argv,
                                          Scheme_Object *instance,
                                          void *data);
```

When an instance of the class is created, *initproc* will be called. The first two arguments are arrays of environment boxes (whose values are manipulated with `SCHEME_ENVBOX_VAL`). These arrays are in parallel: the first array is used for initializing variables from local expressions, and the second array is for looking up the value of a possibly-overridden instance variable. In both arrays, the `public`, `override`, `inherit`, and `rename` variables are ordered as provided in `scheme_make_class_assembly` (with public variables first, then override, then private), but `init_boxes` only contains boxes for `public` and `override` variables. The *argc* and *argv* arguments specify the values passed in as initialization arguments. The *super_init* argument is the procedure for initializing the superclass (use `_scheme_apply` to invoke it). The *instance* argument is the value of `this`. The *data* argument is supplied by the caller of `scheme_create_class`.

The result from `scheme_make_class_assembly` is used with `scheme_create_class` to create an actual class at run-time given the a run-time-determined superclass and interfaces.

```
Scheme_Object *scheme_create_class(struct Scheme_Class_Assembly *a,
               void *data, Scheme_Object *super, Scheme_Object **interfaces)
```

Returns a Scheme class value given the result of a call to `scheme_make_class_assembly`, a superclass, and an array of interface values. (The number of interfaces values must match the number of interfaces specified in the call to `scheme_make_class_assembly`.) Type-checking on the superclass and interface array is performed by `scheme_create_class`.

```
struct Scheme_Interface_Assembly *scheme_make_interface_assembly(
                                  const char *name, int n_supers,
                                  int n_names, Scheme_Object **names)
```

"Compiles" an interface expression, given the interface's name (or NULL), the number of super interfaces that will be extended by the interface in *n_supers*, and names for instance variables as symbols.

The result from `scheme_make_interface_assembly` is used with `scheme_create_interface` to create an actual class at run-time given the run-time-determined superinterfaces.

```
Scheme_Object *scheme_create_interface(struct Scheme_Interface_Assembly *a,
               Scheme_Object **supers)
```

Returns a Scheme interface value given the result of a call to `scheme_make_interface_assembly` and an array of superinterface values. (The number of superinterfaces values must match the number of superinterfaces specified in the call to `scheme_make_interface_assembly`.) Type-checking on the superinterface array is performed by `scheme_create_interface`.

2.14 Custodians

In MzScheme's C library interface, custodians are called "managers".

2.14.1 Library Functions

```
Scheme_Manager *scheme_make_manager(Scheme_Manager *m)
```

Creates a new custodian as a subordinate of m. If m is `NULL`, then the current custodian is used as the new custodian's supervisor.

```
Scheme_Manager_Reference *scheme_add_managed(Scheme_Manager *m, Scheme_Object *o,
                          Scheme_Close_Manager_Client *f, void *data,
                          int strong)
```

Places the value *o* into the management of the custodian *m*. The *f* function is called by the custodian if it is ever asked to "shutdown" its values; *o* and *data* are passed on to *f*, which has the type

