Guile Reference Manual

Edition 1.1, for use with Guile 1.8.1

The Guile Developers

This reference manual documents Guile, GNU's Ubiquitous Intelligent Language for Extensions. This is edition 1.1 corresponding to Guile 1.8.1.

Table of Contents

sec

1 Preface

This reference manual documents Guile, GNU's Ubiquitous Intelligent Language for Extensions. It describes how to use Guile in many useful and interesting ways.

This is edition 1.1 of the reference manual, and corresponds to Guile version 1.8.1.

1.1 Layout of this Manual

The manual is divided into five chapters.

Chapter 1: Introduction to Guile
This part provides an overview of what Guile is and how you can use it. A whirlwind tour shows how Guile can be used interactively and as a script interpreter, how to link Guile into your own applications, and how to write modules of interpreted and compiled code for use with Guile. Everything introduced here is documented again and in full by the later parts of the manual. This part also explains how to obtain and install new versions of Guile, and how to report bugs effectively.

Chapter 2: Programming in Scheme
This part provides an overview over programming in Scheme with Guile. It covers how to invoke the `guile` program from the command-line and how to write scripts in Scheme. It also gives an introduction into the basic ideas of Scheme itself and to the various extensions that Guile offers beyond standard Scheme.

Chapter 3: Programming in C
This part provides an overview of how to use Guile in a C program. It discusses the fundamental concepts that you need to understand to access the features of Guile, such as dynamic types and the garbage collector. It explains in a tutorial like manner how to define new data types and functions for the use by Scheme programs.

Chapter 4: Guile API Reference
This part of the manual documents the Guile API in functionality-based groups with the Scheme and C interfaces presented side by side.

Chapter 5: Guile Modules
Describes some important modules, distributed as part of the Guile distribution, that extend the functionality provided by the Guile Scheme core.

1.2 Conventions used in this Manual

We use some conventions in this manual.

- For some procedures, notably type predicates, we use *iff* to mean "if and only if". The construct is usually something like: 'Return *val* iff *condition*', where *val* is usually "`#t`" or "non-`#f`". This typically means that *val* is returned if *condition* holds, and that '`#f`' is returned otherwise. To clarify: *val* will **only** be returned when *condition* is true.
- In examples and procedure descriptions and all other places where the evaluation of Scheme expression is shown, we use some notation for denoting the output and evaluation results of expressions.

The symbol '⇒' is used to tell which value is returned by an evaluation:

```
(+ 1 2)
⇒ 3
```

Some procedures produce some output besides returning a value. This is denoted by the symbol '⊣'.

```
(begin (display 1) (newline) 'hooray)
⊣ 1
⇒ hooray
```

As you can see, this code prints '`1`' (denoted by '⊣'), and returns `hooray` (denoted by '⇒'). Do not confuse the two.

1.3 Contributors to this Manual

The Guile reference and tutorial manuals were written and edited largely by Mark Galassi and Jim Blandy. In particular, Jim wrote the original tutorial on Guile's data representation and the C API for accessing Guile objects.

Significant portions were contributed by Gary Houston (contributions to POSIX system calls and networking, expect, I/O internals and extensions, slib installation, error handling) and Tim Pierce (sections on script interpreter triggers, alists, function tracing).

Tom Lord contributed a great deal of material with early Guile snapshots; although most of this text has been rewritten, all of it was important, and some of the structure remains.

Aubrey Jaffer wrote the SCM Scheme implementation and manual upon which the Guile program and manual are based. Some portions of the SCM and SLIB manuals have been included here verbatim.

Since Guile 1.4, Neil Jerram has been maintaining and improving the reference manual. Among other contributions, he wrote the Basic Ideas chapter, developed the tools for keeping the manual in sync with snarfed libguile docstrings, and reorganized the structure so as to accommodate docstrings for all Guile's primitives.

Martin Grabmueller has made substantial contributions throughout the reference manual in preparation for the Guile 1.6 release, including filling out a lot of the documentation of Scheme data types, control mechanisms and procedures. In addition, he wrote the documentation for Guile's SRFI modules and modules associated with the Guile REPL.

1.4 The Guile License

Guile is Free Software. Guile is copyrighted, not public domain, and there are restrictions on its distribution or redistribution, but these restrictions are designed to permit everything a cooperating person would want to do.

- The Guile library (libguile) and supporting files are published under the terms of the GNU Lesser General Public License version 2.1. See the file '`COPYING.LIB`'.
- The Guile readline module is published under the terms of the GNU General Public License version 2. See the file '`COPYING`'.
- The manual you're now reading is published under the terms of the GNU Free Documentation License (see Appendix B [GNU Free Documentation License], page 503).

C code linking to the Guile library is subject to terms of that library. Basically such code may be published on any terms, provided users can re-link against a new or modified version of Guile.

C code linking to the Guile readline module is subject to the terms of that module. Basically such code must be published on Free terms.

Scheme level code written to be run by Guile (but not derived from Guile itself) is not resticted in any way, and may be published on any terms. We encourage authors to publish on Free terms.

You must be aware there is no warranty whatsoever for Guile. This is described in full in the licenses.

2 Introduction to Guile

2.1 What is Guile?

Guile is an interpreter for the Scheme programming language, packaged for use in a wide variety of environments. Guile implements Scheme as described in the Revised^5 Report on the Algorithmic Language Scheme (usually known as R5RS), providing clean and general data and control structures. Guile goes beyond the rather austere language presented in R5RS, extending it with a module system, full access to POSIX system calls, networking support, multiple threads, dynamic linking, a foreign function call interface, powerful string processing, and many other features needed for programming in the real world.

Like a shell, Guile can run interactively, reading expressions from the user, evaluating them, and displaying the results, or as a script interpreter, reading and executing Scheme code from a file. However, Guile is also packaged as an object library, allowing other applications to easily incorporate a complete Scheme interpreter. An application can then use Guile as an extension language, a clean and powerful configuration language, or as multi-purpose "glue", connecting primitives provided by the application. It is easy to call Scheme code from C code and vice versa, giving the application designer full control of how and when to invoke the interpreter. Applications can add new functions, data types, control structures, and even syntax to Guile, creating a domain-specific language tailored to the task at hand, but based on a robust language design.

Guile's module system allows one to break up a large program into manageable sections with well-defined interfaces between them. Modules may contain a mixture of interpreted and compiled code; Guile can use either static or dynamic linking to incorporate compiled code. Modules also encourage developers to package up useful collections of routines for general distribution; as of this writing, one can find Emacs interfaces, database access routines, compilers, GUI toolkit interfaces, and HTTP client functions, among others.

In the future, we hope to expand Guile to support other languages like Tcl and Perl by translating them to Scheme code. This means that users can program applications which use Guile in the language of their choice, rather than having the tastes of the application's author imposed on them.

2.2 Obtaining and Installing Guile

Guile can be obtained from the main GNU archive site ftp://ftp.gnu.org or any of its mirrors. The file will be named guile-version.tar.gz. The current version is 1.8.1, so the file you should grab is:

ftp://ftp.gnu.org/pub/gnu/guile-1.8.1.tar.gz

To unbundle Guile use the instruction

```
zcat guile-1.8.1.tar.gz | tar xvf -
```

which will create a directory called '`guile-1.8.1`' with all the sources. You can look at the file '`INSTALL`' for detailed instructions on how to build and install Guile, but you should be able to just do

```
cd guile-1.8.1
./configure
```

```
make
make install
```

This will install the Guile executable '`guile`', the Guile library '`-lguile`' and various associated header files and support libraries. It will also install the Guile tutorial and reference manual.

Since this manual frequently refers to the Scheme "standard", also known as R5RS, or the "Revised^5 Report on the Algorithmic Language Scheme", we have included the report in the Guile distribution; See section "Introduction" in *Revised(5) Report on the Algorithmic Language Scheme*. This will also be installed in your info directory.

2.3 A Whirlwind Tour

This chapter presents a quick tour of all the ways that Guile can be used. There are additional examples in the '`examples/`' directory in the Guile source distribution.

The following examples assume that Guile has been installed in `/usr/local/`.

2.3.1 Running Guile Interactively

In its simplest form, Guile acts as an interactive interpreter for the Scheme programming language, reading and evaluating Scheme expressions the user enters from the terminal. Here is a sample interaction between Guile and a user; the user's input appears after the `$` and `guile>` prompts:

```
$ guile
guile> (+ 1 2 3)                ; add some numbers
6
guile> (define (factorial n)    ; define a function
         (if (zero? n) 1 (* n (factorial (- n 1)))))
guile> (factorial 20)
2432902008176640000
guile> (getpwnam "jimb")        ; find my entry in /etc/passwd
#("jimb" ".0krIpK2VqNbU" 4008 10 "Jim Blandy" "/u/jimb"
  "/usr/local/bin/bash")
guile> C-d
$
```

2.3.2 Running Guile Scripts

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

Here is a trivial Guile script, for more details See Section 3.3 [Guile Scripting], page 33.

```
#!/usr/local/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

2.3.3 Linking Guile into Programs

The Guile interpreter is available as an object library, to be linked into applications using Scheme as a configuration or extension language.

Here is '`simple-guile.c`', source code for a program that will produce a complete Guile interpreter. In addition to all usual functions provided by Guile, it will also offer the function `my-hostname`.

```
#include <stdlib.h>
#include <libguile.h>

static SCM
my_hostname (void)
{
  char *s = getenv ("HOSTNAME");
  if (s == NULL)
    return SCM_BOOL_F;
  else
    return scm_from_locale_string (s);
}

static void
inner_main (void *data, int argc, char **argv)
{
  scm_c_define_gsubr ("my-hostname", 0, 0, 0, my_hostname);
  scm_shell (argc, argv);
}

int
main (int argc, char **argv)
{
  scm_boot_guile (argc, argv, inner_main, 0);
  return 0; /* never reached */
}
```

When Guile is correctly installed on your system, the above program can be compiled and linked like this:

```
$ gcc -o simple-guile simple-guile.c -lguile
```

When it is run, it behaves just like the `guile` program except that you can also call the new `my-hostname` function.

```
$ ./simple-guile
guile> (+ 1 2 3)
6
guile> (my-hostname)
"burns"
```

2.3.4 Writing Guile Extensions

You can link Guile into your program and make Scheme available to the users of your program. You can also link your library into Guile and make its functionality available to all users of Guile.

A library that is linked into Guile is called an *extensions*, but it really just is an ordinary object library.

The following example shows how to write a simple extension for Guile that makes the `j0` function available to Scheme code.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_make_real (j0 (scm_num2dbl (x, "j0")));
}

void
init_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

This C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux:

```
gcc -shared -o libguile-bessel.so -fPIC bessel.c
```

For creating shared libraries portably, we recommend the use of GNU Libtool (see section "Introduction" in *GNU Libtool*).

A shared library can be loaded into a running Guile process with the function `load-extension`. The `j0` is then immediately available:

```
$ guile
guile> (load-extension "./libguile-bessel" "init_bessel")
guile> (j0 2)
0.223890779141236
```

2.3.5 Using the Guile Module System

Guile has support for dividing a program into *modules*. By using modules, you can group related code together and manage the composition of complete programs from largely independent parts.

(Although the module system implementation is in flux, feel free to use it anyway. Guile will provide reasonable backwards compatibility.)

Details on the module system beyond this introductory material can be found in See Section 5.16 [Modules], page 303.

2.3.5.1 Using Modules

Guile comes with a lot of useful modules, for example for string processing or command line parsing. Additionally, there exist many Guile modules written by other Guile hackers, but which have to be installed manually.

Here is a sample interactive session that shows how to use the `(ice-9 popen)` module which provides the means for communicating with other processes over pipes together with the `(ice-9 rdelim)` module that provides the function `read-line`.

```
$ guile
guile> (use-modules (ice-9 popen))
guile> (use-modules (ice-9 rdelim))
guile> (define p (open-input-pipe "ls -l"))
guile> (read-line p)
"total 30"
guile> (read-line p)
"drwxr-sr-x    2 mgrabmue mgrabmue     1024 Mar 29 19:57 CVS"
```

2.3.5.2 Writing new Modules

You can create new modules using the syntactic form `define-module`. All definitions following this form until the next `define-module` are placed into the new module.

One module is usually placed into one file, and that file is installed in a location where Guile can automatically find it. The following session shows a simple example.

```
$ cat /usr/local/share/guile/foo/bar.scm

(define-module (foo bar))
(export frob)

(define (frob x) (* 2 x))

$ guile
guile> (use-modules (foo bar))
guile> (frob 12)
24
```

2.3.5.3 Putting Extensions into Modules

In addition to Scheme code you can also put things that are defined in C into a module.

You do this by writing a small Scheme file that defines the module and call `load-extension` directly in the body of the module.

```
$ cat /usr/local/share/guile/math/bessel.scm

(define-module (math bessel))
(export j0)

(load-extension "libguile-bessel" "init_bessel")

$ file /usr/local/lib/libguile-bessel.so
... ELF 32-bit LSB shared object ...
$ guile
guile> (use-modules (math bessel))
guile> (j0 2)
0.223890779141236
```

There is also a way to manipulate the module system from C but only Scheme files can be autoloaded. Thus, we recommend that you define your modules in Scheme.

2.4 Discouraged and Deprecated

From time to time functions and other features of Guile become obsolete. Guile has some mechanisms in place that can help you cope with this.

Guile has two levels of obsoleteness: things can be *deprecated*, meaning that their use is considered harmful and should be avoided, even in old code; or they can be merely *discouraged*, meaning that they are fine in and of themselves, but that there are better alternatives that should be used in new code.

When you use a feature that is deprecated, you will likely get a warning message at run-time. Also, deprecated features are not ready for production use: they might be very slow. When something is merely discouraged, it performs normally and you wont get any messages at run-time.

The primary source for information about just what things are discouraged or deprecated in a given release is the file '`NEWS`'. That file also documents what you should use instead of the obsoleted things.

The file '`README`' contains instructions on how to control the inclusion or removal of the deprecated and/or discouraged features from the public API of Guile, and how to control the warning messages for deprecated features.

The idea behind those mechanisms is that normally all deprecated and discouraged features are available, but that you can omit them on purpose to check whether your code still relies on them.

2.5 Reporting Bugs

Any problems with the installation should be reported to `bug-guile@gnu.org`.

Whenever you have found a bug in Guile you are encouraged to report it to the Guile developers, so they can fix it. They may also be able to suggest workarounds when it is not possible for you to apply the bug-fix or install a new version of Guile yourself.

Before sending in bug reports, please check with the following list that you really have found a bug.

- Whenever documentation and actual behavior differ, you have certainly found a bug, either in the documentation or in the program.
- When Guile crashes, it is a bug.
- When Guile hangs or takes forever to complete a task, it is a bug.
- When calculations produce wrong results, it is a bug.
- When Guile signals an error for valid Scheme programs, it is a bug.
- When Guile does not signal an error for invalid Scheme programs, it may be a bug, unless this is explicitly documented.
- When some part of the documentation is not clear and does not make sense to you even after re-reading the section, it is a bug.

When you write a bug report, please make sure to include as much of the information described below in the report. If you can't figure out some of the items, it is not a problem, but the more information we get, the more likely we can diagnose and fix the bug.

- The version number of Guile. Without this, we won't know whether there is any point in looking for the bug in the current version of Guile.

  You can get the version number by invoking the command

```
$ guile --version
Guile 1.4.1
Copyright (c) 1995, 1996, 1997, 2000, 2006 Free Software Foundation
Guile may be distributed under the terms of the GNU General Public License;
certain other uses are permitted as well.  For details, see the file
‘COPYING’, which is included in the Guile distribution.
There is no warranty, to the extent permitted by law.
```

- The type of machine you are using, and the operating system name and version number. On GNU systems, you can get it with ‘`uname`’.

  ```
  $ uname -a
  Linux tortoise 2.2.17 #1 Thu Dec 21 17:29:05 CET 2000 i586 unknown
  ```

- The operands given to the ‘`configure`’ command when Guile was installed. It’s often useful to augment this with the output of the command `guile-config info`.
- A complete list of any modifications you have made to the Guile source. (We may not have time to investigate the bug unless it happens in an unmodified Guile. But if you’ve made modifications and you don’t tell us, you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough—send a context diff for them.

  Adding files of your own, or porting to another machine, is a modification of the source.
- Details of any other deviations from the standard procedure for installing Guile.
- The complete text of any source files needed to reproduce the bug.

  If you can tell us a way to cause the problem without loading any source files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents.
- The precise Guile invocation command line we need to type to reproduce the bug.
- A description of what behavior you observe that you believe is incorrect. For example, "The Guile process gets a fatal signal," or, "The resulting output is as follows, which I think is wrong."

  Of course, if the bug is that Guile gets a fatal signal, then one can’t miss it. But if the bug is incorrect results, the maintainer might fail to notice what is wrong. Why leave it to chance?

  If the manifestation of the bug is a Guile error message, it is important to report the precise text of the error message, and a backtrace showing how the Scheme program arrived at the error.

  This can be done using the procedure `backtrace` in the REPL.
- Check whether any programs you have loaded into Guile, including your ‘`.guile`’ file, set any variables that may affect the functioning of Guile. Also, see whether the problem happens in a freshly started Guile without loading your ‘`.guile`’ file (start Guile with the `-q` switch to prevent loading the init file). If the problem does *not* occur then, you must report the precise contents of any programs that you must load into Guile in order to cause the problem to occur.
- If the problem does depend on an init file or other Scheme programs that are not part of the standard Guile distribution, then you should make sure it is not a bug in those

programs by complaining to their maintainers first. After they verify that they are using Guile in a way that is supposed to work, they should report the bug.

- If you wish to mention something in the Guile source, show the line of code with a few lines of context. Don't just give a line number.

  The line numbers in the development sources might not match those in your sources. It would take extra work for the maintainers to determine what code is in your version at a given line number, and we could not be certain.

- Additional information from a C debugger such as GDB might enable someone to find a problem on a machine which he does not have available. If you don't know how to use GDB, please read the GDB manual—it is not very long, and using GDB is easy. You can find the GDB distribution, including the GDB manual in online form, in most of the same places you can find the Guile distribution. To run Guile under GDB, you should switch to the '`libguile`' subdirectory in which Guile was compiled, then do `gdb guile` or `gdb .libs/guile` (if using GNU Libtool).

  However, you need to think when you collect the additional information if you want it to show what causes the bug.

  For example, many people send just a backtrace, but that is not very useful by itself. A simple backtrace with arguments often conveys little about what is happening inside Guile, because most of the arguments listed in the backtrace are pointers to Scheme objects. The numeric values of these pointers have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are themselves pointers).

3 Programming in Scheme

Guile's core language is Scheme, and an awful lot can be achieved simply by using Guile to write and run Scheme programs. In this part of the manual, we explain how to use Guile in this mode, and describe the tools that Guile provides to help you with script writing, debugging and packaging your programs for distribution.

For readers who are not yet familiar with the Scheme language, this part includes a chapter that presents the basic concepts of the language, and gives references to freely available Scheme tutorial material on the web.

For detailed reference information on the variables, functions etc. that make up Guile's application programming interface (API), See Chapter 5 [API Reference], page 93.

3.1 Basic Ideas in Scheme

In this chapter, we introduce the basic concepts that underpin the elegance and power of the Scheme language.

Readers who already possess a background knowledge of Scheme may happily skip this chapter. For the reader who is new to the language, however, the following discussions on data, procedures, expressions and closure are designed to provide a minimum level of Scheme understanding that is more or less assumed by the reference chapters that follow.

The style of this introductory material aims about halfway between the terse precision of R5RS and the discursive randomness of a Scheme tutorial.

3.1.1 Data Types, Values and Variables

This section discusses the representation of data types and values, what it means for Scheme to be a *latently typed* language, and the role of variables. We conclude by introducing the Scheme syntaxes for defining a new variable, and for changing the value of an existing variable.

3.1.1.1 Latent Typing

The term *latent typing* is used to describe a computer language, such as Scheme, for which you cannot, *in general*, simply look at a program's source code and determine what type of data will be associated with a particular variable, or with the result of a particular expression.

Sometimes, of course, you *can* tell from the code what the type of an expression will be. If you have a line in your program that sets the variable `x` to the numeric value 1, you can be certain that, immediately after that line has executed (and in the absence of multiple threads), `x` has the numeric value 1. Or if you write a procedure that is designed to concatenate two strings, it is likely that the rest of your application will always invoke this procedure with two string parameters, and quite probable that the procedure would go wrong in some way if it was ever invoked with parameters that were not both strings.

Nevertheless, the point is that there is nothing in Scheme which requires the procedure parameters always to be strings, or `x` always to hold a numeric value, and there is no way of declaring in your program that such constraints should always be obeyed. In the same vein, there is no way to declare the expected type of a procedure's return value.

Instead, the types of variables and expressions are only known – in general – at run time. If you *need* to check at some point that a value has the expected type, Scheme provides run time procedures that you can invoke to do so. But equally, it can be perfectly valid for two separate invocations of the same procedure to specify arguments with different types, and to return values with different types.

The next subsection explains what this means in practice, for the ways that Scheme programs use data types, values and variables.

3.1.1.2 Values and Variables

Scheme provides many data types that you can use to represent your data. Primitive types include characters, strings, numbers and procedures. Compound types, which allow a group of primitive and compound values to be stored together, include lists, pairs, vectors and multi-dimensional arrays. In addition, Guile allows applications to define their own data types, with the same status as the built-in standard Scheme types.

As a Scheme program runs, values of all types pop in and out of existence. Sometimes values are stored in variables, but more commonly they pass seamlessly from being the result of one computation to being one of the parameters for the next.

Consider an example. A string value is created because the interpreter reads in a literal string from your program's source code. Then a numeric value is created as the result of calculating the length of the string. A second numeric value is created by doubling the calculated length. Finally the program creates a list with two elements – the doubled length and the original string itself – and stores this list in a program variable.

All of the values involved here – in fact, all values in Scheme – carry their type with them. In other words, every value "knows," at runtime, what kind of value it is. A number, a string, a list, whatever.

A variable, on the other hand, has no fixed type. A variable – `x`, say – is simply the name of a location – a box – in which you can store any kind of Scheme value. So the same variable in a program may hold a number at one moment, a list of procedures the next, and later a pair of strings. The "type" of a variable – insofar as the idea is meaningful at all – is simply the type of whatever value the variable happens to be storing at a particular moment.

3.1.1.3 Defining and Setting Variables

To define a new variable, you use Scheme's `define` syntax like this:

```
(define variable-name value)
```

This makes a new variable called *variable-name* and stores *value* in it as the variable's initial value. For example:

```
;; Make a variable ‘x’ with initial numeric value 1.
(define x 1)

;; Make a variable ‘organization’ with an initial string value.
(define organization "Free Software Foundation")
```

(In Scheme, a semicolon marks the beginning of a comment that continues until the end of the line. So the lines beginning `;;` are comments.)

Changing the value of an already existing variable is very similar, except that `define` is replaced by the Scheme syntax `set!`, like this:

```
(set! variable-name new-value)
```

Remember that variables do not have fixed types, so *new-value* may have a completely different type from whatever was previously stored in the location named by *variable-name*. Both of the following examples are therefore correct.

```
;; Change the value of ‘x’ to 5.
(set! x 5)

;; Change the value of ‘organization’ to the FSF’s street number.
(set! organization 545)
```

In these examples, *value* and *new-value* are literal numeric or string values. In general, however, *value* and *new-value* can be any Scheme expression. Even though we have not yet covered the forms that Scheme expressions can take (see Section 3.1.3 [About Expressions], page 18), you can probably guess what the following `set!` example does...

```
(set! x (+ x 1))
```

(Note: this is not a complete description of `define` and `set!`, because we need to introduce some other aspects of Scheme before the missing pieces can be filled in. If, however, you are already familiar with the structure of Scheme, you may like to read about those missing pieces immediately by jumping ahead to the following references.

- Section 3.1.2.4 [Lambda Alternatives], page 18, to read about an alternative form of the `define` syntax that can be used when defining new procedures.
- Section 5.8.5 [Procedures with Setters], page 230, to read about an alternative form of the `set!` syntax that helps with changing a single value in the depths of a compound data structure.)
- See Section 5.10.3 [Internal Definitions], page 249, to read about using `define` other than at top level in a Scheme program, including a discussion of when it works to use `define` rather than `set!` to change the value of an existing variable.

3.1.2 The Representation and Use of Procedures

This section introduces the basics of using and creating Scheme procedures. It discusses the representation of procedures as just another kind of Scheme value, and shows how procedure invocation expressions are constructed. We then explain how `lambda` is used to create new procedures, and conclude by presenting the various shorthand forms of `define` that can be used instead of writing an explicit `lambda` expression.

3.1.2.1 Procedures as Values

One of the great simplifications of Scheme is that a procedure is just another type of value, and that procedure values can be passed around and stored in variables in exactly the same way as, for example, strings and lists. When we talk about a built-in standard Scheme procedure such as `open-input-file`, what we actually mean is that there is a pre-defined top level variable called `open-input-file`, whose value is a procedure that implements what R5RS says that `open-input-file` should do.

Note that this is quite different from many dialects of Lisp — including Emacs Lisp — in which a program can use the same name with two quite separate meanings: one meaning identifies a Lisp function, while the other meaning identifies a Lisp variable, whose value need have nothing to do with the function that is associated with the first meaning. In these dialects, functions and variables are said to live in different *namespaces*.

In Scheme, on the other hand, all names belong to a single unified namespace, and the variables that these names identify can hold any kind of Scheme value, including procedure values.

One consequence of the "procedures as values" idea is that, if you don't happen to like the standard name for a Scheme procedure, you can change it.

For example, `call-with-current-continuation` is a very important standard Scheme procedure, but it also has a very long name! So, many programmers use the following definition to assign the same procedure value to the more convenient name `call/cc`.