```
typedef void (*Scheme_Close_Manager_Client)(Scheme_Object *o, void *data);
```

If *strong* is non-zero, then the newly managed value will be remembered until either the custodian shuts it down or `scheme_remove_managed` is called. If *strong* is zero, the value is allowed to be garbaged collected (and automatically removed from the custodian).

The return value from `scheme_add_managed` can be used to refer to the value's custodian later in a call to `scheme_remove_managed`. A value can be registered with at most one custodian.

`void scheme_remove_managed(Scheme_Manager_Reference *`*mref*`, Scheme_Object *`*o*`)`

Removes *o* from the management of its custodian. The *mref* argument must be a value returned by `scheme_add_managed`.

`void scheme_close_managed(Scheme_Manager *`*m*`)`

Instructs the custodian *m* to shutdown all of its managed values.

2.15 Miscellaneous Utilities

2.15.1 Library Functions

`int scheme_eq(Scheme_Object *`*obj1*`, Scheme_Object *`*obj2*`)`

Returns 1 if the Scheme values are `eq?`.

`int scheme_eqv(Scheme_Object *`*obj1*`, Scheme_Object *`*obj2*`)`

Returns 1 if the Scheme values are `eqv?`.

`int scheme_equal(Scheme_Object *`*obj1*`, Scheme_Object *`*obj2*`)`

Returns 1 if the Scheme values are `equal?`.

`Scheme_Object *scheme_build_list(int `*c*`, Scheme_Object **`*elems*`)`

Creates and returns a list of length *c* with the elements *elems*.

`int scheme_list_length(Scheme_Object *`*list*`)`

Returns the length of the list. If *list* is not a proper list, then the last `cdr` counts as an item. If there is a cycle in *list* (involvng only `cdr`s), this procedure will not terminate.

`int scheme_proper_list_length(Scheme_Object *`*list*`)`

Returns the length of the list, or -1 if it is not a proper list. If there is a cycle in *list* (involvng only `cdr`s), this procedure returns -1.

`Scheme_Object *scheme_car(Scheme_Object *`*pair*`)`

Returns the `car` of the pair.

`Scheme_Object *scheme_cdr(Scheme_Object *`*pair*`)`

Returns the `cdr` of the pair.

`Scheme_Object *scheme_cadr(Scheme_Object *`*pair*`)`

Returns the `cadr` of the pair.

`Scheme_Object *scheme_caddr(Scheme_Object *`*pair*`)`

Returns the `caddr` of the pair.

`Scheme_Object *scheme_vector_to_list(Scheme_Object *`*vec*`)`

Creates a list with the same elements as the given vector.

`Scheme_Object *scheme_list_to_vector(Scheme_Object *`*list*`)`

Creates a vector with the same elements as the given list.

`Scheme_Object *scheme_append(Scheme_Object *`*lstx*, `Scheme_Object *`*lsty*`)`

Non-destructively appends the given lists.

`Scheme_Object *scheme_unbox(Scheme_Object *`*obj*`)`

Returns the contents of the given box.

`void scheme_set_box(Scheme_Object *`*b*, `Scheme_Object *`*v*`)`

Sets the contents of the given box.

`Scheme_Object *scheme_load(char *`*file*`)`

Loads the specified Scheme file, returning the value of the last expression loaded, or `NULL` if the load fails.

`Scheme_Object *scheme_load_extension(char *`*filename*`)`

Loads the specified Scheme extension file, returning the value provided by the extension's initialization function.

`long scheme_double_to_int(char *`*where*, `double` *d*`)`

Returns a fixnum value for the given floating-point number d. If d is not an integer or if it is too large, then an error message is reported; *name* is used for error-reporting.

`void scheme_secure_exceptions(Scheme_Env *`*env*`)`

Secures the primitive exception types, just like `secure-primitive-expcetion-types`.

`long scheme_get_millseconds()`

Returns the current "time" in millseconds, just like `current-millseconds`.

`long scheme_get_process_millseconds()`

Returns the current process "time" in millseconds, just like `current-process-millseconds`.

`char *scheme_banner()`

Returns the string that is used as the MzScheme startup banner.

`char *scheme_version()`

Returns a string for the executing version of MzScheme.

2.16 Flags and Hooks

These flags and hooks are availble when MzScheme is embedded:

- `scheme_exit` — This pointer can be set to a function which takes an integer argument and returns `void`; the function will be used as the default exit handler. The default is `NULL`.
- `scheme_console_printf` — This pointer can be set to a function that takes arguments like `printf`; the function will be called to display internal MzScheme warnings and messages. The default is *NULL*.
- `scheme_console_output` — This pointer can be set to a function that takes a string and a `long` string length; the function will be called to display internal MzScheme warnings and messages that possibly contain non-terminating nuls. The default is *NULL*.
- `scheme_check_for_break` — This points to a function of no arguments that returns an integer. It is used as the default user-break polling procedure in the main thread. (A non-zero return value indicates a user break.) The default is `NULL`.
- `scheme_make_stdin`, `scheme_make_stdout`, `scheme_make_stderr`, — These pointers can be set to a function that takes no arguments and returns a Scheme port `Scheme_Object *` to be used as the starting standard input, output, and/or error port. The defaults are `NULL`.
- `scheme_case_sensitive` — If this flag is set to a non-zero value before `scheme_basic_env` is called, then MzScheme will not ignore capitalization for synbols and global variable names. The value of this flag should not change once it is set. The default is zero.
- `scheme_constant_builtins` — If this flag is set to a non-zero value before `scheme_basic_env` is called, then the standard MzScheme functions and syntax will be defined as constant globals. The default is zero.
- `scheme_no_keywords` — If this flag is set to a non-zero value before `scheme_basic_env` is called, then no keywords are enforced; i.e., the names of the core syntactic forms and all "#%" names are available for local variable names. The default is zero.
- `scheme_allow_set_undefined` — This flag determines the initial value of `compile-allow-set!-undefined`. The default is zero.
- `scheme_allow_cond_auto_else` — This flag determines the initial value of `compile-allow-cond-fallthrough`. The default is non-zero.
- `scheme_secure_primitive_exn` — If this flag is set to non-zero, then the structure type values and constructors for the primitive exception types will not be defined as global variables. The default is zero.

- `scheme_escape_continuations_only` — If this flag is set to a non-zero value before `scheme_basic_env` is called, then `call/cc` will be remapped to `call/ec`; this is useful for speeding up Scheme evaluation when continuations are only used for escaping. The default is zero.

Index