```
(define call/cc call-with-current-continuation)
```

Let's understand exactly how this works. The definition creates a new variable `call/cc`, and then sets its value to the value of the variable `call-with-current-continuation`; the latter value is a procedure that implements the behaviour that R5RS specifies under the name "call-with-current-continuation". So `call/cc` ends up holding this value as well.

Now that `call/cc` holds the required procedure value, you could choose to use `call-with-current-continuation` for a completely different purpose, or just change its value so that you will get an error if you accidentally use `call-with-current-continuation` as a procedure in your program rather than `call/cc`. For example:

```
(set! call-with-current-continuation "Not a procedure any more!")
```

Or you could just leave `call-with-current-continuation` as it was. It's perfectly fine for more than one variable to hold the same procedure value.

3.1.2.2 Simple Procedure Invocation

A procedure invocation in Scheme is written like this:

```
(procedure [arg1 [arg2 ...]])
```

In this expression, *procedure* can be any Scheme expression whose value is a procedure. Most commonly, however, *procedure* is simply the name of a variable whose value is a procedure.

For example, `string-append` is a standard Scheme procedure whose behaviour is to concatenate together all the arguments, which are expected to be strings, that it is given. So the expression

```
(string-append "/home" "/" "andrew")
```

is a procedure invocation whose result is the string value `"/home/andrew"`.

Similarly, `string-length` is a standard Scheme procedure that returns the length of a single string argument, so

```
(string-length "abc")
```

is a procedure invocation whose result is the numeric value 3.

Each of the parameters in a procedure invocation can itself be any Scheme expression. Since a procedure invocation is itself a type of expression, we can put these two examples together to get

```
(string-length (string-append "/home" "/" "andrew"))
```

— a procedure invocation whose result is the numeric value 12.

(You may be wondering what happens if the two examples are combined the other way round. If we do this, we can make a procedure invocation expression that is *syntactically* correct:

```
(string-append "/home" (string-length "abc"))
```

but when this expression is executed, it will cause an error, because the result of `(string-length "abc")` is a numeric value, and `string-append` is not designed to accept a numeric value as one of its arguments.)

3.1.2.3 Creating and Using a New Procedure

Scheme has lots of standard procedures, and Guile provides all of these via predefined top level variables. All of these standard procedures are documented in the later chapters of this reference manual.

Before very long, though, you will want to create new procedures that encapsulate aspects of your own applications' functionality. To do this, you can use the famous `lambda` syntax.

For example, the value of the following Scheme expression

```
(lambda (name address) expression ...)
```

is a newly created procedure that takes two arguments: `name` and `address`. The behaviour of the new procedure is determined by the sequence of *expression*s in the *body* of the procedure definition. (Typically, these *expression*s would use the arguments in some way, or else there wouldn't be any point in giving them to the procedure.) When invoked, the new procedure returns a value that is the value of the last *expression* in the procedure body.

To make things more concrete, let's suppose that the two arguments are both strings, and that the purpose of this procedure is to form a combined string that includes these arguments. Then the full lambda expression might look like this:

```
(lambda (name address)
  (string-append "Name=" name ":Address=" address))
```

We noted in the previous subsection that the *procedure* part of a procedure invocation expression can be any Scheme expression whose value is a procedure. But that's exactly what a lambda expression is! So we can use a lambda expression directly in a procedure invocation, like this:

```
((lambda (name address)
   (string-append "Name=" name ":Address=" address))
 "FSF"
 "Cambridge")
```

This is a valid procedure invocation expression, and its result is the string `"Name=FSF:Address=Cambridge"`.

It is more common, though, to store the procedure value in a variable —

```
(define make-combined-string
  (lambda (name address)
    (string-append "Name=" name ":Address=" address)))
```

— and then to use the variable name in the procedure invocation:

```
(make-combined-string "FSF" "Cambridge")
```

Which has exactly the same result.

It's important to note that procedures created using `lambda` have exactly the same status as the standard built in Scheme procedures, and can be invoked, passed around, and stored in variables in exactly the same ways.

3.1.2.4 Lambda Alternatives

Since it is so common in Scheme programs to want to create a procedure and then store it in a variable, there is an alternative form of the `define` syntax that allows you to do just that.

A `define` expression of the form

```
(define (name [arg1 [arg2 ...]])
  expression ...)
```

is exactly equivalent to the longer form

```
(define name
  (lambda ([arg1 [arg2 ...]])
    expression ...))
```

So, for example, the definition of `make-combined-string` in the previous subsection could equally be written:

```
(define (make-combined-string name address)
  (string-append "Name=" name ":Address=" address))
```

This kind of procedure definition creates a procedure that requires exactly the expected number of arguments. There are two further forms of the `lambda` expression, which create a procedure that can accept a variable number of arguments:

```
(lambda (arg1 ... . args) expression ...)

(lambda args expression ...)
```

The corresponding forms of the alternative `define` syntax are:

```
(define (name arg1 ... . args) expression ...)

(define (name . args) expression ...)
```

For details on how these forms work, see See Section 5.8.1 [Lambda], page 225.

(It could be argued that the alternative `define` forms are rather confusing, especially for newcomers to the Scheme language, as they hide both the role of `lambda` and the fact that procedures are values that are stored in variables in the some way as any other kind of value. On the other hand, they are very convenient, and they are also a good example of another of Scheme's powerful features: the ability to specify arbitrary syntactic transformations at run time, which can be applied to subsequently read input.)

3.1.3 Expressions and Evaluation

So far, we have met expressions that *do* things, such as the `define` expressions that create and initialize new variables, and we have also talked about expressions that have *values*, for example the value of the procedure invocation expression:

```
(string-append "/home" "/" "andrew")
```

but we haven't yet been precise about what causes an expression like this procedure invocation to be reduced to its "value", or how the processing of such expressions relates to the execution of a Scheme program as a whole.

This section clarifies what we mean by an expression's value, by introducing the idea of *evaluation*. It discusses the side effects that evaluation can have, explains how each of the various types of Scheme expression is evaluated, and describes the behaviour and use of the Guile REPL as a mechanism for exploring evaluation. The section concludes with a very brief summary of Scheme's common syntactic expressions.

3.1.3.1 Evaluating Expressions and Executing Programs

In Scheme, the process of executing an expression is known as *evaluation*. Evaluation has two kinds of result:

- the *value* of the evaluated expression
- the *side effects* of the evaluation, which consist of any effects of evaluating the expression that are not represented by the value.

Of the expressions that we have met so far, `define` and `set!` expressions have side effects — the creation or modification of a variable — but no value; `lambda` expressions have values — the newly constructed procedures — but no side effects; and procedure invocation expressions, in general, have either values, or side effects, or both.

It is tempting to try to define more intuitively what we mean by "value" and "side effects", and what the difference between them is. In general, though, this is extremely difficult. It is also unnecessary; instead, we can quite happily define the behaviour of a Scheme program by specifying how Scheme executes a program as a whole, and then by describing the value and side effects of evaluation for each type of expression individually.

So, some[1] definitions...

- A Scheme program consists of a sequence of expressions.
- A Scheme interpreter executes the program by evaluating these expressions in order, one by one.
- An expression can be
  - a piece of literal data, such as a number `2.3` or a string `"Hello world!"`
  - a variable name
  - a procedure invocation expression
  - one of Scheme's special syntactic expressions.

The following subsections describe how each of these types of expression is evaluated.

Evaluating Literal Data

When a literal data expression is evaluated, the value of the expression is simply the value that the expression describes. The evaluation of a literal data expression has no side effects.

So, for example,

[1] These definitions are approximate. For the whole and detailed truth, see See ⟨undefined⟩ [Formal syntax and semantics], page ⟨undefined⟩.

- the value of the expression `"abc"` is the string value `"abc"`
- the value of the expression `3+4i` is the complex number 3 + 4i
- the value of the expression `#(1 2 3)` is a three-element vector containing the numeric values 1, 2 and 3.

For any data type which can be expressed literally like this, the syntax of the literal data expression for that data type — in other words, what you need to write in your code to indicate a literal value of that type — is known as the data type's *read syntax*. This manual specifies the read syntax for each such data type in the section that describes that data type.

Some data types do not have a read syntax. Procedures, for example, cannot be expressed as literal data; they must be created using a `lambda` expression (see Section 3.1.2.3 [Creating a Procedure], page 17) or implicitly using the shorthand form of `define` (see Section 3.1.2.4 [Lambda Alternatives], page 18).

Evaluating a Variable Reference

When an expression that consists simply of a variable name is evaluated, the value of the expression is the value of the named variable. The evaluation of a variable reference expression has no side effects.

So, after

```
(define key "Paul Evans")
```

the value of the expression `key` is the string value `"Paul Evans"`. If *key* is then modified by

```
(set! key 3.74)
```

the value of the expression `key` is the numeric value 3.74.

If there is no variable with the specified name, evaluation of the variable reference expression signals an error.

Evaluating a Procedure Invocation Expression

This is where evaluation starts getting interesting! As already noted, a procedure invocation expression has the form

```
(procedure [arg1 [arg2 ...]])
```

where *procedure* must be an expression whose value, when evaluated, is a procedure.

The evaluation of a procedure invocation expression like this proceeds by

- evaluating individually the expressions *procedure*, *arg1*, *arg2*, and so on
- calling the procedure that is the value of the *procedure* expression with the list of values obtained from the evaluations of *arg1*, *arg2* etc. as its parameters.

For a procedure defined in Scheme, "calling the procedure with the list of values as its parameters" means binding the values to the procedure's formal parameters and then evaluating the sequence of expressions that make up the body of the procedure definition. The value of the procedure invocation expression is the value of the last evaluated expression in the procedure body. The side effects of calling the procedure are the combination of the side effects of the sequence of evaluations of expressions in the procedure body.

For a built-in procedure, the value and side-effects of calling the procedure are best described by that procedure's documentation.

Note that the complete side effects of evaluating a procedure invocation expression consist not only of the side effects of the procedure call, but also of any side effects of the preceding evaluation of the expressions *procedure*, *arg1*, *arg2*, and so on.

To illustrate this, let's look again at the procedure invocation expression:

```
(string-length (string-append "/home" "/" "andrew"))
```

In the outermost expression, *procedure* is `string-length` and *arg1* is `(string-append "/home" "/" "andrew")`.

- Evaluation of `string-length`, which is a variable, gives a procedure value that implements the expected behaviour for "string-length".
- Evaluation of `(string-append "/home" "/" "andrew")`, which is another procedure invocation expression, means evaluating each of
  - `string-append`, which gives a procedure value that implements the expected behaviour for "string-append"
  - `"/home"`, which gives the string value `"/home"`
  - `"/"`, which gives the string value `"/"`
  - `"andrew"`, which gives the string value `"andrew"`

  and then invoking the procedure value with this list of string values as its arguments. The resulting value is a single string value that is the concatenation of all the arguments, namely `"/home/andrew"`.

In the evaluation of the outermost expression, the interpreter can now invoke the procedure value obtained from *procedure* with the value obtained from *arg1* as its arguments. The resulting value is a numeric value that is the length of the argument string, which is 12.

Evaluating Special Syntactic Expressions

When a procedure invocation expression is evaluated, the procedure and *all* the argument expressions must be evaluated before the procedure can be invoked. Special syntactic expressions are special because they are able to manipulate their arguments in an unevaluated form, and can choose whether to evaluate any or all of the argument expressions.

Why is this needed? Consider a program fragment that asks the user whether or not to delete a file, and then deletes the file if the user answers yes.

```
(if (string=? (read-answer "Should I delete this file?")
              "yes")
    (delete-file file))
```

If the outermost `(if ...)` expression here was a procedure invocation expression, the expression `(delete-file file)`, whose side effect is to actually delete a file, would already have been evaluated before the `if` procedure even got invoked! Clearly this is no use — the whole point of an `if` expression is that the *consequent* expression is only evaluated if the condition of the `if` expression is "true".

Therefore `if` must be special syntax, not a procedure. Other special syntaxes that we have already met are `define`, `set!` and `lambda`. `define` and `set!` are syntax because they need to know the variable *name* that is given as the first argument in a `define` or `set!` expression, not that variable's value. `lambda` is syntax because it does not immediately evaluate the expressions that define the procedure body; instead it creates a procedure object that incorporates these expressions so that they can be evaluated in the future, when that procedure is invoked.

The rules for evaluating each special syntactic expression are specified individually for each special syntax. For a summary of standard special syntax, see See Section 3.1.3.4 [Syntax Summary], page 23.

3.1.3.2 Tail calls

Scheme is “properly tail recursive”, meaning that tail calls or recursions from certain contexts do not consume stack space or other resources and can therefore be used on arbitrarily large data or for an arbitrarily long calculation. Consider for example,

```
(define (foo n)
  (display n)
  (newline)
  (foo (1+ n)))

(foo 1)
⊣
1
2
3
...
```

`foo` prints numbers infinitely, starting from the given *n*. It's implemented by printing *n* then recursing to itself to print $n + 1$ and so on. This recursion is a tail call, it's the last thing done, and in Scheme such tail calls can be made without limit.

Or consider a case where a value is returned, a version of the SRFI-1 `last` function (see Section 6.4.3.3 [SRFI-1 Selectors], page 426) returning the last element of a list,

```
(define (my-last lst)
  (if (null? (cdr lst))
      (car lst)
      (my-last (cdr lst))))

(my-last '(1 2 3)) ⇒ 3
```

If the list has more than one element, `my-last` applies itself to the `cdr`. This recursion is a tail call, there's no code after it, and the return value is the return value from that call. In Scheme this can be used on an arbitrarily long list argument.

A proper tail call is only available from certain contexts, namely the following special form positions,

- `and` — last expression
- `begin` — last expression
- `case` — last expression in each clause
- `cond` — last expression in each clause, and the call to a `=>` procedure is a tail call
- `do` — last result expression

- `if` — “true” and “false” leg expressions
- `lambda` — last expression in body
- `let`, `let*`, `letrec`, `let-syntax`, `letrec-syntax` — last expression in body
- `or` — last expression

The following core functions make tail calls,

- `apply` — tail call to given procedure
- `call-with-current-continuation` — tail call to the procedure receiving the new continuation
- `call-with-values` — tail call to the values-receiving procedure
- `eval` — tail call to evaluate the form
- `string-any`, `string-every` — tail call to predicate on the last character (if that point is reached)

The above are just core functions and special forms. Tail calls in other modules are described with the relevant documentation, for example SRFI-1 `any` and `every` (see Section 6.4.3.7 [SRFI-1 Searching], page 432).

It will be noted there are a lot of places which could potentially be tail calls, for instance the last call in a `for-each`, but only those explicitly described are guaranteed.

3.1.3.3 Using the Guile REPL

If you start Guile without specifying a particular program for it to execute, Guile enters its standard Read Evaluate Print Loop — or *REPL* for short. In this mode, Guile repeatedly reads in the next Scheme expression that the user types, evaluates it, and prints the resulting value.

The REPL is a useful mechanism for exploring the evaluation behaviour described in the previous subsection. If you type `string-append`, for example, the REPL replies `#<primitive-procedure string-append>`, illustrating the relationship between the variable `string-append` and the procedure value stored in that variable.

In this manual, the notation ⇒ is used to mean “evaluates to”. Wherever you see an example of the form

```
expression
⇒
result
```

feel free to try it out yourself by typing *expression* into the REPL and checking that it gives the expected *result*.

3.1.3.4 Summary of Common Syntax

This subsection lists the most commonly used Scheme syntactic expressions, simply so that you will recognize common special syntax when you see it. For a full description of each of these syntaxes, follow the appropriate reference.

`lambda` (see Section 5.8.1 [Lambda], page 225) is used to construct procedure objects.

`define` (see Section 5.10.1 [Top Level], page 247) is used to create a new variable and set its initial value.

`set!` (see Section 5.10.1 [Top Level], page 247) is used to modify an existing variable's value.

`let`, `let*` and `letrec` (see Section 5.10.2 [Local Bindings], page 248) create an inner lexical environment for the evaluation of a sequence of expressions, in which a specified set of local variables is bound to the values of a corresponding set of expressions. For an introduction to environments, see See Section 3.1.4 [About Closure], page 24.

`begin` (see Section 5.11.1 [begin], page 251) executes a sequence of expressions in order and returns the value of the last expression. Note that this is not the same as a procedure which returns its last argument, because the evaluation of a procedure invocation expression does not guarantee to evaluate the arguments in order.

`if` and `cond` (see Section 5.11.2 [if cond case], page 251) provide conditional evaluation of argument expressions depending on whether one or more conditions evaluate to "true" or "false".

`case` (see Section 5.11.2 [if cond case], page 251) provides conditional evaluation of argument expressions depending on whether a variable has one of a specified group of values.

`and` (see Section 5.11.3 [and or], page 252) executes a sequence of expressions in order until either there are no expressions left, or one of them evaluates to "false".

`or` (see Section 5.11.3 [and or], page 252) executes a sequence of expressions in order until either there are no expressions left, or one of them evaluates to "true".

3.1.4 The Concept of Closure

The concept of *closure* is the idea that a lambda expression "captures" the variable bindings that are in lexical scope at the point where the lambda expression occurs. The procedure created by the lambda expression can refer to and mutate the captured bindings, and the values of those bindings persist between procedure calls.

This section explains and explores the various parts of this idea in more detail.

3.1.4.1 Names, Locations, Values and Environments

We said earlier that a variable name in a Scheme program is associated with a location in which any kind of Scheme value may be stored. (Incidentally, the term "vcell" is often used in Lisp and Scheme circles as an alternative to "location".) Thus part of what we mean when we talk about "creating a variable" is in fact establishing an association between a name, or identifier, that is used by the Scheme program code, and the variable location to which that name refers. Although the value that is stored in that location may change, the location to which a given name refers is always the same.

We can illustrate this by breaking down the operation of the `define` syntax into three parts: `define`

- creates a new location
- establishes an association between that location and the name specified as the first argument of the `define` expression
- stores in that location the value obtained by evaluating the second argument of the `define` expression.

A collection of associations between names and locations is called an *environment*. When you create a top level variable in a program using `define`, the name-location association for that variable is added to the “top level” environment. The “top level” environment also includes name-location associations for all the procedures that are supplied by standard Scheme.

It is also possible to create environments other than the top level one, and to create variable bindings, or name-location associations, in those environments. This ability is a key ingredient in the concept of closure; the next subsection shows how it is done.

3.1.4.2 Local Variables and Environments

We have seen how to create top level variables using the `define` syntax (see Section 3.1.1.3 [Definition], page 14). It is often useful to create variables that are more limited in their scope, typically as part of a procedure body. In Scheme, this is done using the `let` syntax, or one of its modified forms `let*` and `letrec`. These syntaxes are described in full later in the manual (see Section 5.10.2 [Local Bindings], page 248). Here our purpose is to illustrate their use just enough that we can see how local variables work.

For example, the following code uses a local variable `s` to simplify the computation of the area of a triangle given the lengths of its three sides.

```
(define a 5.3)
(define b 4.7)
(define c 2.8)

(define area
  (let ((s (/ (+ a b c) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

The effect of the `let` expression is to create a new environment and, within this environment, an association between the name `s` and a new location whose initial value is obtained by evaluating `(/ (+ a b c) 2)`. The expressions in the body of the `let`, namely `(sqrt (* s (- s a) (- s b) (- s c)))`, are then evaluated in the context of the new environment, and the value of the last expression evaluated becomes the value of the whole `let` expression, and therefore the value of the variable `area`.

3.1.4.3 Environment Chaining

In the example of the previous subsection, we glossed over an important point. The body of the `let` expression in that example refers not only to the local variable `s`, but also to the top level variables `a`, `b`, `c` and `sqrt`. (`sqrt` is the standard Scheme procedure for calculating a square root.) If the body of the `let` expression is evaluated in the context of the *local* `let` environment, how does the evaluation get at the values of these top level variables?

The answer is that the local environment created by a `let` expression automatically has a reference to its containing environment — in this case the top level environment — and that the Scheme interpreter automatically looks for a variable binding in the containing environment if it doesn’t find one in the local environment. More generally, every environment except for the top level one has a reference to its containing environment, and the interpreter keeps searching back up the chain of environments — from most local to top level — until it either finds a variable binding for the required identifier or exhausts the chain.

This description also determines what happens when there is more than one variable binding with the same name. Suppose, continuing the example of the previous subsection, that there was also a pre-existing top level variable `s` created by the expression:

```
(define s "Some beans, my lord!")
```

Then both the top level environment and the local `let` environment would contain bindings for the name `s`. When evaluating code within the `let` body, the interpreter looks first in the local `let` environment, and so finds the binding for `s` created by the `let` syntax. Even though this environment has a reference to the top level environment, which also has a binding for `s`, the interpreter doesn't get as far as looking there. When evaluating code outside the `let` body, the interpreter looks up variable names in the top level environment, so the name `s` refers to the top level variable.

Within the `let` body, the binding for `s` in the local environment is said to *shadow* the binding for `s` in the top level environment.

3.1.4.4 Lexical Scope

The rules that we have just been describing are the details of how Scheme implements "lexical scoping". This subsection takes a brief diversion to explain what lexical scope means in general and to present an example of non-lexical scoping.

"Lexical scope" in general is the idea that

- an identifier at a particular place in a program always refers to the same variable location — where "always" means "every time that the containing expression is executed", and that
- the variable location to which it refers can be determined by static examination of the source code context in which that identifier appears, without having to consider the flow of execution through the program as a whole.

In practice, lexical scoping is the norm for most programming languages, and probably corresponds to what you would intuitively consider to be "normal". You may even be wondering how the situation could possibly — and usefully — be otherwise. To demonstrate that another kind of scoping is possible, therefore, and to compare it against lexical scoping, the following subsection presents an example of non-lexical scoping and examines in detail how its behavior differs from the corresponding lexically scoped code.

An Example of Non-Lexical Scoping

To demonstrate that non-lexical scoping does exist and can be useful, we present the following example from Emacs Lisp, which is a "dynamically scoped" language.

```
(defvar currency-abbreviation "USD")

(defun currency-string (units hundredths)
  (concat currency-abbreviation
          (number-to-string units)
          "."
          (number-to-string hundredths)))

(defun french-currency-string (units hundredths)
  (let ((currency-abbreviation "FRF"))
        (currency-string units hundredths)))
```

The question to focus on here is: what does the identifier `currency-abbreviation` refer to in the `currency-string` function? The answer, in Emacs Lisp, is that all variable bindings go onto a single stack, and that `currency-abbreviation` refers to the topmost binding from that stack which has the name "currency-abbreviation". The binding that is created by the `defvar` form, to the value `"USD"`, is only relevant if none of the code that calls `currency-string` rebinds the name "currency-abbreviation" in the meanwhile.

The second function `french-currency-string` works precisely by taking advantage of this behaviour. It creates a new binding for the name "currency-abbreviation" which overrides the one established by the `defvar` form.

```
;; Note!  This is Emacs Lisp evaluation, not Scheme!
(french-currency-string 33 44)
⇒
"FRF33.44"
```

Now let's look at the corresponding, *lexically scoped* Scheme code:

```
(define currency-abbreviation "USD")

(define (currency-string units hundredths)
  (string-append currency-abbreviation
                 (number->string units)
                 "."
                 (number->string hundredths)))

(define (french-currency-string units hundredths)
  (let ((currency-abbreviation "FRF"))
    (currency-string units hundredths)))
```

According to the rules of lexical scoping, the `currency-abbreviation` in `currency-string` refers to the variable location in the innermost environment at that point in the code which has a binding for `currency-abbreviation`, which is the variable location in the top level environment created by the preceding `(define currency-abbreviation ...)` expression.

In Scheme, therefore, the `french-currency-string` procedure does not work as intended. The variable binding that it creates for "currency-abbreviation" is purely local to the code that forms the body of the `let` expression. Since this code doesn't directly use the name "currency-abbreviation" at all, the binding is pointless.

```
(french-currency-string 33 44)
⇒
"USD33.44"
```

This begs the question of how the Emacs Lisp behaviour can be implemented in Scheme. In general, this is a design question whose answer depends upon the problem that is being addressed. In this case, the best answer may be that `currency-string` should be redesigned so that it can take an optional third argument. This third argument, if supplied, is interpreted as a currency abbreviation that overrides the default.

It is possible to change `french-currency-string` so that it mostly works without changing `currency-string`, but the fix is inelegant, and susceptible to interrupts that could leave the `currency-abbreviation` variable in the wrong state:

```
(define (french-currency-string units hundredths)
  (set! currency-abbreviation "FRF")
  (let ((result (currency-string units hundredths)))
    (set! currency-abbreviation "USD")
    result))
```

The key point here is that the code does not create any local binding for the identifier `currency-abbreviation`, so all occurrences of this identifier refer to the top level variable.

3.1.4.5 Closure

Consider a `let` expression that doesn't contain any `lambda`s:

```
(let ((s (/ (+ a b c) 2)))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

When the Scheme interpreter evaluates this, it

- creates a new environment with a reference to the environment that was current when it encountered the `let`
- creates a variable binding for `s` in the new environment, with value given by `(/ (+ a b c) 2)`
- evaluates the expression in the body of the `let` in the context of the new local environment, and remembers the value `V`
- forgets the local environment
- continues evaluating the expression that contained the `let`, using the value `V` as the value of the `let` expression, in the context of the containing environment.

After the `let` expression has been evaluated, the local environment that was created is simply forgotten, and there is no longer any way to access the binding that was created in this environment. If the same code is evaluated again, it will follow the same steps again, creating a second new local environment that has no connection with the first, and then forgetting this one as well.

If the `let` body contains a `lambda` expression, however, the local environment is *not* forgotten. Instead, it becomes associated with the procedure that is created by the `lambda` expression, and is reinstated every time that that procedure is called. In detail, this works as follows.

- When the Scheme interpreter evaluates a `lambda` expression, to create a procedure object, it stores the current environment as part of the procedure definition.
- Then, whenever that procedure is called, the interpreter reinstates the environment that is stored in the procedure definition and evaluates the procedure body within the context of that environment.

The result is that the procedure body is always evaluated in the context of the environment that was current when the procedure was created.

This is what is meant by *closure*. The next few subsections present examples that explore the usefulness of this concept.

3.1.4.6 Example 1: A Serial Number Generator

This example uses closure to create a procedure with a variable binding that is private to the procedure, like a local variable, but whose value persists between procedure calls.

```
(define (make-serial-number-generator)
  (let ((current-serial-number 0))
    (lambda ()
      (set! current-serial-number (+ current-serial-number 1))
      current-serial-number)))

(define entry-sn-generator (make-serial-number-generator))

(entry-sn-generator)
⇒
1

(entry-sn-generator)
⇒
2
```

When `make-serial-number-generator` is called, it creates a local environment with a binding for `current-serial-number` whose initial value is 0, then, within this environment, creates a procedure. The local environment is stored within the created procedure object and so persists for the lifetime of the created procedure.

Every time the created procedure is invoked, it increments the value of the `current-serial-number` binding in the captured environment and then returns the current value.

Note that `make-serial-number-generator` can be called again to create a second serial number generator that is independent of the first. Every new invocation of `make-serial-number-generator` creates a new local `let` environment and returns a new procedure object with an association to this environment.

3.1.4.7 Example 2: A Shared Persistent Variable

This example uses closure to create two procedures, `get-balance` and `deposit`, that both refer to the same captured local environment so that they can both access the `balance` variable binding inside that environment. The value of this variable binding persists between calls to either procedure.

Note that the captured `balance` variable binding is private to these two procedures: it is not directly accessible to any other code. It can only be accessed indirectly via `get-balance` or `deposit`, as illustrated by the `withdraw` procedure.

```
(define get-balance #f)
(define deposit #f)

(let ((balance 0))
  (set! get-balance
        (lambda ()
          balance))
  (set! deposit
        (lambda (amount)
          (set! balance (+ balance amount))
          balance)))

(define (withdraw amount)
  (deposit (- amount)))

(get-balance)
⇒
0

(deposit 50)
⇒
50

(withdraw 75)
⇒
-25
```

An important detail here is that the `get-balance` and `deposit` variables must be set up by `define`ing them at top level and then `set!`ing their values inside the `let` body. Using `define` within the `let` body would not work: this would create variable bindings within the local `let` environment that would not be accessible at top level.

3.1.4.8 Example 3: The Callback Closure Problem

A frequently used programming model for library code is to allow an application to register a callback function for the library to call when some particular event occurs. It is often useful for the application to make several such registrations using the same callback function, for example if several similar library events can be handled using the same application code, but the need then arises to distinguish the callback function calls that are associated with one callback registration from those that are associated with different callback registrations.

In languages without the ability to create functions dynamically, this problem is usually solved by passing a `user_data` parameter on the registration call, and including the value of this parameter as one of the parameters on the callback function. Here is an example of declarations using this solution in C:

```
typedef void (event_handler_t) (int event_type,
                                void *user_data);

void register_callback (int event_type,
                        event_handler_t *handler,
                        void *user_data);
```

In Scheme, closure can be used to achieve the same functionality without requiring the library code to store a `user-data` for each callback registration.

```
;; In the library:

(define (register-callback event-type handler-proc)
  ...)
```

```
;; In the application:

(define (make-handler event-type user-data)
  (lambda ()
    ...
    <code referencing event-type and user-data>
    ...))

(register-callback event-type
                   (make-handler event-type ...))
```

As far as the library is concerned, `handler-proc` is a procedure with no arguments, and all the library has to do is call it when the appropriate event occurs. From the application's point of view, though, the handler procedure has used closure to capture an environment that includes all the context that the handler code needs — `event-type` and `user-data` — to handle the event correctly.

3.1.4.9 Example 4: Object Orientation

Closure is the capture of an environment, containing persistent variable bindings, within the definition of a procedure or a set of related procedures. This is rather similar to the idea in some object oriented languages of encapsulating a set of related data variables inside an "object", together with a set of "methods" that operate on the encapsulated data. The following example shows how closure can be used to emulate the ideas of objects, methods and encapsulation in Scheme.

```
(define (make-account)
  (let ((balance 0))
    (define (get-balance)
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (withdraw amount)
      (deposit (- amount)))

    (lambda args
      (apply
        (case (car args)
          ((get-balance) get-balance)
          ((deposit) deposit)
          ((withdraw) withdraw)
          (else (error "Invalid method!")))
        (cdr args)))))
```

Each call to `make-account` creates and returns a new procedure, created by the expression in the example code that begins "(lambda args".

```
(define my-account (make-account))
```

```
my-account
⇒
#<procedure args>
```

This procedure acts as an account object with methods `get-balance`, `deposit` and `withdraw`. To apply one of the methods to the account, you call the procedure with a symbol indicating the required method as the first parameter, followed by any other parameters that are required by that method.

```
(my-account 'get-balance)
⇒
0

(my-account 'withdraw 5)
⇒
-5

(my-account 'deposit 396)
⇒
391

(my-account 'get-balance)
⇒
391
```

Note how, in this example, both the current balance and the helper procedures `get-balance`, `deposit` and `withdraw`, used to implement the guts of the account object's methods, are all stored in variable bindings within the private local environment captured by the `lambda` expression that creates the account object procedure.

3.2 Guile's Implementation of Scheme

Guile's core language is Scheme, which is specified and described in the series of reports known as *RnRS*. *RnRS* is shorthand for the *Revised^n Report on the Algorithmic Language Scheme*. The current latest revision of RnRS is version 5 (see ⟨undefined⟩ [Top], page ⟨undefined⟩), and Guile 1.4 is fully compliant with the Scheme specification in this revision.

But Guile, like most Scheme implementations, also goes beyond R5RS in many ways, because R5RS does not give specifications (or even recommendations) regarding many issues that are important in practical programming. Some of the areas where Guile extends R5RS are:

- Guile's interactive documentation system
- Guile's support for POSIX-compliant network programming
- GOOPS – Guile's framework for object oriented programming.

3.3 Guile Scripting

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

3.3.1 The Top of a Script File

The first line of a Guile script must tell the operating system to use Guile to evaluate the script, and then tell Guile how to go about doing that. Here is the simplest case:

- The first two characters of the file must be '`#!`'.

  The operating system interprets this to mean that the rest of the line is the name of an executable that can interpret the script. Guile, however, interprets these characters as the beginning of a multi-line comment, terminated by the characters '`!#`' on a line by themselves. (This is an extension to the syntax described in R5RS, added to support shell scripts.)
- Immediately after those two characters must come the full pathname to the Guile interpreter. On most systems, this would be '`/usr/local/bin/guile`'.
- Then must come a space, followed by a command-line argument to pass to Guile; this should be '`-s`'. This switch tells Guile to run a script, instead of soliciting the user for input from the terminal. There are more elaborate things one can do here; see Section 3.3.3 [The Meta Switch], page 35.
- Follow this with a newline.
- The second line of the script should contain only the characters '`!#`' — just like the top of the file, but reversed. The operating system never reads this far, but Guile treats this as the end of the comment begun on the first line by the '`#!`' characters.
- The rest of the file should be a Scheme program.

Guile reads the program, evaluating expressions in the order that they appear. Upon reaching the end of the file, Guile exits.

3.3.2 Invoking Guile

Here we describe Guile's command-line processing in detail. Guile processes its arguments from left to right, recognizing the switches described below. For examples, see Section 3.3.5 [Scripting Examples], page 37.

`-s` *`script arg...`*
: Read and evaluate Scheme source code from the file *script*, as the `load` function would. After loading *script*, exit. Any command-line arguments *arg...* following *script* become the script's arguments; the `command-line` function returns a list of strings of the form `(`*`script arg...`*`)`.

`-c` *`expr arg...`*
: Evaluate *expr* as Scheme code, and then exit. Any command-line arguments *arg...* following *expr* become command-line arguments; the `command-line` function returns a list of strings of the form `(`*`guile arg...`*`)`, where *guile* is the path of the Guile executable.

`--` *`arg...`*
: Run interactively, prompting the user for expressions and evaluating them. Any command-line arguments *arg...* following the `--` become command-line arguments for the interactive session; the `command-line` function returns a list of strings of the form `(`*`guile arg...`*`)`, where *guile* is the path of the Guile executable.

`-L` *`directory`*
: Add *directory* to the front of Guile's module load path. The given directories are searched in the order given on the command line and before any directories in the GUILE_LOAD_PATH environment variable. Paths added here are *not* in effect during execution of the user's '`.guile`' file.

`-l` *`file`*
: Load Scheme source code from *file*, and continue processing the command line.

`-e` *`function`*
: Make *function* the *entry point* of the script. After loading the script file (with `-s`) or evaluating the expression (with `-c`), apply *function* to a list containing the program name and the command-line arguments — the list provided by the `command-line` function.

  A `-e` switch can appear anywhere in the argument list, but Guile always invokes the *function* as the *last* action it performs. This is weird, but because of the way script invocation works under POSIX, the `-s` option must always come last in the list.

  The *function* is most often a simple symbol that names a function that is defined in the script. It can also be of the form `(@` *`module-name symbol`*`)` and in that case, the symbol is looked up in the module named *module-name*.

  For compatibility with some versions of Guile 1.4, you can also use the form `(symbol ...)` (that is, a list of only symbols that doesn't start with `@`), which is equivalent to `(@ (symbol ...) main)`, or `(symbol ...) symbol` (that is, a list of only symbols followed by a symbol), which is equivalent to `(@ (symbol ...) symbol)`. We recommend to use the equivalent forms directly since they corresponf to the `(@ ...)` read syntax that can be used in normal code, See Section 5.16.3.2 [Using Guile Modules], page 305.

  See Section 3.3.5 [Scripting Examples], page 37.

`-ds`
: Treat a final `-s` option as if it occurred at this point in the command line; load the script here.

  This switch is necessary because, although the POSIX script invocation mechanism effectively requires the `-s` option to appear last, the programmer may well want to run the script before other actions requested on the command line. For examples, see Section 3.3.5 [Scripting Examples], page 37.

`\`
: Read more command-line arguments, starting from the second line of the script file. See Section 3.3.3 [The Meta Switch], page 35.

`--emacs`
: Assume Guile is running as an inferior process of Emacs, and use a special protocol to communicate with Emacs's Guile interaction mode. This switch sets the global variable use-emacs-interface to `#t`.

This switch is still experimental.

`--use-srfi=`*`list`*
: The option `--use-srfi` expects a comma-separated list of numbers, each representing a SRFI number to be loaded into the interpreter before starting evaluating a script file or the REPL. Additionally, the feature identifier for the loaded SRFIs is recognized by 'cond-expand' when using this option.

```
guile --use-srfi=8,13
```

`--debug`
: Start with the debugging evaluator and enable backtraces. Using the debugging evaluator will give you better error messages but it will slow down execution. By default, the debugging evaluator is only used when entering an interactive session. When executing a script with `-s` or `-c`, the normal, faster evaluator is used by default.

`--no-debug`
: Do not use the debugging evaluator, even when entering an interactive session.

`-h`, `--help`
: Display help on invoking Guile, and then exit.

`-v`, `--version`
: Display the current version of Guile, and then exit.

3.3.3 The Meta Switch

Guile's command-line switches allow the programmer to describe reasonably complicated actions in scripts. Unfortunately, the POSIX script invocation mechanism only allows one argument to appear on the '`#!`' line after the path to the Guile executable, and imposes arbitrary limits on that argument's length. Suppose you wrote a script starting like this:

```
#!/usr/local/bin/guile -e main -s
!#
(define (main args)
  (map (lambda (arg) (display arg) (display " "))
       (cdr args))
  (newline))
```

The intended meaning is clear: load the file, and then call `main` on the command-line arguments. However, the system will treat everything after the Guile path as a single argument — the string `"-e main -s"` — which is not what we want.

As a workaround, the meta switch `\` allows the Guile programmer to specify an arbitrary number of options without patching the kernel. If the first argument to Guile is `\`, Guile will open the script file whose name follows the `\`, parse arguments starting from the file's second line (according to rules described below), and substitute them for the `\` switch.

Working in concert with the meta switch, Guile treats the characters '`#!`' as the beginning of a comment which extends through the next line containing only the characters '`!#`'. This sort of comment may appear anywhere in a Guile program, but it is most useful at the top of a file, meshing magically with the POSIX script invocation mechanism.

Thus, consider a script named '`/u/jimb/ekko`' which starts like this:

```
#!/usr/local/bin/guile \
-e main -s
!#
(define (main args)
        (map (lambda (arg) (display arg) (display " "))
             (cdr args))
        (newline))
```

Suppose a user invokes this script as follows:

```
$ /u/jimb/ekko a b c
```

Here's what happens:

- the operating system recognizes the '`#!`' token at the top of the file, and rewrites the command line to:

  ```
  /usr/local/bin/guile \ /u/jimb/ekko a b c
  ```

  This is the usual behavior, prescribed by POSIX.
- When Guile sees the first two arguments, `\ /u/jimb/ekko`, it opens '`/u/jimb/ekko`', parses the three arguments `-e`, `main`, and `-s` from it, and substitutes them for the `\` switch. Thus, Guile's command line now reads:

  ```
  /usr/local/bin/guile -e main -s /u/jimb/ekko a b c
  ```
- Guile then processes these switches: it loads '`/u/jimb/ekko`' as a file of Scheme code (treating the first three lines as a comment), and then performs the application `(main "/u/jimb/ekko" "a" "b" "c")`.

When Guile sees the meta switch `\`, it parses command-line argument from the script file according to the following rules:

- Each space character terminates an argument. This means that two spaces in a row introduce an argument `""`.
- The tab character is not permitted (unless you quote it with the backslash character, as described below), to avoid confusion.
- The newline character terminates the sequence of arguments, and will also terminate a final non-empty argument. (However, a newline following a space will not introduce a final empty-string argument; it only terminates the argument list.)
- The backslash character is the escape character. It escapes backslash, space, tab, and newline. The ANSI C escape sequences like `\n` and `\t` are also supported. These produce argument constituents; the two-character combination `\n` doesn't act like a terminating newline. The escape sequence `\`*`NNN`* for exactly three octal digits reads as the character whose ASCII code is *NNN*. As above, characters produced this way are argument constituents. Backslash followed by other characters is not allowed.

3.3.4 Command Line Handling

The ability to accept and handle command line arguments is very important when writing Guile scripts to solve particular problems, such as extracting information from text files or interfacing with existing command line applications. This chapter describes how Guile makes command line arguments available to a Guile script, and the utilities that Guile provides to help with the processing of command line arguments.

When a Guile script is invoked, Guile makes the command line arguments accessible via the procedure `command-line`, which returns the arguments as a list of strings.

For example, if the script

```
#! /usr/local/bin/guile -s
!#
(write (command-line))
(newline)
```

is saved in a file '`cmdline-test.scm`' and invoked using the command line `./cmdline-test.scm bar.txt -o foo -frumple grob`, the output is

```
("./cmdline-test.scm" "bar.txt" "-o" "foo" "-frumple" "grob")
```

If the script invocation includes a `-e` option, specifying a procedure to call after loading the script, Guile will call that procedure with `(command-line)` as its argument. So a script that uses `-e` doesn't need to refer explicitly to `command-line` in its code. For example, the script above would have identical behaviour if it was written instead like this:

```
#! /usr/local/bin/guile \
-e main -s
!#
(define (main args)
  (write args)
  (newline))
```

(Note the use of the meta switch `\` so that the script invocation can include more than one Guile option: See Section 3.3.3 [The Meta Switch], page 35.)

These scripts use the `#!` POSIX convention so that they can be executed using their own file names directly, as in the example command line `./cmdline-test.scm bar.txt -o foo -frumple grob`. But they can also be executed by typing out the implied Guile command line in full, as in:

```
$ guile -s ./cmdline-test.scm bar.txt -o foo -frumple grob
```

or

```
$ guile -e main -s ./cmdline-test2.scm bar.txt -o foo -frumple grob
```

Even when a script is invoked using this longer form, the arguments that the script receives are the same as if it had been invoked using the short form. Guile ensures that the `(command-line)` or `-e` arguments are independent of how the script is invoked, by stripping off the arguments that Guile itself processes.

A script is free to parse and handle its command line arguments in any way that it chooses. Where the set of possible options and arguments is complex, however, it can get tricky to extract all the options, check the validity of given arguments, and so on. This task can be greatly simplified by taking advantage of the module `(ice-9 getopt-long)`, which is distributed with Guile, See Section 6.3 [getopt-long], page 418.

3.3.5 Scripting Examples

To start with, here are some examples of invoking Guile directly:

`guile -- a b c`

Run Guile interactively; `(command-line)` will return `("/usr/local/bin/guile" "a" "b" "c")`.

`guile -s /u/jimb/ex2 a b c`
: Load the file '`/u/jimb/ex2`'; `(command-line)` will return `("/u/jimb/ex2" "a" "b" "c")`.

`guile -c '(write %load-path) (newline)'`
: Write the value of the variable `%load-path`, print a newline, and exit.

`guile -e main -s /u/jimb/ex4 foo`
: Load the file '`/u/jimb/ex4`', and then call the function `main`, passing it the list `("/u/jimb/ex4" "foo")`.

`guile -l first -ds -l last -s script`
: Load the files '`first`', '`script`', and '`last`', in that order. The `-ds` switch says when to process the `-s` switch. For a more motivated example, see the scripts below.

Here is a very simple Guile script:

```
#!/usr/local/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

The first line marks the file as a Guile script. When the user invokes it, the system runs '`/usr/local/bin/guile`' to interpret the script, passing `-s`, the script's filename, and any arguments given to the script as command-line arguments. When Guile sees `-s` ***`script`***, it loads *script*. Thus, running this program produces the output:

```
Hello, world!
```

Here is a script which prints the factorial of its argument:

```
#!/usr/local/bin/guile -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(display (fact (string->number (cadr (command-line)))))
(newline)
```

In action:

```
$ fact 5
120
$
```

However, suppose we want to use the definition of `fact` in this file from another script. We can't simply `load` the script file, and then use `fact`'s definition, because the script will try to compute and display a factorial when we load it. To avoid this problem, we might write the script this way:

```
#!/usr/local/bin/guile \
-e main -s
!#
(define (fact n)
    (if (zero? n) 1
      (* n (fact (- n 1)))))

(define (main args)
  (display (fact (string->number (cadr args))))
  (newline))
```

This version packages the actions the script should perform in a function, `main`. This allows us to load the file purely for its definitions, without any extraneous computation taking place. Then we used the meta switch `\` and the entry point switch `-e` to tell Guile to call `main` after loading the script.

```
$ fact 50
30414093201713378043612608166064768844377641568960512000000000000
```

Suppose that we now want to write a script which computes the `choose` function: given a set of *m* distinct objects, `(choose n m)` is the number of distinct subsets containing *n* objects each. It's easy to write `choose` given `fact`, so we might write the script this way:

```
#!/usr/local/bin/guile \
-l fact -e main -s
!#
(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

The command-line arguments here tell Guile to first load the file '`fact`', and then run the script, with `main` as the entry point. In other words, the `choose` script can use definitions made in the `fact` script. Here are some sample runs:

```
$ choose 0 4
1
$ choose 1 4
4
$ choose 2 4
6
$ choose 3 4
4
$ choose 4 4
1
$ choose 50 100
100891344545564193334812497256
```

3.4 Debugging Features

Guile includes debugging tools to help you work out what is going wrong when a program signals an error or behaves differently to how you would expect. This chapter describes how to use these tools.

Broadly speaking, Guile's debugging support allows you to do two things:

- specify *breakpoints* — points in the execution of a program where execution should pause so you can see what is going on
- examine in detail the "scene of the crime" — in other words, the execution context at a breakpoint, or when the last error occurred.

The details are more complex and more powerful ...

3.4.1 Debugging the Most Recent Error

When an error is signalled, Guile remembers the execution context where the error occurred. By default, Guile then displays only the most immediate information about where and why the error occurred, for example:

```
(make-string (* 4 (+ 3 #\s)) #\space)
⊣
standard input:2:19: In procedure + in expression (+ 3 #\s):
standard input:2:19: Wrong type argument: #\s
ABORT: (wrong-type-arg)

Type "(backtrace)" to get more information or "(debug)" to enter
the debugger.
```

However, as the message above says, you can obtain much more information about the context of the error by typing `(backtrace)` or `(debug)`.

`(backtrace)` displays the Scheme call stack at the point where the error occurred:

```
(backtrace)
⊣
Backtrace:
In standard input:
   2: 0* [make-string ...
   2: 1*  [* 4 ...
   2: 2*   [+ 3 #\s]

Type "(debug-enable 'backtrace)" if you would like a backtrace
automatically if an error occurs in the future.
```

In a more complex scenario than this one, this can be extremely useful for understanding where and why the error occurred. For more on the format of the displayed backtrace, see the subsection below.

`(debug)` takes you into Guile's interactive debugger, which provides commands that allow you to

- display the Scheme call stack at the point where the error occurred (the `backtrace` command — see Section 3.4.3.1 [Display Backtrace], page 47)

- move up and down the call stack, to see in detail the expression being evaluated, or the procedure being applied, in each *frame* (the `up`, `down`, `frame`, `position`, `info args` and `info frame` commands — see Section 3.4.3.2 [Frame Selection], page 47 and Section 3.4.3.3 [Frame Information], page 48)
- examine the values of variables and expressions in the context of each frame (the `evaluate` command — see Section 3.4.3.4 [Frame Evaluation], page 48).

Use of the interactive debugger, including these commands, is described in Section 3.4.3 [Interactive Debugger], page 47.

3.4.1.1 How to Interpret a Backtrace

3.4.2 Intro to Breakpoints

If you are not already familiar with the concept of breakpoints, the first subsection below explains how they work are why they are useful.

Broadly speaking, Guile's breakpoint support consists of

- type-specific features for *creating* breakpoints of various types
- relatively generic features for *manipulating* the behaviour of breakpoints once they've been created.

Different breakpoint types are implemented as different classes in a GOOPS hierarchy with common base class `<breakpoint>`. The magic of generic functions then allows most of the manipulation functions to be generic by default but specializable (by breakpoint class) if the need arises.

Generic breakpoint support is provided by the `(ice-9 debugger breakpoints)` module, so you will almost always need to use this module in order to access the functionality described here:

```
(use-modules (ice-9 debugger breakpoints))
```

You may like to add this to your '`.guile`' file.

3.4.2.1 How Breakpoints Work and Why They Are Useful

Often, debugging the last error is not enough to tell you what went wrong. For example, the root cause of the error may have arisen a long time before the error was signalled, in which case the execution context of the error is too late to be useful. Or your program might not signal an error at all, just return an unexpected result or have some incorrect side effect.

In many such cases, it's useful to pause the program at or before the point where you suspect the problem arises. Then you can explore the stack, display the values of key variables, and generally check that the state of the program is as you expect. If all is well, you can let the program continue running normally, or step more slowly through each expression that the Scheme interpreter evaluates. Single-stepping may reveal that the program is going through blocks of code that you didn't intend — a useful data point for understanding what the underlying problem is.

Telling Guile where or when to pause a program is called *setting a breakpoint*. When a breakpoint is hit, Guile's default behaviour is to enter the interactive debugger, where there are now two sets of commands available:

- all the commands as described for last error debugging (see Section 3.4.1 [Debug Last Error], page 40), which allow you to explore the stack and so on
- additional commands for continuing program execution in various ways: `next`, `step`, `finish`, `trace-finish` and `continue`.

Use of the interactive debugger is described in Section 3.4.3 [Interactive Debugger], page 47.

3.4.2.2 Source Breakpoints

A source breakpoint is a breakpoint that triggers whenever program execution hits a particular source location. A source breakpoint can be conveniently set simply by evaluating code that has `##` inserted into it at the position where you want the breakpoint to be.

For example, to set a breakpoint immediately before evaluation of `(= n 0)` in the following procedure definition, evaluate:

```
(define (fact1 n)
  (if ##(= n 0)
      1
      (* n (fact1 (- n 1)))))
⊣
Set breakpoint 1: standard input:4:9: (= n 0)
```

Note the message confirming that you have set a breakpoint. If you don't see this, something isn't working.

`##` is provided by the `(ice-9 debugger breakpoints source)` module, so you must use this module before trying to set breakpoints in this way:

```
(use-modules (ice-9 debugger breakpoints source))
```

You may like to add this to your '`.guile`' file.

The default behaviour for source breakpoints is `debug-here` (see Section 3.4.2.7 [Breakpoint Behaviours], page 44), which means to enter the command line debugger when the breakpoint is hit. So, if you now use `fact1`, that is what happens.

```
guile> (fact1 3)
Hit breakpoint 1: standard input:4:9: (= n 0)
Frame 3 at standard input:4:9
        (= n 0)
debug>
```

3.4.2.3 Procedural Breakpoints

A procedural breakpoint is a breakpoint that triggers whenever Guile is about to apply a specified procedure to its (already evaluated) arguments. To set a procedural breakpoint, call `break!` with the target procedure as a single argument. For example:

```
(define (fact1 n)
  (if (= n 0)
      1
      (* n (fact1 (- n 1)))))

(break! fact1)
⊣
Set breakpoint 1: [fact1]
⇒
#<<procedure-breakpoint> 808b0b0>
```

Alternatives to `break!` are `trace!` and `trace-subtree!`. The difference is that these three calls create a breakpoint in the same place but with three different behaviours, respectively `debug-here`, `trace-here` and `trace-subtree`. Breakpoint behaviours are documented fully later (see Section 3.4.2.7 [Breakpoint Behaviours], page 44), but to give a quick taste, here's an example of running code that includes a procedural breakpoint with the `trace-here` behaviour.

```
(trace! fact1)
⊣
Set breakpoint 1: [fact1]
⇒
#<<procedure-breakpoint> 808b0b0>

(fact1 4)
⊣
|  [fact1 4]
|  |  [fact1 3]
|  |  |  [fact1 2]
|  |  |  |  [fact1 1]
|  |  |  |  |  [fact1 0]
|  |  |  |  |  1
|  |  |  |  2
|  |  |  6
|  |  24
|  24
⇒
24
```

To set and use procedural breakpoints, you will need to use the `(ice-9 debugger breakpoints procedural)` module:

```
(use-modules (ice-9 debugger breakpoints procedural))
```

You may like to add this to your '`.guile`' file.

3.4.2.4 Setting Breakpoints

In general, that is. We've already seen how to set source and procedural breakpoints conveniently in practice. This section explains how those conveniences map onto a more general mechanism.

The general mechanism for setting breakpoints is the generic function `set-breakpoint!`. Different kinds of breakpoints define subclasses of the class `<breakpoint>` and provide their own methods for `set-pbreakpoint!`.

For example, `(ice-9 debugger breakpoints procedural)` implements the `<procedure-breakpoint>` subclass and provides a `set-breakpoint!` method that takes a procedure argument:

```
(set-breakpoint! behavior fact1)
⊣
Set breakpoint 1: [fact1]
⇒
#<<procedure-breakpoint> 808b0b0>
```

A non-type-specific `set-breakpoint!` method is provided by the generic module `(ice-9 debugger breakpoints)`. It allows you to change the behaviour of an existing breakpoint that is identified by its breakpoint number.

```
(set-breakpoint! behavior 1)
```

3.4.2.5 break! trace! trace-subtree!

We have already talked above about the use of `break!`, `trace!` and `trace-subtree!` for setting procedural breakpoints. Now that `set-breakpoint!` has been introduced, we can reveal that `break!`, `trace!` and `trace-subtree!` are in fact just wrappers for `set-breakpoint!` that specify particular breakpoint behaviours, respectively `debug-here`, `trace-here` and `trace-subtree`.

```
(break! . args)
    ≡ (set-breakpoint! debug-here . args)
(trace! . args)
    ≡ (set-breakpoint! trace-here . args)
(trace-subtree! . args)
    ≡ (set-breakpoint! trace-subtree . args)
```

This means that these three procedures can be used to set the corresponding behaviours for any type of breakpoint for which a `set-breakpoint!` method exists, not just procedural ones.

3.4.2.6 Accessing Breakpoints

Information about the state and behaviour of a breakpoint is stored in an instance of the appropriate breakpoint class. To access and change that information, therefore, you need to get hold of the desired breakpoint instance.

The generic function `get-breakpoint` meets this need: For every `set-breakpoint!` method there is a corresponding `get-breakpoint` method. Note especially the useful type-independent case:

```
(get-breakpoint 1)
⇒
#<<procedure-breakpoint> 808b0b0>
```

3.4.2.7 Breakpoint Behaviours

A breakpoint's *behaviour* determines what happens when that breakpoint is hit. Several kinds of behaviour are generally useful.

`debug-here`
: Enter the command line debugger. This gives the opportunity to explore the stack, evaluate expressions in any of the pending stack frames, change breakpoint properties or set new breakpoints, and continue program execution when you are done.

`trace-here`
: Trace the current stack frame. For expressions being evaluated, this shows the expression. For procedure applications, it shows the procedure name and its arguments *post-evaluation*. For both expressions and applications, the indentation of the tracing indicates whether the traced items are mutually tail recursive.

`trace-subtree`
: Trace the current stack frame, and enable tracing for all future evaluations and applications until the current stack frame is exited. `trace-subtree` is a great preliminary exploration tool when all you know is that there is a bug "somewhere in XXX or in something that XXX calls".

`(at-exit thunk)`
: Don't do anything now, but arrange for *thunk* to be executed when the current stack frame is exited. For example, the operation that most debugging tools call "finish" is `(at-exit debug-here)`.

`(at-next count thunk)`
: . . . arrange for *thunk* to be executed when beginning the *count*th next evaluation or application with source location in the current file.

`(at-entry count thunk)`
: . . . arrange for *thunk* to be executed when beginning the *count*th next evaluation (regardless of source location).

`(at-apply count thunk)`
: . . . arrange for *thunk* to be executed just before performing the *count*th next application (regardless of source location).

`(at-step count thunk)`
: Synthesis of `at-entry` and `at-apply`; counts both evaluations and applications.

Every breakpoint instance has a slot in which its behaviour is stored. If you have a breakpoint instance in hand, you can change its behaviour using the `bp-behaviour` accessor.

An *accessor* supports the setting of a property like this:

```
(set! (bp-behaviour breakpoint) new-behaviour)
```

See the GOOPS manual for further information on accessors.

Alternatively, if you know how to specify the *location-args* for the breakpoint in question, you can change its behaviour using `set-breakpoint!`. For example:

```
;; Change behaviour of breakpoint number 2.
(set-breakpoint! new-behaviour 2)

;; Change behaviour of procedural breakpoint on [fact1].
(set-breakpoint! new-behaviour fact1)
```

In all cases, the behaviour that you specify should be either a single thunk, or a list of thunks, to be called when the breakpoint is hit.

The most common behaviours above are exported as thunks from the `(ice-9 debugger behaviour)` module. So, if you use this module, you can use those behaviours directly like this:

```
(use-modules (ice-9 debugger behaviour))
(set-breakpoint! trace-subtree 2)
(set! (bp-behaviour (get-breakpoint 3)) debug-here)
```

You can also use the list option to combine common behaviours:

```
(set-breakpoint! (list trace-here debug-here) 2)
```

Or, for more customized behaviour, you could build and use your own thunk like this:

```
(define (my-behaviour)
  (trace-here)
  (at-exit (lambda ()
             (display "Exiting frame of my-behaviour bp\n")
             ... do something unusual ...)))

(set-breakpoint my-behaviour 2)
```

3.4.2.8 Enabling and Disabling

Independently of its behaviour, each breakpoint also keeps track of whether it is currently enabled. This is a straightforward convenience to allow breakpoints to be temporarily switched off without losing all their carefully constructed properties.

If you have a breakpoint instance in hand, you can enable or disable it using the `bp-enabled?` accessor.

Alternatively, you can enable or disable a breakpoint via its location args by using `enable-breakpoint!` or `disable-breakpoint!`.

```
(disable-breakpoint! fact1)     ; disable the procedural breakpoint on fact1
(enable-breakpoint! 1)          ; enable breakpoint 1
```

`enable-breakpoint!` and `disable-breakpoint!` are implemented using `get-breakpoint` and `bp-enabled?`, so any *location-args* that are valid for `get-breakpoint` will work also for these procedures.

3.4.2.9 Deleting Breakpoints

Given a breakpoint instance in hand, you can deactivate it and remove it from the global list of current breakpoints by calling `bp-delete!`.

Alternatively, you can delete a breakpoint by its location args:

```
(delete-breakpoint! 1)         ; delete breakpoint 1
```

`delete-breakpoint!` is implemented using `get-breakpoint` and `bp-delete!`, so any *location-args* that are valid for `get-breakpoint` will work also for `delete-breakpoint!`.

There is no way to reinstate a deleted breakpoint. Final destruction of the breakpoint instance is determined by the usual garbage collection rules.

3.4.2.10 Breakpoint Information

To get Guile to print a description of a breakpoint instance, use `bp-describe`:

```
(bp-describe (get-breakpoint 1) #t)   ; #t specifies standard output
 ⊣
Breakpoint 1: [fact1]
        enabled? = #t
        behaviour = #<procedure trace-here ()>
```

Following the usual model, `describe-breakpoint` is also provided:

```
(describe-breakpoint 1)
 ⊣
Breakpoint 1: [fact1]
        enabled? = #t
        behaviour = #<procedure trace-here ()>
```

Finally, two stragglers. `all-breakpoints` returns a list of all current breakpoints. `describe-all-breakpoints` combines `bp-describe` and `all-breakpoints` by printing a description of all current breakpoints to standard output.

3.4.2.11 Other Breakpoint Types

Besides source and procedural breakpoints, Guile includes an early implementation of a third class of breakpoints: *range* breakpoints. These are breakpoints that trigger when program execution enters (or perhaps exits) a defined range of source locations.

Sadly, these don't yet work well. The apparent problem is that the extra methods for `set-breakpoint!` and `get-breakpoint` cause some kind of explosion in the time taken by GOOPS to construct its method cache and to dispatch calls involving these generic functions. But we haven't really investigated enough to be sure that this is the real issue.

If you're interested in looking and/or investigating anyway, please feel free to check out and play with the `(ice-9 debugger breakpoints range)` module.

The other kind of breakpoint that we'd like to have is watchpoints, but this hasn't been implemented at all yet. Watchpoints may turn out to be impractical for performance reasons.

3.4.3 Using the Interactive Debugger

Guile's interactive debugger is a command line application that accepts commands from you for examining the stack and, if at a breakpoint, for continuing program execution in various ways. Unlike in the normal Guile REPL, commands are typed mostly without parentheses.

When you first enter the debugger, it introduces itself with a message like this:

```
This is the Guile debugger -- for help, type ‘help’.
There are 3 frames on the stack.

Frame 2 at standard input:36:19
        [+ 3 #\s]
debug>
```

"debug>" is the debugger's prompt, and a useful reminder that you are not in the normal Guile REPL. The available commands are described in detail in the following subsections.

3.4.3.1 Display Backtrace

The `backtrace` command, which can also be invoked as `bt` or `where`, displays the call stack (aka backtrace) at the point where the debugger was entered:

```
debug> bt
In standard input:
  36: 0* [make-string ...
  36: 1*  [* 4 ...
  36: 2*   [+ 3 #\s]
```

`backtrace` [*count*] [Debugger Command]
`bt` [*count*] [Debugger Command]
`where` [*count*] [Debugger Command]

Print backtrace of all stack frames, or of the innermost *count* frames. With a negative argument, print the outermost -*count* frames. If the number of frames isn't explicitly given, the debug option `depth` determines the maximum number of frames printed.

The format of the displayed backtrace is the same as for the `backtrace` procedure — see Section 3.4.1.1 [Backtrace Format], page 41 for details.

3.4.3.2 Frame Selection

A call stack consists of a sequence of stack *frames*, with each frame describing one level of the nested evaluations and applications that the program was executing when it hit a breakpoint or an error. Frames are numbered such that frame 0 is the outermost — i.e. the operation on the call stack that began least recently — and frame N-1 the innermost (where N is the total number of frames on the stack).

When you enter the debugger, the innermost frame is selected, which means that the commands for getting information about the "current" frame, or for evaluating expressions in the context of the current frame, will do so by default with respect to the innermost frame. To select a different frame, so that these operations will apply to it instead, use the `up`, `down` and `frame` commands like this:

```
debug> up
Frame 1 at standard input:36:14
        [* 4 ...
debug> frame 0
Frame 0 at standard input:36:1
        [make-string ...
debug> down
Frame 1 at standard input:36:14
        [* 4 ...
```

`up` [*n*] [Debugger Command]

Move *n* frames up the stack. For positive *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one.

`down` [*n*] [Debugger Command]

Move *n* frames down the stack. For positive *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one.

`frame` [*n*] [Debugger Command]

Select and print a stack frame. With no argument, print the selected stack frame. (See also "info frame".) An argument specifies the frame to select; it must be a stack-frame number.

3.4.3.3 Frame Information

[to be completed]

`info frame` [Debugger Command]

All about selected stack frame.

`info args` [Debugger Command]

Argument variables of current stack frame.

`position` [Debugger Command]

Display the position of the current expression.

3.4.3.4 Frame Evaluation

[to be completed]

`evaluate` *expression* [Debugger Command]

Evaluate an expression. The expression must appear on the same line as the command, however it may be continued over multiple lines.

3.4.3.5 Single Stepping

[to be completed]

`step` [*n*] [Debugger Command]

Continue until entry to *n*th next frame.

`next` [*n*] [Debugger Command]

Continue until entry to *n*th next frame in same file.

3.4.3.6 Run To Frame Exit

[to be completed]

`finish` [Debugger Command]

Continue until evaluation of the current frame is complete, and print the result obtained.

`trace-finish` [Debugger Command]

Trace until evaluation of the current frame is complete.

3.4.3.7 Continue Execution

[to be completed]

`continue` [Debugger Command]

Continue program execution.

3.4.3.8 Leave Debugger

[to be completed]

`quit` [Debugger Command]

Exit the debugger.

3.4.4 Tracing

Tracing has already been described as a breakpoint behaviour (see Section 3.4.2.7 [Breakpoint Behaviours], page 44), but we mention it again here because it is so useful, and because Guile actually now has *two* mechanisms for tracing, and its worth clarifying the differences between them.

3.4.4.1 Tracing Provided by `(ice-9 debug)`

The `(ice-9 debug)` module implements tracing of procedure applications. When a procedure is *traced*, it means that every call to that procedure is reported to the user during a program run. The idea is that you can mark a collection of procedures for tracing, and Guile will subsequently print out a line of the form

```
|  |  [procedure args ...]
```

whenever a marked procedure is about to be applied to its arguments. This can help a programmer determine whether a function is being called at the wrong time or with the wrong set of arguments.

In addition, the indentation of the output is useful for demonstrating how the traced applications are or are not tail recursive with respect to each other. Thus, a trace of a non-tail recursive factorial implementation looks like this:

```
[fact1 4]
|  [fact1 3]
|  |  [fact1 2]
|  |  |  [fact1 1]
|  |  |  |  [fact1 0]
|  |  |  |  1
|  |  |  1
|  |  2
|  6
24
```

While a typical tail recursive implementation would look more like this:

```
[fact2 4]
[facti 1 4]
[facti 4 3]
[facti 12 2]
[facti 24 1]
[facti 24 0]
24
```

`trace` *procedure* [Scheme Procedure]

Enable tracing for `procedure`. While a program is being run, Guile will print a brief report at each call to a traced procedure, advising the user which procedure was called and the arguments that were passed to it.

`untrace` *procedure* [Scheme Procedure]

Disable tracing for `procedure`.

Here is another example:

```
(define (rev ls)
  (if (null? ls)
      '()
      (append (rev (cdr ls))
              (cons (car ls) '())))) ⇒ rev

(trace rev) ⇒ (rev)

(rev '(a b c d e))
⇒ [rev (a b c d e)]
   |  [rev (b c d e)]
   |  |  [rev (c d e)]
   |  |  |  [rev (d e)]
   |  |  |  |  [rev (e)]
   |  |  |  |  |  [rev ()]
```

```
        |  |  |  |  |  ()
        |  |  |  |  (e)
        |  |  |  (e d)
        |  |  (e d c)
        |  (e d c b)
        (e d c b a)
        (e d c b a)
```

Note the way Guile indents the output, illustrating the depth of execution at each procedure call. This can be used to demonstrate, for example, that Guile implements self-tail-recursion properly:

```
(define (rev ls sl)
  (if (null? ls)
      sl
      (rev (cdr ls)
           (cons (car ls) sl)))) ⇒ rev

(trace rev) ⇒ (rev)

(rev '(a b c d e) '())
⇒ [rev (a b c d e) ()]
   [rev (b c d e) (a)]
   [rev (c d e) (b a)]
   [rev (d e) (c b a)]
   [rev (e) (d c b a)]
   [rev () (e d c b a)]
   (e d c b a)
   (e d c b a)
```

Since the tail call is effectively optimized to a `goto` statement, there is no need for Guile to create a new stack frame for each iteration. Tracing reveals this optimization in operation.

3.4.4.2 Breakpoint-based Tracing

Guile's newer mechanism implements tracing as an optional behaviour for any kind of breakpoint.

To trace a procedure (in the same kind of way as the older tracing), use the `trace!` procedure to set a procedure breakpoint with `trace-here` behaviour:

```
(trace! fact1)
⊣
Set breakpoint 1: [fact1]
⇒
#<<procedure-breakpoint> 40337bf0>

(fact1 4)
⊣
|  [fact1 4]
|  |  [fact1 3]
|  |  |  [fact1 2]
|  |  |  |  [fact1 1]
|  |  |  |  |  [fact1 0]
|  |  |  |  |  1
|  |  |  |  2
|  |  |  6
|  |  24
|  24
⇒
24
```

To trace evaluation of a source expression, evaluate code containing a breakpoint marker `##` in the appropriate place, then use `set-breakpoint` to change the behaviour of the new breakpoint to `trace-here`:

```
(define (fact1 n)
  (if ##(= n 0)
      1
      (* n (fact1 (- n 1)))))
⊣
Set breakpoint 4: standard input:13:9: (= n 0)

(use-modules (ice-9 debugger behaviour))
(set-breakpoint! trace-here 4)
⊣
Breakpoint 4: standard input:13:9: (= n 0)
        enabled? = #t
        behaviour = #<procedure trace-here ()>

(fact1 4)
⊣
|  (= n 0)
|  #f
|  (= n 0)
|  #f
|  (= n 0)
|  #f
|  (= n 0)
|  #f
|  (= n 0)
|  #t
⇒
24
```

(Note — this example reveals a bug: each occurrence of `(= n 0)` should be shown indented with respect to the one before it, as `fact1` does not call itself tail-recursively.)

You can also give a breakpoint the `trace-subtree` behaviour, which means to trace the breakpoint location itself plus any evaluations and applications that occur below it in the call stack. In the following example, this allows us to see the evaluated arguments that are being compared by the = procedure:

```
(set-breakpoint! trace-subtree 4)
⊣
Breakpoint 4: standard input:13:9: (= n 0)
        enabled? = #t
        behaviour = #<procedure trace-subtree ()>

(fact1 4)
⊣
|  (= n 0)
|  [= 4 0]
|  #f
|  (= n 0)
|  [= 3 0]
|  #f
|  (= n 0)
|  [= 2 0]
|  #f
|  (= n 0)
|  [= 1 0]
|  #f
|  (= n 0)
|  [= 0 0]
|  #t
⇒
24
```

3.4.4.3 Differences Between Old and New Tracing Mechanisms

The newer tracing mechanism is more general and so more powerful than the older one: it works for expressions as well as procedure applications, and it implements the useful `trace-subtree` behaviour as well as the more traditional `trace-here`.

The older mechanism will probably become obsolete eventually, but it's worth keeping it around for a while until we are sure that the new mechanism is correct and does what programmers need.

3.5 Further Reading

- The website `http://www.schemers.org` is a good starting point for all things Scheme.
- Dorai Sitaram's online Scheme tutorial, *Teach Yourself Scheme in Fixnum Days*, at `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`. Includes a nice explanation of continuations.
- The complete text of *Structure and Interpretation of Computer Programs*, the classic introduction to computer science and Scheme by Hal Abelson, Jerry Sussman and Julie Sussman, is now available online at `http://mitpress.mit.edu/sicp/sicp.html`. This site also provides teaching materials related to the book, and all the source code used in the book, in a form suitable for loading and running.

4 Programming in C

This part of the manual explains the general concepts that you need to understand when interfacing to Guile from C. You will learn about how the latent typing of Scheme is embedded into the static typing of C, how the garbage collection of Guile is made available to C code, and how continuations influence the control flow in a C program.

This knowledge should make it straightforward to add new functions to Guile that can be called from Scheme. Adding new data types is also possible and is done by defining *smobs*.

The Section 4.6 [Programming Overview], page 82 section of this part contains general musings and guidelines about programming with Guile. It explores different ways to design a program around Guile, or how to embed Guile into existing programs.

There is also a pedagogical yet detailed explanation of how the data representation of Guile is implemented, See Appendix A [Data Representation], page 485. You don't need to know the details given there to use Guile from C, but they are useful when you want to modify Guile itself or when you are just curious about how it is all done.

For detailed reference information on the variables, functions etc. that make up Guile's application programming interface (API), See Chapter 5 [API Reference], page 93.

4.1 Linking Programs With Guile

This section covers the mechanics of linking your program with Guile on a typical POSIX system.

The header file `<libguile.h>` provides declarations for all of Guile's functions and constants. You should `#include` it at the head of any C source file that uses identifiers described in this manual. Once you've compiled your source files, you need to link them against the Guile object code library, `libguile`.

On most systems, you should not need to tell the compiler and linker explicitly where they can find '`libguile.h`' and '`libguile`'. When Guile has been installed in a peculiar way, or when you are on a peculiar system, things might not be so easy and you might need to pass additional `-I` or `-L` options to the compiler. Guile provides the utility program `guile-config` to help you find the right values for these options. You would typically run `guile-config` during the configuration phase of your program and use the obtained information in the Makefile.

4.1.1 Guile Initialization Functions

To initialize Guile, you can use one of several functions. The first, `scm_with_guile`, is the most portable way to initialize Guile. It will initialize Guile when necessary and then call a function that you can specify. Multiple threads can call `scm_with_guile` concurrently and it can also be called more than once in a given thread. The global state of Guile will survive from one call of `scm_with_guile` to the next. Your function is called from within `scm_with_guile` since the garbage collector of Guile needs to know where the stack of each thread is.

A second function, `scm_init_guile`, initializes Guile for the current thread. When it returns, you can use the Guile API in the current thread. This function employs some non-portable magic to learn about stack bounds and might thus not be available on all platforms.

One common way to use Guile is to write a set of C functions which perform some useful task, make them callable from Scheme, and then link the program with Guile. This yields a Scheme interpreter just like `guile`, but augmented with extra functions for some specific application — a special-purpose scripting language.

In this situation, the application should probably process its command-line arguments in the same manner as the stock Guile interpreter. To make that straightforward, Guile provides the `scm_boot_guile` and `scm_shell` function.

4.1.2 A Sample Guile Main Program

Here is '`simple-guile.c`', source code for a `main` and an `inner_main` function that will produce a complete Guile interpreter.

```
/* simple-guile.c --- how to start up the Guile
   interpreter from C code.  */

/* Get declarations for all the scm_ functions.  */
#include <libguile.h>

static void
inner_main (void *closure, int argc, char **argv)
{
  /* module initializations would go here */
  scm_shell (argc, argv);
}

int
main (int argc, char **argv)
{
  scm_boot_guile (argc, argv, inner_main, 0);
  return 0; /* never reached */
}
```

The `main` function calls `scm_boot_guile` to initialize Guile, passing it `inner_main`. Once `scm_boot_guile` is ready, it invokes `inner_main`, which calls `scm_shell` to process the command-line arguments in the usual way.

Here is a Makefile which you can use to compile the above program. It uses `guile-config` to learn about the necessary compiler and linker flags.

```
# Use GCC, if you have it installed.
CC=gcc

# Tell the C compiler where to find <libguile.h>
CFLAGS=`guile-config compile`

# Tell the linker what libraries to use and where to find them.
LIBS=`guile-config link`
```

```
simple-guile: simple-guile.o
        ${CC} simple-guile.o ${LIBS} -o simple-guile

simple-guile.o: simple-guile.c
        ${CC} -c ${CFLAGS} simple-guile.c
```

If you are using the GNU Autoconf package to make your application more portable, Autoconf will settle many of the details in the Makefile above automatically, making it much simpler and more portable; we recommend using Autoconf with Guile. Guile also provides the `GUILE_FLAGS` macro for autoconf that performs all necessary checks. Here is a '`configure.in`' file for `simple-guile` that uses this macro. Autoconf can use this file as a template to generate a `configure` script. In order for Autoconf to find the `GUILE_FLAGS` macro, you will need to run `aclocal` first (see section "Invoking aclocal" in *GNU Automake*).

```
AC_INIT(simple-guile.c)

# Find a C compiler.
AC_PROG_CC

# Check for Guile
GUILE_FLAGS

# Generate a Makefile, based on the results.
AC_OUTPUT(Makefile)
```

Here is a `Makefile.in` template, from which the `configure` script produces a Makefile customized for the host system:

```
# The configure script fills in these values.
CC=@CC@
CFLAGS=@GUILE_CFLAGS@
LIBS=@GUILE_LDFLAGS@

simple-guile: simple-guile.o
        ${CC} simple-guile.o ${LIBS} -o simple-guile
simple-guile.o: simple-guile.c
        ${CC} -c ${CFLAGS} simple-guile.c
```

The developer should use Autoconf to generate the '`configure`' script from the '`configure.in`' template, and distribute '`configure`' with the application. Here's how a user might go about building the application:

```
$ ls
Makefile.in     configure*      configure.in    simple-guile.c
$ ./configure
creating cache ./config.cache
checking for gcc... (cached) gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... (cached) yes
checking whether gcc accepts -g... (cached) yes
checking for Guile... yes
creating ./config.status
creating Makefile
$ make
gcc -c -I/usr/local/include simple-guile.c
gcc simple-guile.o -L/usr/local/lib -lguile -lqthreads -lpthread -lm -o simple-guile
$ ./simple-guile
guile> (+ 1 2 3)
6
guile> (getpwnam "jimb")
#("jimb" "83Z7d75W2tyJQ" 4008 10 "Jim Blandy" "/u/jimb"
  "/usr/local/bin/bash")
guile> (exit)
$
```

4.2 Linking Guile with Libraries

The previous section has briefly explained how to write programs that make use of an embedded Guile interpreter. But sometimes, all you want to do is make new primitive procedures and data types available to the Scheme programmer. Writing a new version of `guile` is inconvenient in this case and it would in fact make the life of the users of your new features needlessly hard.

For example, suppose that there is a program `guile-db` that is a version of Guile with additional features for accessing a database. People who want to write Scheme programs that use these features would have to use `guile-db` instead of the usual `guile` program. Now suppose that there is also a program `guile-gtk` that extends Guile with access to the popular Gtk+ toolkit for graphical user interfaces. People who want to write GUIs in Scheme would have to use `guile-gtk`. Now, what happens when you want to write a Scheme application that uses a GUI to let the user access a database? You would have to write a *third* program that incorporates both the database stuff and the GUI stuff. This might not be easy (because `guile-gtk` might be a quite obscure program, say) and taking this example further makes it easy to see that this approach can not work in practice.

It would have been much better if both the database features and the GUI feature had been provided as libraries that can just be linked with `guile`. Guile makes it easy to do just this, and we encourage you to make your extensions to Guile available as libraries whenever possible.

You write the new primitive procedures and data types in the normal fashion, and link them into a shared library instead of into a stand-alone program. The shared library can then be loaded dynamically by Guile.

4.2.1 A Sample Guile Extension

This section explains how to make the Bessel functions of the C library available to Scheme. First we need to write the appropriate glue code to convert the arguments and return values of the functions from Scheme to C and back. Additionally, we need a function that will add them to the set of Guile primitives. Because this is just an example, we will only implement this for the `j0` function.

Consider the following file '`bessel.c`'.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_make_real (j0 (scm_num2dbl (x, "j0")));
}

void
init_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

This C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux:

```
gcc -shared -o libguile-bessel.so -fPIC bessel.c
```

For creating shared libraries portably, we recommend the use of GNU Libtool (see section "Introduction" in *GNU Libtool*).

A shared library can be loaded into a running Guile process with the function `load-extension`. In addition to the name of the library to load, this function also expects the name of a function from that library that will be called to initialize it. For our example, we are going to call the function `init_bessel` which will make `j0_wrapper` available to Scheme programs with the name `j0`. Note that we do not specify a filename extension such as '`.so`' when invoking `load-extension`. The right extension for the host platform will be provided automatically.

```
(load-extension "libguile-bessel" "init_bessel")
(j0 2)
⇒ 0.223890779141236
```

For this to work, `load-extension` must be able to find '`libguile-bessel`', of course. It will look in the places that are usual for your operating system, and it will additionally look into the directories listed in the `LTDL_LIBRARY_PATH` environment variable.

To see how these Guile extensions via shared libraries relate to the module system, See Section 2.3.5.3 [Putting Extensions into Modules], page 9.

4.3 General concepts for using libguile

When you want to embed the Guile Scheme interpreter into your program or library, you need to link it against the '`libguile`' library (see Section 4.1 [Linking Programs With Guile], page 55). Once you have done this, your C code has access to a number of data types and functions that can be used to invoke the interpreter, or make new functions that you have written in C available to be called from Scheme code, among other things.

Scheme is different from C in a number of significant ways, and Guile tries to make the advantages of Scheme available to C as well. Thus, in addition to a Scheme interpreter, libguile also offers dynamic types, garbage collection, continuations, arithmetic on arbitrary sized numbers, and other things.

The two fundamental concepts are dynamic types and garbage collection. You need to understand how libguile offers them to C programs in order to use the rest of libguile. Also, the more general control flow of Scheme caused by continuations needs to be dealt with.

Running asynchronous signal handlers and multi-threading is known to C code already, but there are of course a few additional rules when using them together with libguile.

4.3.1 Dynamic Types

Scheme is a dynamically-typed language; this means that the system cannot, in general, determine the type of a given expression at compile time. Types only become apparent at run time. Variables do not have fixed types; a variable may hold a pair at one point, an integer at the next, and a thousand-element vector later. Instead, values, not variables, have fixed types.

In order to implement standard Scheme functions like `pair?` and `string?` and provide garbage collection, the representation of every value must contain enough information to accurately determine its type at run time. Often, Scheme systems also use this information to determine whether a program has attempted to apply an operation to an inappropriately typed value (such as taking the `car` of a string).

Because variables, pairs, and vectors may hold values of any type, Scheme implementations use a uniform representation for values — a single type large enough to hold either a complete value or a pointer to a complete value, along with the necessary typing information.

In Guile, this uniform representation of all Scheme values is the C type `SCM`. This is an opaque type and its size is typically equivalent to that of a pointer to `void`. Thus, `SCM` values can be passed around efficiently and they take up reasonably little storage on their own.

The most important rule is: You never access a `SCM` value directly; you only pass it to functions or macros defined in libguile.

As an obvious example, although a `SCM` variable can contain integers, you can of course not compute the sum of two `SCM` values by adding them with the C `+` operator. You must use the libguile function `scm_sum`.

Less obvious and therefore more important to keep in mind is that you also cannot directly test `SCM` values for trueness. In Scheme, the value `#f` is considered false and of course a `SCM` variable can represent that value. But there is no guarantee that the `SCM` representation of `#f` looks false to C code as well. You need to use `scm_is_true` or `scm_is_false` to test a `SCM` value for trueness or falseness, respectively.

You also can not directly compare two `SCM` values to find out whether they are identical (that is, whether they are `eq?` in Scheme terms). You need to use `scm_is_eq` for this.

The one exception is that you can directly assign a `SCM` value to a `SCM` variable by using the C `=` operator.

The following (contrived) example shows how to do it right. It implements a function of two arguments (*a* and *flag*) that returns *a*+1 if *flag* is true, else it returns *a* unchanged.

```
SCM
my_incrementing_function (SCM a, SCM flag)
{
  SCM result;

  if (scm_is_true (flag))
    result = scm_sum (a, scm_from_int (1));
  else
    result = a;

  return result;
}
```

Often, you need to convert between `SCM` values and approriate C values. For example, we needed to convert the integer `1` to its `SCM` representation in order to add it to *a*. Libguile provides many function to do these conversions, both from C to `SCM` and from `SCM` to C.

The conversion functions follow a common naming pattern: those that make a `SCM` value from a C value have names of the form `scm_from_`*`type`*` (...)` and those that convert a `SCM` value to a C value use the form `scm_to_`*`type`*` (...)`.

However, it is best to avoid converting values when you can. When you must combine C values and `SCM` values in a computation, it is often better to convert the C values to `SCM` values and do the computation by using libguile functions than to the other way around (converting `SCM` to C and doing the computation some other way).

As a simple example, consider this version of `my_incrementing_function` from above:

```
SCM
my_other_incrementing_function (SCM a, SCM flag)
{
  int result;

  if (scm_is_true (flag))
    result = scm_to_int (a) + 1;
  else
    result = scm_to_int (a);

  return scm_from_int (result);
}
```

This version is much less general than the original one: it will only work for values *A* that can fit into a `int`. The original function will work for all values that Guile can represent and that `scm_sum` can understand, including integers bigger than `long long`, floating point numbers, complex numbers, and new numerical types that have been added to Guile by third-party libraries.

Also, computing with `SCM` is not necessarily inefficient. Small integers will be encoded directly in the `SCM` value, for example, and do not need any additional memory on the heap. See Appendix A [Data Representation], page 485 to find out the details.

Some special `SCM` values are available to C code without needing to convert them from C values:

Scheme value	C representation
`#f`	`SCM_BOOL_F`
`#t`	`SCM_BOOL_T`
`()`	`SCM_EOL`

In addition to `SCM`, Guile also defines the related type `scm_t_bits`. This is an unsigned integral type of sufficient size to hold all information that is directly contained in a `SCM` value. The `scm_t_bits` type is used internally by Guile to do all the bit twiddling explained in Appendix A [Data Representation], page 485, but you will encounter it occasionally in low-level user code as well.

4.3.2 Garbage Collection

As explained above, the `SCM` type can represent all Scheme values. Some values fit entirely into a `SCM` value (such as small integers), but other values require additional storage in the heap (such as strings and vectors). This additional storage is managed automatically by Guile. You don't need to explicitely deallocate it when a `SCM` value is no longer used.

Two things must be guaranteed so that Guile is able to manage the storage automatically: it must know about all blocks of memory that have ever been allocated for Scheme values, and it must know about all Scheme values that are still being used. Given this knowledge, Guile can periodically free all blocks that have been allocated but are not used by any active Scheme values. This activity is called *garbage collection*.

It is easy for Guile to remember all blocks of memory that it has allocated for use by Scheme values, but you need to help it with finding all Scheme values that are in use by C code.

You do this when writing a SMOB mark function, for example (see Section 4.4.4 [Garbage Collecting Smobs], page 73). By calling this function, the garbage collector learns about all references that your SMOB has to other `SCM` values.

Other references to `SCM` objects, such as global variables of type `SCM` or other random data structures in the heap that contain fields of type `SCM`, can be made visible to the garbage collector by calling the functions `scm_gc_protect` or `scm_permanent_object`. You normally use these funtions for long lived objects such as a hash table that is stored in a global variable. For temporary references in local variables or function arguments, using these functions would be too expensive.

These references are handled differently: Local variables (and function arguments) of type `SCM` are automatically visible to the garbage collector. This works because the collector scans the stack for potential references to `SCM` objects and considers all referenced objects to be alive. The scanning considers each and every word of the stack, regardless of what it is actually used for, and then decides whether it could possibly be a reference to a `SCM` object. Thus, the scanning is guaranteed to find all actual references, but it might also find words that only accidentally look like references. These 'false positives' might keep `SCM` objects alive that would otherwise be considered dead. While this might waste memory, keeping an object around longer than it strictly needs to is harmless. This is why this technique is called "conservative garbage collection". In practice, the wasted memory seems to be no problem.

The stack of every thread is scanned in this way and the registers of the CPU and all other memory locations where local variables or function parameters might show up are included in this scan as well.

The consequence of the conservative scanning is that you can just declare local variables and function parameters of type `SCM` and be sure that the garbage collector will not free the corresponding objects.

However, a local variable or function parameter is only protected as long as it is really on the stack (or in some register). As an optimization, the C compiler might reuse its location for some other value and the `SCM` object would no longer be protected. Normally, this leads to exactly the right behabvior: the compiler will only overwrite a reference when it is no longer needed and thus the object becomes unprotected precisely when the reference disappears, just as wanted.

There are situations, however, where a `SCM` object needs to be around longer than its reference from a local variable or function parameter. This happens, for example, when you retrieve some pointer from a smob and work with that pointer directly. The reference to the `SCM` smob object might be dead after the pointer has been retrieved, but the pointer itself (and the memory pointed to) is still in use and thus the smob object must be protected. The compiler does not know about this connection and might overwrite the `SCM` reference too early.

To get around this problem, you can use `scm_remember_upto_here_1` and its cousins. It will keep the compiler from overwriting the reference. For a typical example of its use, see Section 4.4.6 [Remembering During Operations], page 75.

4.3.3 Control Flow

Scheme has a more general view of program flow than C, both locally and non-locally.

Controlling the local flow of control involves things like gotos, loops, calling functions and returning from them. Non-local control flow refers to situations where the program jumps across one or more levels of function activations without using the normal call or return operations.

The primitive means of C for local control flow is the `goto` statement, together with `if`. Loops done with `for`, `while` or `do` could in principle be rewritten with just `goto` and `if`. In Scheme, the primitive means for local control flow is the *function call* (together with `if`). Thus, the repetition of some computation in a loop is ultimately implemented by a function that calls itself, that is, by recursion.

This approach is theoretically very powerful since it is easier to reason formally about recursion than about gotos. In C, using recursion exclusively would not be practical, though, since it would eat up the stack very quickly. In Scheme, however, it is practical: function calls that appear in a *tail position* do not use any additional stack space (see Section 3.1.3.2 [Tail Calls], page 22).

A function call is in a tail position when it is the last thing the calling function does. The value returned by the called function is immediately returned from the calling function. In the following example, the call to `bar-1` is in a tail position, while the call to `bar-2` is not. (The call to `1-` in `foo-2` is in a tail position, though.)

```
(define (foo-1 x)
  (bar-1 (1- x)))

(define (foo-2 x)
  (1- (bar-2 x)))
```

Thus, when you take care to recurse only in tail positions, the recursion will only use constant stack space and will be as good as a loop constructed from gotos.

Scheme offers a few syntactic abstractions (`do` and *named* `let`) that make writing loops slightly easier.

But only Scheme functions can call other functions in a tail position: C functions can not. This matters when you have, say, two functions that call each other recursively to form a common loop. The following (unrealistic) example shows how one might go about determing whether a non-negative integer *n* is even or odd.

```
(define (my-even? n)
  (cond ((zero? n) #t)
        (else (my-odd? (1- n)))))

(define (my-odd? n)
  (cond ((zero? n) #f)
        (else (my-even? (1- n)))))
```

Because the calls to `my-even?` and `my-odd?` are in tail positions, these two procedures can be applied to arbitrary large integers without overflowing the stack. (They will still take a lot of time, of course.)

However, when one or both of the two procedures would be rewritten in C, it could no longer call its companion in a tail position (since C does not have this concept). You might need to take this consideration into account when deciding which parts of your program to write in Scheme and which in C.

In addition to calling functions and returning from them, a Scheme program can also exit non-locally from a function so that the control flow returns directly to an outer level. This means that some functions might not return at all.

Even more, it is not only possible to jump to some outer level of control, a Scheme program can also jump back into the middle of a function that has already exited. This might cause some functions to return more than once.

In general, these non-local jumps are done by invoking *continuations* that have previously been captured using `call-with-current-continuation`. Guile also offers a slightly restricted set of functions, `catch` and `throw`, that can only be used for non-local exits. This restriction makes them more efficient. Error reporting (with the function `error`) is implemented by invoking `throw`, for example. The functions `catch` and `throw` belong to the topic of *exceptions*.

Since Scheme functions can call C functions and vice versa, C code can experience the more general control flow of Scheme as well. It is possible that a C function will not return at all, or will return more than once. While C does offer `setjmp` and `longjmp` for non-local exits, it is still an unusual thing for C code. In contrast, non-local exits are very common in Scheme, mostly to report errors.

You need to be prepared for the non-local jumps in the control flow whenever you use a function from `libguile`: it is best to assume that any `libguile` function might signal an error or run a pending signal handler (which in turn can do arbitrary things).

It is often necessary to take cleanup actions when the control leaves a function non-locally. Also, when the control returns non-locally, some setup actions might be called for. For example, the Scheme function `with-output-to-port` needs to modify the global state so that `current-output-port` returns the port passed to `with-output-to-port`. The global output port needs to be reset to its previous value when `with-output-to-port` returns normally or when it is exited non-locally. Likewise, the port needs to be set again when control enters non-locally.

Scheme code can use the `dynamic-wind` function to arrange for the setting and resetting of the global state. C code can use the corresponding `scm_internal_dynamic_wind` function, or a `scm_dynwind_begin`/`scm_dynwind_end` pair together with suitable 'dynwind actions' (see Section 5.11.9 [Dynamic Wind], page 266).

Instead of coping with non-local control flow, you can also prevent it by erecting a *continuation barrier*, See Section 5.17.3 [Continuation Barriers], page 325. The function `scm_c_with_continuation_barrier`, for example, is guaranteed to return exactly once.

4.3.4 Asynchronous Signals

You can not call libguile functions from handlers for POSIX signals, but you can register Scheme handlers for POSIX signals such as `SIGINT`. These handlers do not run during the actual signal delivery. Instead, they are run when the program (more precisely, the thread that the handler has been registered for) reaches the next *safe point*.

The libguile functions themselves have many such safe points. Consequently, you must be prepared for arbitrary actions anytime you call a libguile function. For example, even `scm_cons` can contain a safe point and when a signal handler is pending for your thread, calling `scm_cons` will run this handler and anything might happen, including a non-local exit although `scm_cons` would not ordinarily do such a thing on its own.

If you do not want to allow the running of asynchronous signal handlers, you can block them temporarily with `scm_dynwind_block_asyncs`, for example. See See Section 5.17.2.1 [System asyncs], page 323.

Since signal handling in Guile relies on safe points, you need to make sure that your functions do offer enough of them. Normally, calling libguile functions in the normal course of action is all that is needed. But when a thread might spent a long time in a code section that calls no libguile function, it is good to include explicit safe points. This can allow the user to interrupt your code with ⟨C-c⟩, for example.

You can do this with the macro `SCM_TICK`. This macro is syntactically a statement. That is, you could use it like this:

```
while (1)
  {
    SCM_TICK;
    do_some_work ();
```

```
     }
```

Frequent execution of a safe point is even more important in multi threaded programs, See Section 4.3.5 [Multi-Threading], page 66.

4.3.5 Multi-Threading

Guile can be used in multi-threaded programs just as well as in single-threaded ones.

Each thread that wants to use functions from libguile must put itself into *guile mode* and must then follow a few rules. If it doesn't want to honor these rules in certain situations, a thread can temporarily leave guile mode (but can no longer use libguile functions during that time, of course).

Threads enter guile mode by calling `scm_with_guile`, `scm_boot_guile`, or `scm_init_guile`. As explained in the reference documentation for these functions, Guile will then learn about the stack bounds of the thread and can protect the `SCM` values that are stored in local variables. When a thread puts itself into guile mode for the first time, it gets a Scheme representation and is listed by `all-threads`, for example.

While in guile mode, a thread promises to reach a safe point reasonably frequently (see Section 4.3.4 [Asynchronous Signals], page 65). In addition to running signal handlers, these points are also potential rendezvous points of all guile mode threads where Guile can orchestrate global things like garbage collection. Consequently, when a thread in guile mode blocks and does no longer frequent safe points, it might cause all other guile mode threads to block as well. To prevent this from happening, a guile mode thread should either only block in libguile functions (who know how to do it right), or should temporarily leave guile mode with `scm_without_guile`.

For some common blocking operations, Guile provides convenience functions. For example, if you want to lock a pthread mutex while in guile mode, you might want to use `scm_pthread_mutex_lock` which is just like `pthread_mutex_lock` except that it leaves guile mode while blocking.

All libguile functions are (intended to be) robust in the face of multiple threads using them concurrently. This means that there is no risk of the internal data structures of libguile becoming corrupted in such a way that the process crashes.

A program might still produce non-sensical results, though. Taking hashtables as an example, Guile guarantees that you can use them from multiple threads concurrently and a hashtable will always remain a valid hashtable and Guile will not crash when you access it. It does not guarantee, however, that inserting into it concurrently from two threads will give useful results: only one insertion might actually happen, none might happen, or the table might in general be modified in a totally arbitrary manner. (It will still be a valid hashtable, but not the one that you might have expected.) Guile might also signal an error when it detects a harmful race condition.

Thus, you need to put in additional synchronizations when multiple threads want to use a single hashtable, or any other mutable Scheme object.

When writing C code for use with libguile, you should try to make it robust as well. An example that converts a list into a vector will help to illustrate. Here is a correct version:

```
SCM
my_list_to_vector (SCM list)
```

```
{
  SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
  size_t len, i;

  len = SCM_SIMPLE_VECTOR_LENGTH (vector);
  i = 0;
  while (i < len && scm_is_pair (list))
    {
      SCM_SIMPLE_VECTOR_SET (vector, i, SCM_CAR (list));
      list = SCM_CDR (list);
      i++;
    }

  return vector;
}
```

The first thing to note is that storing into a `SCM` location concurrently from multiple threads is guaranteed to be robust: you don't know which value wins but it will in any case be a valid `SCM` value.

But there is no guarantee that the list referenced by *list* is not modified in another thread while the loop iterates over it. Thus, while copying its elements into the vector, the list might get longer or shorter. For this reason, the loop must check both that it doesn't overrun the vector (`SCM_SIMPLE_VECTOR_SET` does no range-checking) and that it doesn't overrung the list (`SCM_CAR` and `SCM_CDR` likewise do no type checking).

It is safe to use `SCM_CAR` and `SCM_CDR` on the local variable *list* once it is known that the variable contains a pair. The contents of the pair might change spontaneously, but it will always stay a valid pair (and a local variable will of course not spontaneously point to a different Scheme object).

Likewise, a simple vector such as the one returned by `scm_make_vector` is guaranteed to always stay the same length so that it is safe to only use SCM_SIMPLE_VECTOR_LENGTH once and store the result. (In the example, *vector* is safe anyway since it is a fresh object that no other thread can possibly know about until it is returned from `my_list_to_vector`.)

Of course the behavior of `my_list_to_vector` is suboptimal when *list* does indeed get asynchronously lengthened or shortened in another thread. But it is robust: it will always return a valid vector. That vector might be shorter than expected, or its last elements might be unspecified, but it is a valid vector and if a program wants to rule out these cases, it must avoid modifying the list asynchronously.

Here is another version that is also correct:

```
SCM
my_pedantic_list_to_vector (SCM list)
{
  SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
  size_t len, i;

  len = SCM_SIMPLE_VECTOR_LENGTH (vector);
  i = 0;
  while (i < len)
    {
      SCM_SIMPLE_VECTOR_SET (vector, i, scm_car (list));
      list = scm_cdr (list);
      i++;
    }

  return vector;
}
```

This version uses the type-checking and thread-robust functions `scm_car` and `scm_cdr` instead of the faster, but less robust macros `SCM_CAR` and `SCM_CDR`. When the list is shortened (that is, when *list* holds a non-pair), `scm_car` will throw an error. This might be preferable to just returning a half-initialized vector.

The API for accessing vectors and arrays of various kinds from C takes a slightly different approach to thread-robustness. In order to get at the raw memory that stores the elements of an array, you need to *reserve* that array as long as you need the raw memory. During the time an array is reserved, its elements can still spontaneously change their values, but the memory itself and other things like the size of the array are guaranteed to stay fixed. Any operation that would change these parameters of an array that is currently reserved will signal an error. In order to avoid these errors, a program should of course put suitable synchronization mechanisms in place. As you can see, Guile itself is again only concerned about robustness, not about correctness: without proper synchronization, your program will likely not be correct, but the worst consequence is an error message.

Real thread-safeness often requires that a critical section of code is executed in a certain restricted manner. A common requirement is that the code section is not entered a second time when it is already being executed. Locking a mutex while in that section ensures that no other thread will start executing it, blocking asyncs ensures that no asynchronous code enters the section again from the current thread, and the error checking of Guile mutexes guarantees that an error is signalled when the current thread accidentally reenters the critical section via recursive function calls.

Guile provides two mechanisms to support critical sections as outlined above. You can either use the macros `SCM_CRITICAL_SECTION_START` and `SCM_CRITICAL_SECTION_END` for very simple sections; or use a dynwind context together with a call to `scm_dynwind_critical_section`.

The macros only work reliably for critical sections that are guaranteed to not cause a non-local exit. They also do not detect an accidental reentry by the current thread. Thus, you should probably only use them to delimit critical sections that do not contain calls to libguile functions or to other external functions that might do complicated things.

The function `scm_dynwind_critical_section`, on the other hand, will correctly deal with non-local exits because it requires a dynwind context. Also, by using a separate mutex for each critical section, it can detect accidental reentries.

4.4 Defining New Types (Smobs)

Smobs are Guile's mechanism for adding new primitive types to the system. The term "smob" was coined by Aubrey Jaffer, who says it comes from "small object", referring to the fact that they are quite limited in size: they can hold just one pointer to a larger memory block plus 16 extra bits.

To define a new smob type, the programmer provides Guile with some essential information about the type — how to print it, how to garbage collect it, and so on — and Guile allocates a fresh type tag for it. The programmer can then use `scm_c_define_gsubr` to make a set of C functions visible to Scheme code that create and operate on these objects.

(You can find a complete version of the example code used in this section in the Guile distribution, in '`doc/example-smob`'. That directory includes a makefile and a suitable `main` function, so you can build a complete interactive Guile shell, extended with the datatypes described here.)

4.4.1 Describing a New Type

To define a new type, the programmer must write four functions to manage instances of the type:

`mark` Guile will apply this function to each instance of the new type it encounters during garbage collection. This function is responsible for telling the collector about any other `SCM` values that the object has stored. The default smob mark function does nothing. See Section 4.4.4 [Garbage Collecting Smobs], page 73, for more details.

`free` Guile will apply this function to each instance of the new type that is to be deallocated. The function should release all resources held by the object. This is analogous to the Java finalization method– it is invoked at an unspecified time (when garbage collection occurs) after the object is dead. The default free function frees the smob data (if the size of the struct passed to `scm_make_smob_type` is non-zero) using `scm_gc_free`. See Section 4.4.4 [Garbage Collecting Smobs], page 73, for more details.

This function operates while the heap is in an inconsistent state and must therefore be careful. See Section 5.7 [Smobs], page 222, for details about what this function is allowed to do.

`print` Guile will apply this function to each instance of the new type to print the value, as for `display` or `write`. The default print function prints `#<NAME ADDRESS>` where `NAME` is the first argument passed to `scm_make_smob_type`. For more information on printing, see Section A.2.5.6 [Port Data], page 497.

`equalp` If Scheme code asks the `equal?` function to compare two instances of the same smob type, Guile calls this function. It should return `SCM_BOOL_T` if *a* and *b* should be considered `equal?`, or `SCM_BOOL_F` otherwise. If `equalp` is `NULL`, `equal?` will assume that two instances of this type are never `equal?` unless they are `eq?`.

To actually register the new smob type, call `scm_make_smob_type`. It returns a value of type `scm_t_bits` which identifies the new smob type.

The four special functions described above are registered by calling one of `scm_set_smob_mark`, `scm_set_smob_free`, `scm_set_smob_print`, or `scm_set_smob_equalp`, as appropriate. Each function is intended to be used at most once per type, and the call should be placed immediately following the call to `scm_make_smob_type`.

There can only be at most 256 different smob types in the system. Instead of registering a huge number of smob types (for example, one for each relevant C struct in your application), it is sometimes better to register just one and implement a second layer of type dispatching on top of it. This second layer might use the 16 extra bits to extend its type, for example.

Here is how one might declare and register a new type representing eight-bit gray-scale images:

```
#include <libguile.h>

struct image {
  int width, height;
  char *pixels;

  /* The name of this image */
  SCM name;

  /* A function to call when this image is
     modified, e.g., to update the screen,
     or SCM_BOOL_F if no action necessary */
  SCM update_func;
};

static scm_t_bits image_tag;

void
init_image_type (void)
{
  image_tag = scm_make_smob_type ("image", sizeof (struct image));
  scm_set_smob_mark (image_tag, mark_image);
  scm_set_smob_free (image_tag, free_image);
  scm_set_smob_print (image_tag, print_image);
}
```

4.4.2 Creating Instances

Normally, smobs can have one *immediate* word of data. This word stores either a pointer to an additional memory block that holds the real data, or it might hold the data itself when it fits. The word is large enough for a `SCM` value, a pointer to `void`, or an integer that fits into a `size_t` or `ssize_t`.

You can also create smobs that have two or three immediate words, and when these words suffice to store all data, it is more efficient to use these super-sized smobs instead of using a normal smob plus a memory block. See Section 4.4.7 [Double Smobs], page 76, for their discussion.

Guile provides functions for managing memory which are often helpful when implementing smobs. See Section 5.14.2 [Memory Blocks], page 297.

To retrieve the immediate word of a smob, you use the macro `SCM_SMOB_DATA`. It can be set with `SCM_SET_SMOB_DATA`. The 16 extra bits can be accessed with `SCM_SMOB_FLAGS` and `SCM_SET_SMOB_FLAGS`.

The two macros `SCM_SMOB_DATA` and `SCM_SET_SMOB_DATA` treat the immediate word as if it were of type `scm_t_bits`, which is an unsigned integer type large enough to hold a pointer to `void`. Thus you can use these macros to store arbitrary pointers in the smob word.

When you want to store a `SCM` value directly in the immediate word of a smob, you should use the macros `SCM_SMOB_OBJECT` and `SCM_SET_SMOB_OBJECT` to access it.

Creating a smob instance can be tricky when it consists of multiple steps that allocate resources and might fail. It is recommended that you go about creating a smob in the following way:

- Allocate the memory block for holding the data with `scm_gc_malloc`.
- Initialize it to a valid state without calling any functions that might cause a non-local exits. For example, initialize pointers to NULL. Also, do not store `SCM` values in it that must be protected. Initialize these fields with `SCM_BOOL_F`.

  A valid state is one that can be safely acted upon by the *mark* and *free* functions of your smob type.
- Create the smob using `SCM_NEWSMOB`, passing it the initialized memory block. (This step will always succeed.)
- Complete the initialization of the memory block by, for example, allocating additional resources and making it point to them.

This procedure ensures that the smob is in a valid state as soon as it exists, that all resources that are allocated for the smob are properly associated with it so that they can be properly freed, and that no `SCM` values that need to be protected are stored in it while the smob does not yet competely exist and thus can not protect them.

Continuing the example from above, if the global variable `image_tag` contains a tag returned by `scm_make_smob_type`, here is how we could construct a smob whose immediate word contains a pointer to a freshly allocated `struct image`:

```
SCM
make_image (SCM name, SCM s_width, SCM s_height)
{
  SCM smob;
  struct image *image;
  int width = scm_to_int (s_width);
  int height = scm_to_int (s_height);

  /* Step 1: Allocate the memory block.
   */
  image = (struct image *) scm_gc_malloc (sizeof (struct image), "image");

  /* Step 2: Initialize it with straight code.
```

```
     */
    image->width = width;
    image->height = height;
    image->pixels = NULL;
    image->name = SCM_BOOL_F;
    image->update_func = SCM_BOOL_F;

    /* Step 3: Create the smob.
     */
    SCM_NEWSMOB (smob, image_tag, image);

    /* Step 4: Finish the initialization.
     */
    image->name = name;
    image->pixels = scm_gc_malloc (width * height, "image pixels");

    return smob;
  }
```

Let us look at what might happen when `make_image` is called.

The conversions of *s_width* and *s_height* to `int`s might fail and signal an error, thus causing a non-local exit. This is not a problem since no resources have been allocated yet that would have to be freed.

The allocation of *image* in step 1 might fail, but this is likewise no problem.

Step 2 can not exit non-locally. At the end of it, the *image* struct is in a valid state for the `mark_image` and `free_image` functions (see below).

Step 3 can not exit non-locally either. This is guaranteed by Guile. After it, *smob* contains a valid smob that is properly initialized and protected, and in turn can properly protect the Scheme values in its *image* struct.

But before the smob is completely created, `SCM_NEWSMOB` might cause the garbage collector to run. During this garbage collection, the `SCM` values in the *image* struct would be invisible to Guile. It only gets to know about them via the `mark_image` function, but that function can not yet do its job since the smob has not been created yet. Thus, it is important to not store `SCM` values in the *image* struct until after the smob has been created.

Step 4, finally, might fail and cause a non-local exit. In that case, the complete creation of the smob has not been successful, but it does nevertheless exist in a valid state. It will eventually be freed by the garbage collector, and all the resources that have been allocated for it will be correctly freed by `free_image`.

4.4.3 Type checking

Functions that operate on smobs should check that the passed `SCM` value indeed is a suitable smob before accessing its data. They can do this with `scm_assert_smob_type`.

For example, here is a simple function that operates on an image smob, and checks the type of its argument.

```
  SCM
  clear_image (SCM image_smob)
```

```
{
  int area;
  struct image *image;

  scm_assert_smob_type (image_tag, image_smob);

  image = (struct image *) SCM_SMOB_DATA (image_smob);
  area = image->width * image->height;
  memset (image->pixels, 0, area);

  /* Invoke the image's update function.
   */
  if (scm_is_true (image->update_func))
    scm_call_0 (image->update_func);

  scm_remember_upto_here_1 (image_smob);

  return SCM_UNSPECIFIED;
}
```

See Section 4.4.6 [Remembering During Operations], page 75 for an explanation of the call to `scm_remember_upto_here_1`.

4.4.4 Garbage Collecting Smobs

Once a smob has been released to the tender mercies of the Scheme system, it must be prepared to survive garbage collection. Guile calls the *mark* and *free* functions of the smob to manage this.

As described in more detail elsewhere (see Section A.2.2 [Conservative GC], page 490), every object in the Scheme system has a *mark bit*, which the garbage collector uses to tell live objects from dead ones. When collection starts, every object's mark bit is clear. The collector traces pointers through the heap, starting from objects known to be live, and sets the mark bit on each object it encounters. When it can find no more unmarked objects, the collector walks all objects, live and dead, frees those whose mark bits are still clear, and clears the mark bit on the others.

The two main portions of the collection are called the *mark phase*, during which the collector marks live objects, and the *sweep phase*, during which the collector frees all unmarked objects.

The mark bit of a smob lives in a special memory region. When the collector encounters a smob, it sets the smob's mark bit, and uses the smob's type tag to find the appropriate *mark* function for that smob. It then calls this *mark* function, passing it the smob as its only argument.

The *mark* function is responsible for marking any other Scheme objects the smob refers to. If it does not do so, the objects' mark bits will still be clear when the collector begins to sweep, and the collector will free them. If this occurs, it will probably break, or at least confuse, any code operating on the smob; the smob's `SCM` values will have become dangling references.

To mark an arbitrary Scheme object, the *mark* function calls `scm_gc_mark`.

Thus, here is how we might write `mark_image`:

```
SCM
mark_image (SCM image_smob)
{
  /* Mark the image's name and update function.  */
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_mark (image->name);
  scm_gc_mark (image->update_func);

  return SCM_BOOL_F;
}
```

Note that, even though the image's `update_func` could be an arbitrarily complex structure (representing a procedure and any values enclosed in its environment), `scm_gc_mark` will recurse as necessary to mark all its components. Because `scm_gc_mark` sets an object's mark bit before it recurses, it is not confused by circular structures.

As an optimization, the collector will mark whatever value is returned by the *mark* function; this helps limit depth of recursion during the mark phase. Thus, the code above should really be written as:

```
SCM
mark_image (SCM image_smob)
{
  /* Mark the image's name and update function.  */
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_mark (image->name);
  return image->update_func;
}
```

Finally, when the collector encounters an unmarked smob during the sweep phase, it uses the smob's tag to find the appropriate *free* function for the smob. It then calls that function, passing it the smob as its only argument.

The *free* function must release any resources used by the smob. However, it must not free objects managed by the collector; the collector will take care of them. For historical reasons, the return type of the *free* function should be `size_t`, an unsigned integral type; the *free* function should always return zero.

Here is how we might write the `free_image` function for the image smob type:

```
size_t
free_image (SCM image_smob)
{
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_free (image->pixels, image->width * image->height, "image pixels");
  scm_gc_free (image, sizeof (struct image), "image");
```

```
    return 0;
  }
```

During the sweep phase, the garbage collector will clear the mark bits on all live objects. The code which implements a smob need not do this itself.

There is no way for smob code to be notified when collection is complete.

It is usually a good idea to minimize the amount of processing done during garbage collection; keep the *mark* and *free* functions very simple. Since collections occur at unpredictable times, it is easy for any unusual activity to interfere with normal code.

4.4.5 Garbage Collecting Simple Smobs

It is often useful to define very simple smob types — smobs which have no data to mark, other than the cell itself, or smobs whose immediate data word is simply an ordinary Scheme object, to be marked recursively. Guile provides some functions to handle these common cases; you can use this function as your smob type's *mark* function, if your smob's structure is simple enough.

If the smob refers to no other Scheme objects, then no action is necessary; the garbage collector has already marked the smob cell itself. In that case, you can use zero as your mark function.

If the smob refers to exactly one other Scheme object via its first immediate word, you can use `scm_markcdr` as its mark function. Its definition is simply:

```
SCM
scm_markcdr (SCM obj)
{
  return SCM_SMOB_OBJECT (obj);
}
```

4.4.6 Remembering During Operations

It's important that a smob is visible to the garbage collector whenever its contents are being accessed. Otherwise it could be freed while code is still using it.

For example, consider a procedure to convert image data to a list of pixel values.

```
SCM
image_to_list (SCM image_smob)
{
  struct image *image;
  SCM lst;
  int i;

  scm_assert_smob_type (image_tag, image_smob);

  image = (struct image *) SCM_SMOB_DATA (image_smob);
  lst = SCM_EOL;
  for (i = image->width * image->height - 1; i >= 0; i--)
    lst = scm_cons (scm_from_char (image->pixels[i]), lst);

  scm_remember_upto_here_1 (image_smob);
  return lst;
```

```
    }
```

In the loop, only the `image` pointer is used and the C compiler has no reason to keep the `image_smob` value anywhere. If `scm_cons` results in a garbage collection, `image_smob` might not be on the stack or anywhere else and could be freed, leaving the loop accessing freed data. The use of `scm_remember_upto_here_1` prevents this, by creating a reference to `image_smob` after all data accesses.

There's no need to do the same for `lst`, since that's the return value and the compiler will certainly keep it in a register or somewhere throughout the routine.

The `clear_image` example previously shown (see Section 4.4.3 [Type checking], page 72) also used `scm_remember_upto_here_1` for this reason.

It's only in quite rare circumstances that a missing `scm_remember_upto_here_1` will bite, but when it happens the consequences are serious. Fortunately the rule is simple: whenever calling a Guile library function or doing something that might, ensure that the `SCM` of a smob is referenced past all accesses to its insides. Do this by adding an `scm_remember_upto_here_1` if there are no other references.

In a multi-threaded program, the rule is the same. As far as a given thread is concerned, a garbage collection still only occurs within a Guile library function, not at an arbitrary time. (Guile waits for all threads to reach one of its library functions, and holds them there while the collector runs.)

4.4.7 Double Smobs

Smobs are called smob because they are small: they normally have only room for one `void*` or `SCM` value plus 16 bits. The reason for this is that smobs are directly implemented by using the low-level, two-word cells of Guile that are also used to implement pairs, for example. (see Appendix A [Data Representation], page 485 for the details.) One word of the two-word cells is used for `SCM_SMOB_DATA` (or `SCM_SMOB_OBJECT`), the other contains the 16-bit type tag and the 16 extra bits.

In addition to the fundamental two-word cells, Guile also has four-word cells, which are appropriately called *double cells*. You can use them for *double smobs* and get two more immediate words of type `scm_t_bits`.

A double smob is created with `SCM_NEWSMOB2` or `SCM_NEWSMOB3` instead of `SCM_NEWSMOB`. Its immediate words can be retrieved as `scm_t_bits` with `SCM_SMOB_DATA_2` and `SCM_SMOB_DATA_3` in addition to `SCM_SMOB_DATA`. Unsurprisingly, the words can be set to `scm_t_bits` values with `SCM_SET_SMOB_DATA_2` and `SCM_SET_SMOB_DATA_3`.

Of course there are also `SCM_SMOB_OBJECT_2`, `SCM_SMOB_OBJECT_3`, `SCM_SET_SMOB_OBJECT_2`, and `SCM_SET_SMOB_OBJECT_3`.

4.4.8 The Complete Example

Here is the complete text of the implementation of the image datatype, as presented in the sections above. We also provide a definition for the smob's *print* function, and make some objects and functions static, to clarify exactly what the surrounding code is using.

As mentioned above, you can find this code in the Guile distribution, in '`doc/example-smob`'. That directory includes a makefile and a suitable `main` function, so you can build a complete interactive Guile shell, extended with the datatypes described here.)

```
/* file "image-type.c" */

#include <stdlib.h>
#include <libguile.h>

static scm_t_bits image_tag;

struct image {
  int width, height;
  char *pixels;

  /* The name of this image */
  SCM name;

  /* A function to call when this image is
     modified, e.g., to update the screen,
     or SCM_BOOL_F if no action necessary */
  SCM update_func;
};

static SCM
make_image (SCM name, SCM s_width, SCM s_height)
{
  SCM smob;
  struct image *image;
  int width = scm_to_int (s_width);
  int height = scm_to_int (s_height);

  /* Step 1: Allocate the memory block.
   */
  image = (struct image *) scm_gc_malloc (sizeof (struct image), "image");

  /* Step 2: Initialize it with straight code.
   */
  image->width = width;
  image->height = height;
  image->pixels = NULL;
  image->name = SCM_BOOL_F;
  image->update_func = SCM_BOOL_F;

  /* Step 3: Create the smob.
   */
  SCM_NEWSMOB (smob, image_tag, image);

  /* Step 4: Finish the initialization.
   */
  image->name = name;
  image->pixels = scm_gc_malloc (width * height, "image pixels");

  return smob;
}

SCM
clear_image (SCM image_smob)
{
  int area;
  struct image *image;

  scm_assert_smob_type (image_tag, image_smob);

  image = (struct image *) SCM_SMOB_DATA (image_smob);
  area = image->width * image->height;
  memset (image->pixels, 0, area);

  /* Invoke the image's update function.
   */
  if (scm_is_true (image->update_func))
    scm_call_0 (image->update_func);

  scm_remember_upto_here_1 (image_smob);

  return SCM_UNSPECIFIED;
}

static SCM
mark_image (SCM image_smob)
{
  /* Mark the image's name and update function.  */
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_mark (image->name);
  return image->update_func;
}

static size_t
free_image (SCM image_smob)
{
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_free (image->pixels, image->width * image->height, "image pixels");
  scm_gc_free (image, sizeof (struct image), "image");

  return 0;
}
```

```
static int
print_image (SCM image_smob, SCM port, scm_print_state *pstate)
{
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_puts ("#<image ", port);
  scm_display (image->name, port);
  scm_puts (">", port);

  /* non-zero means success */
  return 1;
}

void
init_image_type (void)
{
  image_tag = scm_make_smob_type ("image", sizeof (struct image));
  scm_set_smob_mark (image_tag, mark_image);
  scm_set_smob_free (image_tag, free_image);
  scm_set_smob_print (image_tag, print_image);

  scm_c_define_gsubr ("clear-image", 1, 0, 0, clear_image);
  scm_c_define_gsubr ("make-image", 3, 0, 0, make_image);
}
```

Here is a sample build and interaction with the code from the ‘example-smob’ directory, on the author’s machine:

```
zwingli:example-smob$ make CC=gcc
gcc ‘guile-config compile‘   -c image-type.c -o image-type.o
gcc ‘guile-config compile‘   -c myguile.c -o myguile.o
gcc image-type.o myguile.o ‘guile-config link‘ -o myguile
zwingli:example-smob$ ./myguile
guile> make-image
#<primitive-procedure make-image>
guile> (define i (make-image "Whistler’s Mother" 100 100))
guile> i
#<image Whistler’s Mother>
guile> (clear-image i)
guile> (clear-image 4)
ERROR: In procedure clear-image in expression (clear-image 4):
ERROR: Wrong type (expecting image): 4
ABORT: (wrong-type-arg)

Type "(backtrace)" to get more information.
guile>
```

4.5 Function Snarfing

When writing C code for use with Guile, you typically define a set of C functions, and then make some of them visible to the Scheme world by calling `scm_c_define_gsubr` or related functions. If you have many functions to publish, it can sometimes be annoying to keep the list of calls to `scm_c_define_gsubr` in sync with the list of function definitions.

Guile provides the `guile-snarf` program to manage this problem. Using this tool, you can keep all the information needed to define the function alongside the function definition itself; `guile-snarf` will extract this information from your source code, and automatically generate a file of calls to `scm_c_define_gsubr` which you can `#include` into an initialization function.

The snarfing mechanism works for many kind of initialiation actions, not just for collecting calls to `scm_c_define_gsubr`. For a full list of what can be done, See Section 5.4 [Snarfing Macros], page 97.

The `guile-snarf` program is invoked like this:

```
guile-snarf [-o outfile] [cpp-args ...]
```

This command will extract initialization actions to *outfile*. When no *outfile* has been specified or when *outfile* is `-`, standard output will be used. The C preprocessor is called with *cpp-args* (which usually include an input file) and the output is filtered to extract the initialization actions.

If there are errors during processing, *outfile* is deleted and the program exits with non-zero status.

During snarfing, the pre-processor macro `SCM_MAGIC_SNARFER` is defined. You could use this to avoid including snarfer output files that don't yet exist by writing code like this:

```
#ifndef SCM_MAGIC_SNARFER
#include "foo.x"
#endif
```

Here is how you might define the Scheme function `clear-image`, implemented by the C function `clear_image`:

```
#include <libguile.h>

SCM_DEFINE (clear_image, "clear-image", 1, 0, 0,
            (SCM image_smob),
            "Clear the image.")
{
  /* C code to clear the image in image_smob... */
}

void
init_image_type ()
{
#include "image-type.x"
}
```

The `SCM_DEFINE` declaration says that the C function `clear_image` implements a Scheme function called `clear-image`, which takes one required argument (of type `SCM` and named `image_smob`), no optional arguments, and no rest argument. The string `"Clear the image."` provides a short help text for the function, it is called a *docstring*.

For historical reasons, the `SCM_DEFINE` macro also defines a static array of characters named `s_clear_image`, initialized to the string "clear-image". You shouldn't use this array, but you might need to be aware that it exists.

Assuming the text above lives in a file named '`image-type.c`', you will need to execute the following command to prepare this file for compilation:

```
guile-snarf -o image-type.x image-type.c
```

This scans '`image-type.c`' for `SCM_DEFINE` declarations, and writes to '`image-type.x`' the output:

```
scm_c_define_gsubr ("clear-image", 1, 0, 0, (SCM (*)() ) clear_image);
```

When compiled normally, `SCM_DEFINE` is a macro which expands to the function header for `clear_image`.

Note that the output file name matches the `#include` from the input file. Also, you still need to provide all the same information you would if you were using `scm_c_define_gsubr` yourself, but you can place the information near the function definition itself, so it is less likely to become incorrect or out-of-date.

If you have many files that `guile-snarf` must process, you should consider using a fragment like the following in your Makefile:

```
snarfcppopts = $(DEFS) $(INCLUDES) $(CPPFLAGS) $(CFLAGS)
.SUFFIXES: .x
.c.x:
guile-snarf -o $@ $< $(snarfcppopts)
```

This tells make to run `guile-snarf` to produce each needed '`.x`' file from the corresponding '`.c`' file.

The program `guile-snarf` passes its command-line arguments directly to the C preprocessor, which it uses to extract the information it needs from the source code. this means you can pass normal compilation flags to `guile-snarf` to define preprocessor symbols, add header file directories, and so on.

4.6 An Overview of Guile Programming

Guile is designed as an extension language interpreter that is straightforward to integrate with applications written in C (and C++). The big win here for the application developer is that Guile integration, as the Guile web page says, "lowers your project's hacktivation energy." Lowering the hacktivation energy means that you, as the application developer, *and your users*, reap the benefits that flow from being able to extend the application in a high level extension language rather than in plain old C.

In abstract terms, it's difficult to explain what this really means and what the integration process involves, so instead let's begin by jumping straight into an example of how you might integrate Guile into an existing program, and what you could expect to gain by so doing. With that example under our belts, we'll then return to a more general analysis of the arguments involved and the range of programming options available.

4.6.1 How One Might Extend Dia Using Guile

Dia is a free software program for drawing schematic diagrams like flow charts and floor plans (`http://www.gnome.org/projects/dia/`). This section conducts the thought experiment of adding Guile to Dia. In so doing, it aims to illustrate several of the steps and considerations involved in adding Guile to applications in general.

4.6.1.1 Deciding Why You Want to Add Guile

First off, you should understand why you want to add Guile to Dia at all, and that means forming a picture of what Dia does and how it does it. So, what are the constituents of the Dia application?

- Most importantly, the *application domain objects* — in other words, the concepts that differentiate Dia from another application such as a word processor or spreadsheet: shapes, templates, connectors, pages, plus the properties of all these things.
- The code that manages the graphical face of the application, including the layout and display of the objects above.
- The code that handles input events, which indicate that the application user is wanting to do something.

(In other words, a textbook example of the *model - view - controller* paradigm.)

Next question: how will Dia benefit once the Guile integration is complete? Several (positive!) answers are possible here, and the choice is obviously up to the application developers. Still, one answer is that the main benefit will be the ability to manipulate Dia's application domain objects from Scheme.

Suppose that Dia made a set of procedures available in Scheme, representing the most basic operations on objects such as shapes, connectors, and so on. Using Scheme, the application user could then write code that builds upon these basic operations to create more complex procedures. For example, given basic procedures to enumerate the objects on a page, to determine whether an object is a square, and to change the fill pattern of a single shape, the user can write a Scheme procedure to change the fill pattern of all squares on the current page:

```
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
          (change-fill-pattern shape new-pattern)))))
```

4.6.1.2 Four Steps Required to Add Guile

Assuming this objective, four steps are needed to achieve it.

First, you need a way of representing your application-specific objects — such as `shape` in the previous example — when they are passed into the Scheme world. Unless your objects are so simple that they map naturally into builtin Scheme data types like numbers and strings, you will probably want to use Guile's *SMOB* interface to create a new Scheme data type for your objects.

Second, you need to write code for the basic operations like `for-each-shape` and `square?` such that they access and manipulate your existing data structures correctly, and then make these operations available as *primitives* on the Scheme level.

Third, you need to provide some mechanism within the Dia application that a user can hook into to cause arbitrary Scheme code to be evaluated.

Finally, you need to restructure your top-level application C code a little so that it initializes the Guile interpreter correctly and declares your *SMOBs* and *primitives* to the Scheme world.

The following subsections expand on these four points in turn.

4.6.1.3 How to Represent Dia Data in Scheme

For all but the most trivial applications, you will probably want to allow some representation of your domain objects to exist on the Scheme level. This is where the idea of SMOBs comes in, and with it issues of lifetime management and garbage collection.

To get more concrete about this, let's look again at the example we gave earlier of how application users can use Guile to build higher-level functions from the primitives that Dia itself provides.

```
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
          (change-fill-pattern shape new-pattern)))))
```

Consider what is stored here in the variable `shape`. For each shape on the current page, the `for-each-shape` primitive calls `(lambda (shape) ...)` with an argument representing that shape. Question is: how is that argument represented on the Scheme level? The issues are as follows.

- Whatever the representation, it has to be decodable again by the C code for the `square?` and `change-fill-pattern` primitives. In other words, a primitive like `square?` has somehow to be able to turn the value that it receives back into something that points to the underlying C structure describing a shape.
- The representation must also cope with Scheme code holding on to the value for later use. What happens if the Scheme code stores `shape` in a global variable, but then that shape is deleted (in a way that the Scheme code is not aware of), and later on some other Scheme code uses that global variable again in a call to, say, `square?`?

- The lifetime and memory allocation of objects that exist *only* in the Scheme world is managed automatically by Guile's garbage collector using one simple rule: when there are no remaining references to an object, the object is considered dead and so its memory is freed. But for objects that exist in both C and Scheme, the picture is more complicated; in the case of Dia, where the `shape` argument passes transiently in and out of the Scheme world, it would be quite wrong the **delete** the underlying C shape just because the Scheme code has finished evaluation. How do we avoid this happening?

One resolution of these issues is for the Scheme-level representation of a shape to be a new, Scheme-specific C structure wrapped up as a SMOB. The SMOB is what is passed into and out of Scheme code, and the Scheme-specific C structure inside the SMOB points to Dia's underlying C structure so that the code for primitives like `square?` can get at it.

To cope with an underlying shape being deleted while Scheme code is still holding onto a Scheme shape value, the underlying C structure should have a new field that points to the Scheme-specific SMOB. When a shape is deleted, the relevant code chains through to the Scheme-specific structure and sets its pointer back to the underlying structure to NULL. Thus the SMOB value for the shape continues to exist, but any primitive code that tries to use it will detect that the underlying shape has been deleted because the underlying structure pointer is NULL.

So, to summarize the steps involved in this resolution of the problem (and assuming that the underlying C structure for a shape is `struct dia_shape`):

- Define a new Scheme-specific structure that *points* to the underlying C structure:

  ```
  struct dia_guile_shape
  {
    struct dia_shape * c_shape;   /* NULL => deleted */
  }
  ```

- Add a field to `struct dia_shape` that points to its `struct dia_guile_shape` if it has one —

  ```
  struct dia_shape
  {
    ...
    struct dia_guile_shape * guile_shape;
  }
  ```

  — so that C code can set `guile_shape->c_shape` to NULL when the underlying shape is deleted.
- Wrap `struct dia_guile_shape` as a SMOB type.
- Whenever you need to represent a C shape onto the Scheme level, create a SMOB instance for it, and pass that.
- In primitive code that receives a shape SMOB instance, check the `c_shape` field when decoding it, to find out whether the underlying C shape is still there.

As far as memory management is concerned, the SMOB values and their Scheme-specific structures are under the control of the garbage collector, whereas the underlying C structures are explicitly managed in exactly the same way that Dia managed them before we thought of adding Guile.

When the garbage collector decides to free a shape SMOB value, it calls the *SMOB free* function that was specified when defining the shape SMOB type. To maintain the correctness of the `guile_shape` field in the underlying C structure, this function should chain through to the underlying C structure (if it still exists) and set its `guile_shape` field to NULL.

For full documentation on defining and using SMOB types, see Section 4.4 [Defining New Types (Smobs)], page 68.

4.6.1.4 Writing Guile Primitives for Dia

Once the details of object representation are decided, writing the primitive function code that you need is usually straightforward.

A primitive is simply a C function whose arguments and return value are all of type `SCM`, and whose body does whatever you want it to do. As an example, here is a possible implementation of the `square?` primitive:

```
#define FUNC_NAME "square?"
static SCM square_p (SCM shape)
{
  struct dia_guile_shape * guile_shape;

  /* Check that arg is really a shape SMOB. */
  SCM_VALIDATE_SHAPE (SCM_ARG1, shape);

  /* Access Scheme-specific shape structure. */
  guile_shape = SCM_SMOB_DATA (shape);

  /* Find out if underlying shape exists and is a
     square; return answer as a Scheme boolean. */
  return scm_from_bool (guile_shape->c_shape &&
                        (guile_shape->c_shape->type == DIA_SQUARE));
}
#undef FUNC_NAME
```

Notice how easy it is to chain through from the `SCM shape` parameter that `square_p` receives — which is a SMOB — to the Scheme-specific structure inside the SMOB, and thence to the underlying C structure for the shape.

In this code, `SCM_SMOB_DATA` and `scm_from_bool` are from the standard Guile API. `SCM_VALIDATE_SHAPE` is a macro that you should define as part of your SMOB definition: it checks that the passed parameter is of the expected type. This is needed to guard against Scheme code using the `square?` procedure incorrectly, as in `(square? "hello")`; Scheme's latent typing means that usage errors like this must be caught at run time.

Having written the C code for your primitives, you need to make them available as Scheme procedures by calling the `scm_c_define_gsubr` function. `scm_c_define_gsubr` (see Section 5.8.2 [Primitive Procedures], page 226) takes arguments that specify the Scheme-level name for the primitive and how many required, optional and rest arguments it can accept. The `square?` primitive always requires exactly one argument, so the call to make it available in Scheme reads like this:

```
scm_c_define_gsubr ("square?", 1, 0, 0, square_p);
```

For where to put this call, see the subsection after next on the structure of Guile-enabled code (see Section 4.6.1.6 [Dia Structure], page 86).

4.6.1.5 Providing a Hook for the Evaluation of Scheme Code

To make the Guile integration useful, you have to design some kind of hook into your application that application users can use to cause their Scheme code to be evaluated.

Technically, this is straightforward; you just have to decide on a mechanism that is appropriate for your application. Think of Emacs, for example: when you type ⟨ESC⟩ *:*, you get a prompt where you can type in any Elisp code, which Emacs will then evaluate. Or, again like Emacs, you could provide a mechanism (such as an init file) to allow Scheme code to be associated with a particular key sequence, and evaluate the code when that key sequence is entered.

In either case, once you have the Scheme code that you want to evaluate, as a null terminated string, you can tell Guile to evaluate it by calling the `scm_c_eval_string` function.

4.6.1.6 Top-level Structure of Guile-enabled Dia

Let's assume that the pre-Guile Dia code looks structurally like this:

- `main ()`
  - do lots of initialization and setup stuff
  - enter Gtk main loop

When you add Guile to a program, one (rather technical) requirement is that Guile's garbage collector needs to know where the bottom of the C stack is. The easiest way to ensure this is to use `scm_boot_guile` like this:

- `main ()`
  - do lots of initialization and setup stuff
  - `scm_boot_guile (argc, argv, inner_main, NULL)`
- `inner_main ()`
  - define all SMOB types
  - export primitives to Scheme using `scm_c_define_gsubr`
  - enter Gtk main loop

In other words, you move the guts of what was previously in your `main` function into a new function called `inner_main`, and then add a `scm_boot_guile` call, with `inner_main` as a parameter, to the end of `main`.

Assuming that you are using SMOBs and have written primitive code as described in the preceding subsections, you also need to insert calls to declare your new SMOBs and export the primitives to Scheme. These declarations must happen *inside* the dynamic scope of the `scm_boot_guile` call, but also *before* any code is run that could possibly use them — the beginning of `inner_main` is an ideal place for this.

4.6.1.7 Going Further with Dia and Guile

The steps described so far implement an initial Guile integration that already gives a lot of additional power to Dia application users. But there are further steps that you could take, and it's interesting to consider a few of these.

In general, you could progressively move more of Dia's source code from C into Scheme. This might make the code more maintainable and extensible, and it could open the door to new programming paradigms that are tricky to effect in C but straightforward in Scheme.

A specific example of this is that you could use the guile-gtk package, which provides Scheme-level procedures for most of the Gtk+ library, to move the code that lays out and displays Dia objects from C to Scheme.

As you follow this path, it naturally becomes less useful to maintain a distinction between Dia's original non-Guile-related source code, and its later code implementing SMOBs and primitives for the Scheme world.

For example, suppose that the original source code had a `dia_change_fill_pattern` function:

```
void dia_change_fill_pattern (struct dia_shape * shape,
                              struct dia_pattern * pattern)
{
  /* real pattern change work */
}
```

During initial Guile integration, you add a `change_fill_pattern` primitive for Scheme purposes, which accesses the underlying structures from its SMOB values and uses `dia_change_fill_pattern` to do the real work:

```
SCM change_fill_pattern (SCM shape, SCM pattern)
{
  struct dia_shape * d_shape;
  struct dia_pattern * d_pattern;

  ...

  dia_change_fill_pattern (d_shape, d_pattern);

  return SCM_UNSPECIFIED;
}
```

At this point, it makes sense to keep `dia_change_fill_pattern` and `change_fill_pattern` separate, because `dia_change_fill_pattern` can also be called without going through Scheme at all, say because the user clicks a button which causes a C-registered Gtk+ callback to be called.

But, if the code for creating buttons and registering their callbacks is moved into Scheme (using guile-gtk), it may become true that `dia_change_fill_pattern` can no longer be called other than through Scheme. In which case, it makes sense to abolish it and move its contents directly into `change_fill_pattern`, like this:

```
SCM change_fill_pattern (SCM shape, SCM pattern)
{
      struct dia_shape * d_shape;
      struct dia_pattern * d_pattern;

      ...

      /* real pattern change work */

      return SCM_UNSPECIFIED;
    }
```

So further Guile integration progressively *reduces* the amount of functional C code that you have to maintain over the long term.

A similar argument applies to data representation. In the discussion of SMOBs earlier, issues arose because of the different memory management and lifetime models that normally apply to data structures in C and in Scheme. However, with further Guile integration, you can resolve this issue in a more radical way by allowing all your data structures to be under the control of the garbage collector, and kept alive by references from the Scheme world. Instead of maintaining an array or linked list of shapes in C, you would instead maintain a list in Scheme.

Rather like the coalescing of `dia_change_fill_pattern` and `change_fill_pattern`, the practical upshot of such a change is that you would no longer have to keep the `dia_shape` and `dia_guile_shape` structures separate, and so wouldn't need to worry about the pointers between them. Instead, you could change the SMOB definition to wrap the `dia_shape` structure directly, and send `dia_guile_shape` off to the scrap yard. Cut out the middle man!

Finally, we come to the holy grail of Guile's free software / extension language approach. Once you have a Scheme representation for interesting Dia data types like shapes, and a handy bunch of primitives for manipulating them, it suddenly becomes clear that you have a bundle of functionality that could have far-ranging use beyond Dia itself. In other words, the data types and primitives could now become a library, and Dia becomes just one of the many possible applications using that library — albeit, at this early stage, a rather important one!

In this model, Guile becomes just the glue that binds everything together. Imagine an application that usefully combined functionality from Dia, Gnumeric and GnuCash — it's tricky right now, because no such application yet exists; but it'll happen some day . . .

4.6.2 Why Scheme is More Hackable Than C

Underlying Guile's value proposition is the assumption that programming in a high level language, specifically Guile's implementation of Scheme, is necessarily better in some way than programming in C. What do we mean by this claim, and how can we be so sure?

One class of advantages applies not only to Scheme, but more generally to any interpretable, high level, scripting language, such as Emacs Lisp, Python, Ruby, or TeX's macro language. Common features of all such languages, when compared to C, are that:

- They lend themselves to rapid and experimental development cycles, owing usually to a combination of their interpretability and the integrated development environment in which they are used.

- They free developers from some of the low level bookkeeping tasks associated with C programming, notably memory management.
- They provide high level features such as container objects and exception handling that make common programming tasks easier.

In the case of Scheme, particular features that make programming easier — and more fun! — are its powerful mechanisms for abstracting parts of programs (closures — see Section 3.1.4 [About Closure], page 24) and for iteration (see Section 5.11.4 [while do], page 252).

The evidence in support of this argument is empirical: the huge amount of code that has been written in extension languages for applications that support this mechanism. Most notable are extensions written in Emacs Lisp for GNU Emacs, in TEX's macro language for TEX, and in Script-Fu for the Gimp, but there is increasingly now a significant code eco-system for Guile-based applications as well, such as Lilypond and GnuCash. It is close to inconceivable that similar amounts of functionality could have been added to these applications just by writing new code in their base implementation languages.

4.6.3 Example: Using Guile for an Application Testbed

As an example of what this means in practice, imagine writing a testbed for an application that is tested by submitting various requests (via a C interface) and validating the output received. Suppose further that the application keeps an idea of its current state, and that the "correct" output for a given request may depend on the current application state. A complete "white box"[1] test plan for this application would aim to submit all possible requests in each distinguishable state, and validate the output for all request/state combinations.

To write all this test code in C would be very tedious. Suppose instead that the testbed code adds a single new C function, to submit an arbitrary request and return the response, and then uses Guile to export this function as a Scheme procedure. The rest of the testbed can then be written in Scheme, and so benefits from all the advantages of programming in Scheme that were described in the previous section.

(In this particular example, there is an additional benefit of writing most of the testbed in Scheme. A common problem for white box testing is that mistakes and mistaken assumptions in the application under test can easily be reproduced in the testbed code. It is more difficult to copy mistakes like this when the testbed is written in a different language from the application.)

4.6.4 A Choice of Programming Options

The preceding arguments and example point to a model of Guile programming that is applicable in many cases. According to this model, Guile programming involves a balance between C and Scheme programming, with the aim being to extract the greatest possible Scheme level benefit from the least amount of C level work.

The C level work required in this model usually consists of packaging and exporting functions and application objects such that they can be seen and manipulated on the Scheme

[1] A *white box* test plan is one that incorporates knowledge of the internal design of the application under test.

level. To help with this, Guile's C language interface includes utility features that aim to make this kind of integration very easy for the application developer. These features are documented later in this part of the manual: see REFFIXME.

This model, though, is really just one of a range of possible programming options. If all of the functionality that you need is available from Scheme, you could choose instead to write your whole application in Scheme (or one of the other high level languages that Guile supports through translation), and simply use Guile as an interpreter for Scheme. (In the future, we hope that Guile will also be able to compile Scheme code, so lessening the performance gap between C and Scheme code.) Or, at the other end of the C–Scheme scale, you could write the majority of your application in C, and only call out to Guile occasionally for specific actions such as reading a configuration file or executing a user-specified extension. The choices boil down to two basic questions:

- Which parts of the application do you write in C, and which in Scheme (or another high level translated language)?
- How do you design the interface between the C and Scheme parts of your application?

These are of course design questions, and the right design for any given application will always depend upon the particular requirements that you are trying to meet. In the context of Guile, however, there are some generally applicable considerations that can help you when designing your answers.

4.6.4.1 What Functionality is Already Available?

Suppose, for the sake of argument, that you would prefer to write your whole application in Scheme. Then the API available to you consists of:

- standard Scheme
- plus the extensions to standard Scheme provided by Guile in its core distribution
- plus any additional functionality that you or others have packaged so that it can be loaded as a Guile Scheme module.

A module in the last category can either be a pure Scheme module — in other words a collection of utility procedures coded in Scheme — or a module that provides a Scheme interface to an extension library coded in C — in other words a nice package where someone else has done the work of wrapping up some useful C code for you. The set of available modules is growing quickly and already includes such useful examples as `(gtk gtk)`, which makes Gtk+ drawing functions available in Scheme, and `(database postgres)`, which provides SQL access to a Postgres database.

Given the growing collection of pre-existing modules, it is quite feasible that your application could be implemented by combining a selection of these modules together with new application code written in Scheme.

If this approach is not enough, because the functionality that your application needs is not already available in this form, and it is impossible to write the new functionality in Scheme, you will need to write some C code. If the required function is already available in C (e.g. in a library), all you need is a little glue to connect it to the world of Guile. If not, you need both to write the basic code and to plumb it into Guile.

In either case, two general considerations are important. Firstly, what is the interface by which the functionality is presented to the Scheme world? Does the interface consist only of function calls (for example, a simple drawing interface), or does it need to include *objects* of some kind that can be passed between C and Scheme and manipulated by both worlds. Secondly, how does the lifetime and memory management of objects in the C code relate to the garbage collection governed approach of Scheme objects? In the case where the basic C code is not already written, most of the difficulties of memory management can be avoided by using Guile's C interface features from the start.

For the full documentation on writing C code for Guile and connecting existing C code to the Guile world, see REFFIXME.

4.6.4.2 Functional and Performance Constraints

4.6.4.3 Your Preferred Programming Style

4.6.4.4 What Controls Program Execution?

4.6.5 How About Application Users?

So far we have considered what Guile programming means for an application developer. But what if you are instead *using* an existing Guile-based application, and want to know what your options are for programming and extending this application?

The answer to this question varies from one application to another, because the options available depend inevitably on whether the application developer has provided any hooks for you to hang your own code on and, if there are such hooks, what they allow you to do.[2] For example...

- If the application permits you to load and execute any Guile code, the world is your oyster. You can extend the application in any way that you choose.
- A more cautious application might allow you to load and execute Guile code, but only in a *safe* environment, where the interface available is restricted by the application from the standard Guile API.
- Or a really fearful application might not provide a hook to really execute user code at all, but just use Scheme syntax as a convenient way for users to specify application data or configuration options.

In the last two cases, what you can do is, by definition, restricted by the application, and you should refer to the application's own manual to find out your options.

The most well known example of the first case is Emacs, with its extension language Emacs Lisp: as well as being a text editor, Emacs supports the loading and execution of arbitrary Emacs Lisp code. The result of such openness has been dramatic: Emacs now benefits from user-contributed Emacs Lisp libraries that extend the basic editing function to do everything from reading news to psychoanalysis and playing adventure games. The only limitation is that extensions are restricted to the functionality provided by Emacs's built-in set of primitive operations. For example, you can interact and display data by manipulating the contents of an Emacs buffer, but you can't pop-up and draw a window with a layout that is totally different to the Emacs standard.

[2] Of course, in the world of free software, you always have the freedom to modify the application's source code to your own requirements. Here we are concerned with the extension options that the application has provided for without your needing to modify its source code.

This situation with a Guile application that supports the loading of arbitrary user code is similar, except perhaps even more so, because Guile also supports the loading of extension libraries written in C. This last point enables user code to add new primitive operations to Guile, and so to bypass the limitation present in Emacs Lisp.

At this point, the distinction between an application developer and an application user becomes rather blurred. Instead of seeing yourself as a user extending an application, you could equally well say that you are developing a new application of your own using some of the primitive functionality provided by the original application. As such, all the discussions of the preceding sections of this chapter are relevant to how you can proceed with developing your extension.

5 API Reference

Guile provides an application programming interface (*API*) to developers in two core languages: Scheme and C. This part of the manual contains reference documentation for all of the functionality that is available through both Scheme and C interfaces.

5.1 Overview of the Guile API

Guile's application programming interface (*API*) makes functionality available that an application developer can use in either C or Scheme programming. The interface consists of *elements* that may be macros, functions or variables in C, and procedures, variables, syntax or other types of object in Scheme.

Many elements are available to both Scheme and C, in a form that is appropriate. For example, the `assq` Scheme procedure is also available as `scm_assq` to C code. These elements are documented only once, addressing both the Scheme and C aspects of them.

The Scheme name of an element is related to its C name in a regular way. Also, a C function takes its parameters in a systematic way.

Normally, the name of a C function can be derived given its Scheme name, using some simple textual transformations:

- Replace `-` (hyphen) with `_` (underscore).
- Replace `?` (question mark) with `_p`.
- Replace `!` (exclamation point) with `_x`.
- Replace internal `->` with `_to_`.
- Replace `<=` (less than or equal) with `_leq`.
- Replace `>=` (greater than or equal) with `_geq`.
- Replace `<` (less than) with `_less`.
- Replace `>` (greater than) with `_gr`.
- Prefix with `scm_`.

A C function always takes a fixed number of arguments of type `SCM`, even when the corresponding Scheme function takes a variable number.

For some Scheme functions, some last arguments are optional; the corresponding C function must always be invoked with all optional arguments specified. To get the effect as if an argument has not been specified, pass `SCM_UNDEFINED` as its value. You can not do this for an argument in the middle; when one argument is `SCM_UNDEFINED` all the ones following it must be `SCM_UNDEFINED` as well.

Some Scheme functions take an arbitrary number of *rest* arguments; the corresponding C function must be invoked with a list of all these arguments. This list is always the last argument of the C function.

These two variants can also be combined.

The type of the return value of a C function that corresponds to a Scheme function is always `SCM`. In the descriptions below, types are therefore often omitted bot for the return value and for the arguments.

5.2 The SCM Type

Guile represents all Scheme values with the single C type `SCM`. For an introduction to this topic, See Section 4.3.1 [Dynamic Types], page 60.

`SCM` [C Type]

`SCM` is the user level abstract C type that is used to represent all of Guile's Scheme objects, no matter what the Scheme object type is. No C operation except assignment is guaranteed to work with variables of type `SCM`, so you should only use macros and functions to work with `SCM` values. Values are converted between C data types and the `SCM` type with utility functions and macros.

`scm_t_bits` [C Type]
: `scm_t_bits` is an unsigned integral data type that is guaranteed to be large enough to hold all information that is required to represent any Scheme object. While this data type is mostly used to implement Guile's internals, the use of this type is also necessary to write certain kinds of extensions to Guile.

`scm_t_signed_bits` [C Type]
: This is a signed integral type of the same size as `scm_t_bits`.

`scm_t_bits SCM_UNPACK` (*SCM* `x`) [C Macro]
: Transforms the `SCM` value *x* into its representation as an integral type. Only after applying `SCM_UNPACK` it is possible to access the bits and contents of the `SCM` value.

`SCM SCM_PACK` (*scm_t_bits* `x`) [C Macro]
: Takes a valid integral representation of a Scheme object and transforms it into its representation as a `SCM` value.

5.3 Initializing Guile

Each thread that wants to use functions from the Guile API needs to put itself into guile mode with either `scm_with_guile` or `scm_init_guile`. The global state of Guile is initialized automatically when the first thread enters guile mode.

When a thread wants to block outside of a Guile API function, it should leave guile mode temporarily with `scm_without_guile`, See Section 5.17.6 [Blocking], page 329.

Threads that are created by `call-with-new-thread` or `scm_spawn_thread` start out in guile mode so you don't need to initialize them.

`void * scm_with_guile` (*void *(*func)(void *), void *data*) [C Function]
: Call *func*, passing it *data* and return what *func* returns. While *func* is running, the current thread is in guile mode and can thus use the Guile API.

  When `scm_with_guile` is called from guile mode, the thread remains in guile mode when `scm_with_guile` returns.

  Otherwise, it puts the current thread into guile mode and, if needed, gives it a Scheme representation that is contained in the list returned by `all-threads`, for example. This Scheme representation is not removed when `scm_with_guile` returns so that a given thread is always represented by the same Scheme value during its lifetime, if at all.

  When this is the first thread that enters guile mode, the global state of Guile is initialized before calling `func`.

  The function *func* is called via `scm_with_continuation_barrier`; thus, `scm_with_guile` returns exactly once.

  When `scm_with_guile` returns, the thread is no longer in guile mode (except when `scm_with_guile` was called from guile mode, see above). Thus, only `func` can store

`SCM` variables on the stack and be sure that they are protected from the garbage collector. See `scm_init_guile` for another approach at initializing Guile that does not have this restriction.

It is OK to call `scm_with_guile` while a thread has temporarily left guile mode via `scm_without_guile`. It will then simply temporarily enter guile mode again.

`void scm_init_guile ()` [C Function]

Arrange things so that all of the code in the current thread executes as if from within a call to `scm_with_guile`. That is, all functions called by the current thread can assume that `SCM` values on their stack frames are protected from the garbage collector (except when the thread has explicitely left guile mode, of course).

When `scm_init_guile` is called from a thread that already has been in guile mode once, nothing happens. This behavior matters when you call `scm_init_guile` while the thread has only temporarily left guile mode: in that case the thread will not be in guile mode after `scm_init_guile` returns. Thus, you should not use `scm_init_guile` in such a scenario.

When a uncaught throw happens in a thread that has been put into guile mode via `scm_init_guile`, a short message is printed to the current error port and the thread is exited via `scm_pthread_exit (NULL)`. No restrictions are placed on continuations.

The function `scm_init_guile` might not be available on all platforms since it requires some stack-bounds-finding magic that might not have been ported to all platforms that Guile runs on. Thus, if you can, it is better to use `scm_with_guile` or its variation `scm_boot_guile` instead of this function.

`void scm_boot_guile` (*int* `argc`, *char* `**argv`, *void* (`*main_func`) (*void* `*data`, *int* `argc`, *char* `**argv`), *void* `*data`) [C Function]

Enter guile mode as with `scm_with_guile` and call *main_func*, passing it *data*, *argc*, and *argv* as indicated. When *main_func* returns, `scm_boot_guile` calls `exit (0)`; `scm_boot_guile` never returns. If you want some other exit value, have *main_func* call `exit` itself. If you don't want to exit at all, use `scm_with_guile` instead of `scm_boot_guile`.

The function `scm_boot_guile` arranges for the Scheme `command-line` function to return the strings given by *argc* and *argv*. If *main_func* modifies *argc* or *argv*, it should call `scm_set_program_arguments` with the final list, so Scheme code will know which arguments have been processed.

`void scm_shell` (*int* `argc`, *char* `**argv`) [C Function]

Process command-line arguments in the manner of the `guile` executable. This includes loading the normal Guile initialization files, interacting with the user or running any scripts or expressions specified by `-s` or `-e` options, and then exiting. See Section 3.3.2 [Invoking Guile], page 33, for more details.

Since this function does not return, you must do all application-specific initialization before calling this function.

5.4 Snarfing Macros

The following macros do two different things: when compiled normally, they expand in one way; when processed during snarfing, they cause the `guile-snarf` program to pick up some initialization code, See Section 4.5 [Function Snarfing], page 79.

The descriptions below use the term 'normally' to refer to the case when the code is compiled normally, and 'while snarfing' when the code is processed by `guile-snarf`.

`SCM_SNARF_INIT` (*code*) [C Macro]

Normally, `SCM_SNARF_INIT` expands to nothing; while snarfing, it causes *code* to be included in the initialization action file, followed by a semicolon.

This is the fundamental macro for snarfing initialization actions. The more specialized macros below use it internally.

`SCM_DEFINE` (*c_name*, *scheme_name*, *req*, *opt*, *var*, *arglist*, *docstring*) [C Macro]

Normally, this macro expands into

```
static const char s_c_name[] = scheme_name;
SCM
c_name arglist
```

While snarfing, it causes

```
scm_c_define_gsubr (s_c_name, req, opt, var,
                    c_name);
```

to be added to the initialization actions. Thus, you can use it to declare a C function named *c_name* that will be made available to Scheme with the name *scheme_name*.

Note that the *arglist* argument must have parentheses around it.

`SCM_SYMBOL` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_SYMBOL` (*c_name*, *scheme_name*) [C Macro]

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_from_locale_symbol (scheme_name));
```

Thus, you can use them declare a static or global variable of type `SCM` that will be initialized to the symbol named *scheme_name*.

`SCM_KEYWORD` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_KEYWORD` (*c_name*, *scheme_name*) [C Macro]

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_c_make_keyword (scheme_name));
```

Thus, you can use them declare a static or global variable of type `SCM` that will be initialized to the keyword named *scheme_name*.

`SCM_VARIABLE` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_VARIABLE` (*c_name*, *scheme_name*) [C Macro]

These macros are equivalent to `SCM_VARIABLE_INIT` and `SCM_GLOBAL_VARIABLE_INIT`, respectively, with a *value* of `SCM_BOOL_F`.

`SCM_VARIABLE_INIT` (*c_name*, *scheme_name*, *value*) [C Macro]
`SCM_GLOBAL_VARIABLE_INIT` (*c_name*, *scheme_name*, *value*) [C Macro]

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_c_define (scheme_name, value));
```

Thus, you can use them declare a static or global C variable of type `SCM` that will be initialized to the object representing the Scheme variable named *scheme_name* in the current module. The variable will be defined when it doesn't already exist. It is always set to *value*.

5.5 Simple Generic Data Types

This chapter describes those of Guile's simple data types which are primarily used for their role as items of generic data. By *simple* we mean data types that are not primarily used as containers to hold other data — i.e. pairs, lists, vectors and so on. For the documentation of such *compound* data types, see Section 5.6 [Compound Data Types], page 166.

5.5.1 Booleans

The two boolean values are `#t` for true and `#f` for false.

Boolean values are returned by predicate procedures, such as the general equality predicates `eq?`, `eqv?` and `equal?` (see Section 5.9.1 [Equality], page 236) and numerical and string comparison operators like `string=?` (see Section 5.5.5.7 [String Comparison], page 136) and `<=` (see Section 5.5.2.8 [Comparison], page 111).

```
(<= 3 8)
⇒ #t

(<= 3 -3)
⇒ #f

(equal? "house" "houses")
⇒ #f

(eq? #f #f)
⇒
#t
```

In test condition contexts like `if` and `cond` (see Section 5.11.2 [if cond case], page 251), where a group of subexpressions will be evaluated only if a *condition* expression evaluates to "true", "true" means any value at all except `#f`.

```
(if #t "yes" "no")
⇒ "yes"

(if 0 "yes" "no")
⇒ "yes"

(if #f "yes" "no")
⇒ "no"
```

A result of this asymmetry is that typical Scheme source code more often uses `#f` explicitly than `#t`: `#f` is necessary to represent an `if` or `cond` false value, whereas `#t` is not necessary to represent an `if` or `cond` true value.

It is important to note that `#f` is **not** equivalent to any other Scheme value. In particular, `#f` is not the same as the number 0 (like in C and C++), and not the same as the "empty list" (like in some Lisp dialects).

In C, the two Scheme boolean values are available as the two constants `SCM_BOOL_T` for `#t` and `SCM_BOOL_F` for `#f`. Care must be taken with the false value `SCM_BOOL_F`: it is not false when used in C conditionals. In order to test for it, use `scm_is_false` or `scm_is_true`.

`not` *x* [Scheme Procedure]
`scm_not` (*x*) [C Function]

Return `#t` if *x* is `#f`, else return `#f`.

`boolean?` *obj* [Scheme Procedure]
`scm_boolean_p` (*obj*) [C Function]

Return `#t` if *obj* is either `#t` or `#f`, else return `#f`.

`SCM SCM_BOOL_T` [C Macro]

The `SCM` representation of the Scheme object `#t`.

`SCM SCM_BOOL_F` [C Macro]

The `SCM` representation of the Scheme object `#f`.

`int scm_is_true` (*SCM obj*) [C Function]

Return `0` if *obj* is `#f`, else return `1`.

`int scm_is_false` (*SCM obj*) [C Function]

Return `1` if *obj* is `#f`, else return `0`.

`int scm_is_bool` (*SCM obj*) [C Function]

Return `1` if *obj* is either `#t` or `#f`, else return `0`.

`SCM scm_from_bool` (*int val*) [C Function]

Return `#f` if *val* is `0`, else return `#t`.

`int scm_to_bool` (*SCM val*) [C Function]

Return `1` if *val* is `SCM_BOOL_T`, return `0` when *val* is `SCM_BOOL_F`, else signal a 'wrong type' error.

You should probably use `scm_is_true` instead of this function when you just want to test a `SCM` value for trueness.

5.5.2 Numerical data types

Guile supports a rich "tower" of numerical types — integer, rational, real and complex — and provides an extensive set of mathematical and scientific functions for operating on numerical data. This section of the manual documents those types and functions.

You may also find it illuminating to read R5RS's presentation of numbers in Scheme, which is particularly clear and accessible: see section "Numbers" in *R5RS*.

5.5.2.1 Scheme's Numerical "Tower"

Scheme's numerical "tower" consists of the following categories of numbers:

integers Whole numbers, positive or negative; e.g. –5, 0, 18.

rationals The set of numbers that can be expressed as p/q where p and q are integers; e.g. 9/16 works, but pi (an irrational number) doesn't. These include integers ($n/1$).

real numbers
The set of numbers that describes all possible positions along a one-dimensional line. This includes rationals as well as irrational numbers.

complex numbers
: The set of numbers that describes all possible positions in a two dimensional space. This includes real as well as imaginary numbers ($a + bi$, where a is the *real part*, b is the *imaginary part*, and i is the square root of -1.)

It is called a tower because each category "sits on" the one that follows it, in the sense that every integer is also a rational, every rational is also real, and every real number is also a complex number (but with zero imaginary part).

In addition to the classification into integers, rationals, reals and complex numbers, Scheme also distinguishes between whether a number is represented exactly or not. For example, the result of $2\sin(\pi/4)$ is exactly $\sqrt{2}$, but Guile can represent neither $\pi/4$ nor $\sqrt{2}$ exactly. Instead, it stores an inexact approximation, using the C type `double`.

Guile can represent exact rationals of any magnitude, inexact rationals that fit into a C `double`, and inexact complex numbers with `double` real and imaginary parts.

The `number?` predicate may be applied to any Scheme value to discover whether the value is any of the supported numerical types.

`number?` *obj* [Scheme Procedure]
`scm_number_p` (*obj*) [C Function]

Return `#t` if *obj* is any kind of number, else `#f`.

For example:

```
(number? 3)
⇒ #t

(number? "hello there!")
⇒ #f

(define pi 3.141592654)
(number? pi)
⇒ #t
```

`int scm_is_number` (*SCM obj*) [C Function]

This is equivalent to `scm_is_true (scm_number_p (obj))`.

The next few subsections document each of Guile's numerical data types in detail.

5.5.2.2 Integers

Integers are whole numbers, that is numbers with no fractional part, such as 2, 83, and −3789.

Integers in Guile can be arbitrarily big, as shown by the following example.

```
(define (factorial n)
  (let loop ((n n) (product 1))
    (if (= n 0)
        product
        (loop (- n 1) (* product n)))))
```

```
(factorial 3)
⇒ 6

(factorial 20)
⇒ 2432902008176640000

(- (factorial 45))
⇒ -119622220865480194561963161495657715064383733760000000000
```

Readers whose background is in programming languages where integers are limited by the need to fit into just 4 or 8 bytes of memory may find this surprising, or suspect that Guile's representation of integers is inefficient. In fact, Guile achieves a near optimal balance of convenience and efficiency by using the host computer's native representation of integers where possible, and a more general representation where the required number does not fit in the native form. Conversion between these two representations is automatic and completely invisible to the Scheme level programmer.

The infinities '`+inf.0`' and '`-inf.0`' are considered to be inexact integers. They are explained in detail in the next section, together with reals and rationals.

C has a host of different integer types, and Guile offers a host of functions to convert between them and the `SCM` representation. For example, a C `int` can be handled with `scm_to_int` and `scm_from_int`. Guile also defines a few C integer types of its own, to help with differences between systems.

C integer types that are not covered can be handled with the generic `scm_to_signed_integer` and `scm_from_signed_integer` for signed types, or with `scm_to_unsigned_integer` and `scm_from_unsigned_integer` for unsigned types.

Scheme integers can be exact and inexact. For example, a number written as `3.0` with an explicit decimal-point is inexact, but it is also an integer. The functions `integer?` and `scm_is_integer` report true for such a number, but the functions `scm_is_signed_integer` and `scm_is_unsigned_integer` only allow exact integers and thus report false. Likewise, the conversion functions like `scm_to_signed_integer` only accept exact integers.

The motivation for this behavior is that the inexactness of a number should not be lost silently. If you want to allow inexact integers, you can explicitely insert a call to `inexact->exact` or to its C equivalent `scm_inexact_to_exact`. (Only inexact integers will be converted by this call into exact integers; inexact non-integers will become exact fractions.)

`integer?` *x* [Scheme Procedure]
`scm_integer_p` (*x*) [C Function]

Return `#t` if *x* is an exact or inexact integer number, else `#f`.

```
(integer? 487)
⇒ #t

(integer? 3.0)
⇒ #t

(integer? -3.4)
⇒ #f
```

```
(integer? +inf.0)
⇒ #t
```

`int scm_is_integer` (*SCM x*) [C Function]

This is equivalent to `scm_is_true (scm_integer_p (x))`.

`scm_t_int8` [C Type]
`scm_t_uint8` [C Type]
`scm_t_int16` [C Type]
`scm_t_uint16` [C Type]
`scm_t_int32` [C Type]
`scm_t_uint32` [C Type]
`scm_t_int64` [C Type]
`scm_t_uint64` [C Type]
`scm_t_intmax` [C Type]
`scm_t_uintmax` [C Type]

The C types are equivalent to the corresponding ISO C types but are defined on all platforms, with the exception of `scm_t_int64` and `scm_t_uint64`, which are only defined when a 64-bit type is available. For example, `scm_t_int8` is equivalent to `int8_t`.

You can regard these definitions as a stop-gap measure until all platforms provide these types. If you know that all the platforms that you are interested in already provide these types, it is better to use them directly instead of the types provided by Guile.

`int scm_is_signed_integer` (*SCM x, scm_t_intmax min, scm_t_intmax max*) [C Function]
`int scm_is_unsigned_integer` (*SCM x, scm_t_uintmax min, scm_t_uintmax max*) [C Function]

Return `1` when *x* represents an exact integer that is between *min* and *max*, inclusive.

These functions can be used to check whether a `SCM` value will fit into a given range, such as the range of a given C integer type. If you just want to convert a `SCM` value to a given C integer type, use one of the conversion functions directly.

`scm_t_intmax scm_to_signed_integer` (*SCM x, scm_t_intmax min, scm_t_intmax max*) [C Function]
`scm_t_uintmax scm_to_unsigned_integer` (*SCM x, scm_t_uintmax min, scm_t_uintmax max*) [C Function]

When *x* represents an exact integer that is between *min* and *max* inclusive, return that integer. Else signal an error, either a 'wrong-type' error when *x* is not an exact integer, or an 'out-of-range' error when it doesn't fit the given range.

`SCM scm_from_signed_integer` (*scm_t_intmax x*) [C Function]
`SCM scm_from_unsigned_integer` (*scm_t_uintmax x*) [C Function]

Return the `SCM` value that represents the integer *x*. This function will always succeed and will always return an exact number.

`char scm_to_char` (*SCM x*) [C Function]
`signed char scm_to_schar` (*SCM x*) [C Function]
`unsigned char scm_to_uchar` (*SCM x*) [C Function]
`short scm_to_short` (*SCM x*) [C Function]
`unsigned short scm_to_ushort` (*SCM x*) [C Function]
`int scm_to_int` (*SCM x*) [C Function]
`unsigned int scm_to_uint` (*SCM x*) [C Function]
`long scm_to_long` (*SCM x*) [C Function]
`unsigned long scm_to_ulong` (*SCM x*) [C Function]
`long long scm_to_long_long` (*SCM x*) [C Function]
`unsigned long long scm_to_ulong_long` (*SCM x*) [C Function]
`size_t scm_to_size_t` (*SCM x*) [C Function]
`ssize_t scm_to_ssize_t` (*SCM x*) [C Function]
`scm_t_int8 scm_to_int8` (*SCM x*) [C Function]
`scm_t_uint8 scm_to_uint8` (*SCM x*) [C Function]
`scm_t_int16 scm_to_int16` (*SCM x*) [C Function]
`scm_t_uint16 scm_to_uint16` (*SCM x*) [C Function]
`scm_t_int32 scm_to_int32` (*SCM x*) [C Function]
`scm_t_uint32 scm_to_uint32` (*SCM x*) [C Function]
`scm_t_int64 scm_to_int64` (*SCM x*) [C Function]
`scm_t_uint64 scm_to_uint64` (*SCM x*) [C Function]
`scm_t_intmax scm_to_intmax` (*SCM x*) [C Function]
`scm_t_uintmax scm_to_uintmax` (*SCM x*) [C Function]

When *x* represents an exact integer that fits into the indicated C type, return that integer. Else signal an error, either a 'wrong-type' error when *x* is not an exact integer, or an 'out-of-range' error when it doesn't fit the given range.

The functions `scm_to_long_long`, `scm_to_ulong_long`, `scm_to_int64`, and `scm_to_uint64` are only available when the corresponding types are.

`SCM scm_from_char` (*char x*) [C Function]
`SCM scm_from_schar` (*signed char x*) [C Function]
`SCM scm_from_uchar` (*unsigned char x*) [C Function]
`SCM scm_from_short` (*short x*) [C Function]
`SCM scm_from_ushort` (*unsigned short x*) [C Function]
`SCM scm_from_int` (*int x*) [C Function]
`SCM scm_from_uint` (*unsigned int x*) [C Function]
`SCM scm_from_long` (*long x*) [C Function]
`SCM scm_from_ulong` (*unsigned long x*) [C Function]
`SCM scm_from_long_long` (*long long x*) [C Function]
`SCM scm_from_ulong_long` (*unsigned long long x*) [C Function]
`SCM scm_from_size_t` (*size_t x*) [C Function]
`SCM scm_from_ssize_t` (*ssize_t x*) [C Function]
`SCM scm_from_int8` (*scm_t_int8 x*) [C Function]
`SCM scm_from_uint8` (*scm_t_uint8 x*) [C Function]
`SCM scm_from_int16` (*scm_t_int16 x*) [C Function]
`SCM scm_from_uint16` (*scm_t_uint16 x*) [C Function]
`SCM scm_from_int32` (*scm_t_int32 x*) [C Function]

`SCM scm_from_uint32` (*scm_t_uint32 x*) [C Function]
`SCM scm_from_int64` (*scm_t_int64 x*) [C Function]
`SCM scm_from_uint64` (*scm_t_uint64 x*) [C Function]
`SCM scm_from_intmax` (*scm_t_intmax x*) [C Function]
`SCM scm_from_uintmax` (*scm_t_uintmax x*) [C Function]

Return the `SCM` value that represents the integer *x*. These functions will always succeed and will always return an exact number.

`void scm_to_mpz` (*SCM val, mpz_t rop*) [C Function]

Assign *val* to the multiple precision integer *rop*. *val* must be an exact integer, otherwise an error will be signalled. *rop* must have been initialized with `mpz_init` before this function is called. When *rop* is no longer needed the occupied space must be freed with `mpz_clear`. See section "Initializing Integers" in *GNU MP Manual*, for details.

`SCM scm_from_mpz` (*mpz_t val*) [C Function]

Return the `SCM` value that represents *val*.

5.5.2.3 Real and Rational Numbers

Mathematically, the real numbers are the set of numbers that describe all possible points along a continuous, infinite, one-dimensional line. The rational numbers are the set of all numbers that can be written as fractions p/q, where p and q are integers. All rational numbers are also real, but there are real numbers that are not rational, for example $\sqrt{2}$, and π.

Guile can represent both exact and inexact rational numbers, but it can not represent irrational numbers. Exact rationals are represented by storing the numerator and denominator as two exact integers. Inexact rationals are stored as floating point numbers using the C type `double`.

Exact rationals are written as a fraction of integers. There must be no whitespace around the slash:

```
1/2
-22/7
```

Even though the actual encoding of inexact rationals is in binary, it may be helpful to think of it as a decimal number with a limited number of significant figures and a decimal point somewhere, since this corresponds to the standard notation for non-whole numbers. For example:

```
0.34
-0.00000142857931198
-5648394822220000000000.0
4.0
```

The limited precision of Guile's encoding means that any "real" number in Guile can be written in a rational form, by multiplying and then dividing by sufficient powers of 10 (or in fact, 2). For example, '`-0.00000142857931198`' is the same as -142857931198 divided by 100000000000000000. In Guile's current incarnation, therefore, the `rational?` and `real?` predicates are equivalent.

Dividing by an exact zero leads to a error message, as one might expect. However, dividing by an inexact zero does not produce an error. Instead, the result of the division is either plus or minus infinity, depending on the sign of the divided number.

The infinities are written '`+inf.0`' and '`-inf.0`', respectivly. This syntax is also recognized by `read` as an extension to the usual Scheme syntax.

Dividing zero by zero yields something that is not a number at all: '`+nan.0`'. This is the special 'not a number' value.

On platforms that follow IEEE 754 for their floating point arithmetic, the '`+inf.0`', '`-inf.0`', and '`+nan.0`' values are implemented using the corresponding IEEE 754 values. They behave in arithmetic operations like IEEE 754 describes it, i.e., `(= +nan.0 +nan.0)` ⇒ `#f`.

The infinities are inexact integers and are considered to be both even and odd. While '`+nan.0`' is not `=` to itself, it is `eqv?` to itself.

To test for the special values, use the functions `inf?` and `nan?`.

`real?` *obj* [Scheme Procedure]
`scm_real_p` (*obj*) [C Function]

Return `#t` if *obj* is a real number, else `#f`. Note that the sets of integer and rational values form subsets of the set of real numbers, so the predicate will also be fulfilled if *obj* is an integer number or a rational number.

`rational?` *x* [Scheme Procedure]
`scm_rational_p` (*x*) [C Function]

Return `#t` if *x* is a rational number, `#f` otherwise. Note that the set of integer values forms a subset of the set of rational numbers, i. e. the predicate will also be fulfilled if *x* is an integer number.

Since Guile can not represent irrational numbers, every number satisfying `real?` also satisfies `rational?` in Guile.

`rationalize` *x eps* [Scheme Procedure]
`scm_rationalize` (*x, eps*) [C Function]

Returns the *simplest* rational number differing from *x* by no more than *eps*.

As required by R5RS, `rationalize` only returns an exact result when both its arguments are exact. Thus, you might need to use `inexact->exact` on the arguments.

```
(rationalize (inexact->exact 1.2) 1/100)
⇒ 6/5
```

`inf?` *x* [Scheme Procedure]
`scm_inf_p` (*x*) [C Function]

Return `#t` if *x* is either '`+inf.0`' or '`-inf.0`', `#f` otherwise.

`nan?` *x* [Scheme Procedure]
`scm_nan_p` (*x*) [C Function]

Return `#t` if *x* is '`+nan.0`', `#f` otherwise.

`nan` [Scheme Procedure]
`scm_nan` () [C Function]

Return NaN.

`inf` [Scheme Procedure]
`scm_inf ()` [C Function]

Return Inf.

`numerator` *x* [Scheme Procedure]
`scm_numerator` (*x*) [C Function]

Return the numerator of the rational number *x*.

`denominator` *x* [Scheme Procedure]
`scm_denominator` (*x*) [C Function]

Return the denominator of the rational number *x*.

`int scm_is_real` (*SCM val*) [C Function]
`int scm_is_rational` (*SCM val*) [C Function]

Equivalent to `scm_is_true (scm_real_p (val))` and `scm_is_true (scm_rational_p (val))`, respectively.

`double scm_to_double` (*SCM val*) [C Function]

Returns the number closest to *val* that is representable as a `double`. Returns infinity for a *val* that is too large in magnitude. The argument *val* must be a real number.

`SCM scm_from_double` (*double val*) [C Function]

Return the `SCM` value that representats *val*. The returned value is inexact according to the predicate `inexact?`, but it will be exactly equal to *val*.

5.5.2.4 Complex Numbers

Complex numbers are the set of numbers that describe all possible points in a two-dimensional space. The two coordinates of a particular point in this space are known as the *real* and *imaginary* parts of the complex number that describes that point.

In Guile, complex numbers are written in rectangular form as the sum of their real and imaginary parts, using the symbol `i` to indicate the imaginary part.

```
3+4i
⇒
3.0+4.0i

(* 3-8i 2.3+0.3i)
⇒
9.3-17.5i
```

Polar form can also be used, with an '`@`' between magnitude and angle,

```
1@3.141592 ⇒ -1.0      (approx)
-1@1.57079 ⇒ 0.0-1.0i  (approx)
```

Guile represents a complex number with a non-zero imaginary part as a pair of inexact rationals, so the real and imaginary parts of a complex number have the same properties of inexactness and limited precision as single inexact rational numbers. Guile can not represent exact complex numbers with non-zero imaginary parts.

`complex?` *z* [Scheme Procedure]
`scm_complex_p` (*z*) [C Function]

Return `#t` if *x* is a complex number, `#f` otherwise. Note that the sets of real, rational and integer values form subsets of the set of complex numbers, i. e. the predicate will also be fulfilled if *x* is a real, rational or integer number.

`int scm_is_complex` (*SCM val*) [C Function]

Equivalent to `scm_is_true (scm_complex_p (val))`.

5.5.2.5 Exact and Inexact Numbers

R5RS requires that a calculation involving inexact numbers always produces an inexact result. To meet this requirement, Guile distinguishes between an exact integer value such as '`5`' and the corresponding inexact real value which, to the limited precision available, has no fractional part, and is printed as '`5.0`'. Guile will only convert the latter value to the former when forced to do so by an invocation of the `inexact->exact` procedure.

`exact?` *z* [Scheme Procedure]
`scm_exact_p` (*z*) [C Function]

Return `#t` if the number *z* is exact, `#f` otherwise.

```
(exact? 2)
⇒ #t

(exact? 0.5)
⇒ #f

(exact? (/ 2))
⇒ #t
```

`inexact?` *z* [Scheme Procedure]
`scm_inexact_p` (*z*) [C Function]

Return `#t` if the number *z* is inexact, `#f` else.

`inexact->exact` *z* [Scheme Procedure]
`scm_inexact_to_exact` (*z*) [C Function]

Return an exact number that is numerically closest to *z*, when there is one. For inexact rationals, Guile returns the exact rational that is numerically equal to the inexact rational. Inexact complex numbers with a non-zero imaginary part can not be made exact.

```
(inexact->exact 0.5)
⇒ 1/2
```

The following happens because 12/10 is not exactly representable as a `double` (on most platforms). However, when reading a decimal number that has been marked exact with the "#e" prefix, Guile is able to represent it correctly.

```
(inexact->exact 1.2)
⇒ 5404319552844595/4503599627370496

#e1.2
⇒ 6/5
```

`exact->inexact` *z* [Scheme Procedure]
`scm_exact_to_inexact` (*z*) [C Function]

Convert the number *z* to its inexact representation.

5.5.2.6 Read Syntax for Numerical Data

The read syntax for integers is a string of digits, optionally preceded by a minus or plus character, a code indicating the base in which the integer is encoded, and a code indicating whether the number is exact or inexact. The supported base codes are:

`#b` `#B`	the integer is written in binary (base 2)
`#o` `#O`	the integer is written in octal (base 8)
`#d` `#D`	the integer is written in decimal (base 10)
`#x` `#X`	the integer is written in hexadecimal (base 16)

If the base code is omitted, the integer is assumed to be decimal. The following examples show how these base codes are used.

```
-13
⇒ -13

#d-13
⇒ -13

#x-13
⇒ -19

#b+1101
⇒ 13

#o377
⇒ 255
```

The codes for indicating exactness (which can, incidentally, be applied to all numerical values) are:

`#e` `#E`	the number is exact
`#i` `#I`	the number is inexact.

If the exactness indicator is omitted, the number is exact unless it contains a radix point. Since Guile can not represent exact complex numbers, an error is signalled when asking for them.

```
(exact? 1.2)
```

```
⇒ #f

(exact? #e1.2)
⇒ #t

(exact? #e+1i)
ERROR: Wrong type argument
```

Guile also understands the syntax '`+inf.0`' and '`-inf.0`' for plus and minus infinity, respectively. The value must be written exactly as shown, that is, they always must have a sign and exactly one zero digit after the decimal point. It also understands '`+nan.0`' and '`-nan.0`' for the special 'not-a-number' value. The sign is ignored for 'not-a-number' and the value is always printed as '`+nan.0`'.

5.5.2.7 Operations on Integer Values

`odd?` *n* [Scheme Procedure]
`scm_odd_p` (*n*) [C Function]

Return `#t` if *n* is an odd number, `#f` otherwise.

`even?` *n* [Scheme Procedure]
`scm_even_p` (*n*) [C Function]

Return `#t` if *n* is an even number, `#f` otherwise.

`quotient` *n d* [Scheme Procedure]
`remainder` *n d* [Scheme Procedure]
`scm_quotient` (*n*, *d*) [C Function]
`scm_remainder` (*n*, *d*) [C Function]

Return the quotient or remainder from *n* divided by *d*. The quotient is rounded towards zero, and the remainder will have the same sign as *n*. In all cases quotient and remainder satisfy $n = q * d + r$.

```
(remainder 13 4) ⇒ 1
(remainder -13 4) ⇒ -1
```

`modulo` *n d* [Scheme Procedure]
`scm_modulo` (*n*, *d*) [C Function]

Return the remainder from *n* divided by *d*, with the same sign as *d*.

```
(modulo 13 4) ⇒ 1
(modulo -13 4) ⇒ 3
(modulo 13 -4) ⇒ -3
(modulo -13 -4) ⇒ -1
```

`gcd` *x...* [Scheme Procedure]
`scm_gcd` (*x*, *y*) [C Function]

Return the greatest common divisor of all arguments. If called without arguments, 0 is returned.

The C function `scm_gcd` always takes two arguments, while the Scheme function can take an arbitrary number.

`lcm` *x...* [Scheme Procedure]
`scm_lcm` (*x*, *y*) [C Function]

Return the least common multiple of the arguments. If called without arguments, 1 is returned.

The C function `scm_lcm` always takes two arguments, while the Scheme function can take an arbitrary number.

`modulo-expt` *n k m* [Scheme Procedure]
`scm_modulo_expt` (*n*, *k*, *m*) [C Function]

Return *n* raised to the integer exponent *k*, modulo *m*.

```
(modulo-expt 2 3 5)
   ⇒ 3
```

5.5.2.8 Comparison Predicates

The C comparison functions below always takes two arguments, while the Scheme functions can take an arbitrary number. Also keep in mind that the C functions return one of the Scheme boolean values `SCM_BOOL_T` or `SCM_BOOL_F` which are both true as far as C is concerned. Thus, always write `scm_is_true (scm_num_eq_p (x, y))` when testing the two Scheme numbers `x` and `y` for equality, for example.

`=` [Scheme Procedure]
`scm_num_eq_p` (*x*, *y*) [C Function]

Return `#t` if all parameters are numerically equal.

`<` [Scheme Procedure]
`scm_less_p` (*x*, *y*) [C Function]

Return `#t` if the list of parameters is monotonically increasing.

`>` [Scheme Procedure]
`scm_gr_p` (*x*, *y*) [C Function]

Return `#t` if the list of parameters is monotonically decreasing.

`<=` [Scheme Procedure]
`scm_leq_p` (*x*, *y*) [C Function]

Return `#t` if the list of parameters is monotonically non-decreasing.

`>=` [Scheme Procedure]
`scm_geq_p` (*x*, *y*) [C Function]

Return `#t` if the list of parameters is monotonically non-increasing.

`zero?` *z* [Scheme Procedure]
`scm_zero_p` (*z*) [C Function]

Return `#t` if *z* is an exact or inexact number equal to zero.

`positive?` *x* [Scheme Procedure]
`scm_positive_p` (*x*) [C Function]

Return `#t` if *x* is an exact or inexact number greater than zero.

`negative?` *x* [Scheme Procedure]
`scm_negative_p` (*x*) [C Function]

Return `#t` if *x* is an exact or inexact number less than zero.

5.5.2.9 Converting Numbers To and From Strings

`number->string` *n* [*radix*] [Scheme Procedure]
`scm_number_to_string` (*n, radix*) [C Function]

Return a string holding the external representation of the number *n* in the given *radix*. If *n* is inexact, a radix of 10 will be used.

`string->number` *string* [*radix*] [Scheme Procedure]
`scm_string_to_number` (*string, radix*) [C Function]

Return a number of the maximally precise representation expressed by the given *string*. *radix* must be an exact integer, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g. "#o177"). If *radix* is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number, then `string->number` returns `#f`.

`SCM scm_c_locale_stringn_to_number` (*const char *string, size_t len, unsigned radix*) [C Function]

As per `string->number` above, but taking a C string, as pointer and length. The string characters should be in the current locale encoding (`locale` in the name refers only to that, there's no locale-dependent parsing).

5.5.2.10 Complex Number Operations

`make-rectangular` *real imaginary* [Scheme Procedure]
`scm_make_rectangular` (*real, imaginary*) [C Function]

Return a complex number constructed of the given *real* and *imaginary* parts.

`make-polar` *x y* [Scheme Procedure]
`scm_make_polar` (*x, y*) [C Function]

Return the complex number *x* * e^(i * *y*).

`real-part` *z* [Scheme Procedure]
`scm_real_part` (*z*) [C Function]

Return the real part of the number *z*.

`imag-part` *z* [Scheme Procedure]
`scm_imag_part` (*z*) [C Function]

Return the imaginary part of the number *z*.

`magnitude` *z* [Scheme Procedure]
`scm_magnitude` (*z*) [C Function]

Return the magnitude of the number *z*. This is the same as `abs` for real arguments, but also allows complex numbers.

`angle` *z* [Scheme Procedure]
`scm_angle` (*z*) [C Function]

Return the angle of the complex number *z*.

`SCM scm_c_make_rectangular` (*double re, double im*) [C Function]
`SCM scm_c_make_polar` (*double x, double y*) [C Function]

Like `scm_make_rectangular` or `scm_make_polar`, respectively, but these functions take `double`s as their arguments.

`double scm_c_real_part` (*z*) [C Function]
`double scm_c_imag_part` (*z*) [C Function]

Returns the real or imaginary part of *z* as a `double`.

`double scm_c_magnitude` (*z*) [C Function]
`double scm_c_angle` (*z*) [C Function]

Returns the magnitude or angle of *z* as a `double`.

5.5.2.11 Arithmetic Functions

The C arithmetic functions below always takes two arguments, while the Scheme functions can take an arbitrary number. When you need to invoke them with just one argument, for example to compute the equivalent od `(- x)`, pass `SCM_UNDEFINED` as the second one: `scm_difference (x, SCM_UNDEFINED)`.

`+` *z1 . . .* [Scheme Procedure]
`scm_sum` (*z1*, *z2*) [C Function]

Return the sum of all parameter values. Return 0 if called without any parameters.

`-` *z1 z2 . . .* [Scheme Procedure]
`scm_difference` (*z1*, *z2*) [C Function]

If called with one argument *z1*, -*z1* is returned. Otherwise the sum of all but the first argument are subtracted from the first argument.

`*` *z1 . . .* [Scheme Procedure]
`scm_product` (*z1*, *z2*) [C Function]

Return the product of all arguments. If called without arguments, 1 is returned.

`/` *z1 z2 . . .* [Scheme Procedure]
`scm_divide` (*z1*, *z2*) [C Function]

Divide the first argument by the product of the remaining arguments. If called with one argument *z1*, 1/*z1* is returned.

`abs` *x* [Scheme Procedure]
`scm_abs` (*x*) [C Function]

Return the absolute value of *x*.

x must be a number with zero imaginary part. To calculate the magnitude of a complex number, use `magnitude` instead.

`max` *x1 x2 . . .* [Scheme Procedure]
`scm_max` (*x1*, *x2*) [C Function]

Return the maximum of all parameter values.

`min` *x1 x2 . . .* [Scheme Procedure]
`scm_min` (*x1*, *x2*) [C Function]

Return the minimum of all parameter values.

`truncate` *x* [Scheme Procedure]
`scm_truncate_number` (*x*) [C Function]

Round the inexact number *x* towards zero.

`round` *x* [Scheme Procedure]
`scm_round_number` (*x*) [C Function]

Round the inexact number *x* to the nearest integer. When exactly halfway between two integers, round to the even one.

`floor` *x* [Scheme Procedure]
`scm_floor` (*x*) [C Function]

Round the number *x* towards minus infinity.

`ceiling` *x* [Scheme Procedure]
`scm_ceiling` (*x*) [C Function]

Round the number *x* towards infinity.

`double scm_c_truncate` (*double x*) [C Function]
`double scm_c_round` (*double x*) [C Function]

Like `scm_truncate_number` or `scm_round_number`, respectively, but these functions take and return `double` values.

5.5.2.12 Scientific Functions

The following procedures accept any kind of number as arguments, including complex numbers.

`sqrt` *z* [Scheme Procedure]

Return the square root of *z*. Of the two possible roots (positive and negative), the one with the a positive real part is returned, or if that's zero then a positive imaginary part. Thus,

```
(sqrt 9.0)       ⇒ 3.0
(sqrt -9.0)      ⇒ 0.0+3.0i
(sqrt 1.0+1.0i)  ⇒ 1.09868411346781+0.455089860562227i
(sqrt -1.0-1.0i) ⇒ 0.455089860562227-1.09868411346781i
```

`expt` *z1 z2* [Scheme Procedure]

Return *z1* raised to the power of *z2*.

`sin` *z* [Scheme Procedure]

Return the sine of *z*.

`cos` *z* [Scheme Procedure]

Return the cosine of *z*.

`tan` *z* [Scheme Procedure]

Return the tangent of *z*.

`asin` *z* [Scheme Procedure]

Return the arcsine of *z*.

`acos` *z* [Scheme Procedure]

Return the arccosine of *z*.

`atan` *z* [Scheme Procedure]
`atan` *y x* [Scheme Procedure]
: Return the arctangent of z, or of y/x.

`exp` *z* [Scheme Procedure]
: Return e to the power of z, where e is the base of natural logarithms (2.71828...).

`log` *z* [Scheme Procedure]
: Return the natural logarithm of z.

`log10` *z* [Scheme Procedure]
: Return the base 10 logarithm of z.

`sinh` *z* [Scheme Procedure]
: Return the hyperbolic sine of z.

`cosh` *z* [Scheme Procedure]
: Return the hyperbolic cosine of z.

`tanh` *z* [Scheme Procedure]
: Return the hyperbolic tangent of z.

`asinh` *z* [Scheme Procedure]
: Return the hyperbolic arcsine of z.

`acosh` *z* [Scheme Procedure]
: Return the hyperbolic arccosine of z.

`atanh` *z* [Scheme Procedure]
: Return the hyperbolic arctangent of z.

5.5.2.13 Primitive Numeric Functions

Many of Guile's numeric procedures which accept any kind of numbers as arguments, including complex numbers, are implemented as Scheme procedures that use the following real number-based primitives. These primitives signal an error if they are called with complex arguments.

`$abs` *x* [Scheme Procedure]
: Return the absolute value of x.

`$sqrt` *x* [Scheme Procedure]
: Return the square root of x.

`$expt` *x y* [Scheme Procedure]
`scm_sys_expt` (*x*, *y*) [C Function]
: Return x raised to the power of y. This procedure does not accept complex arguments.

`$sin` *x* [Scheme Procedure]
: Return the sine of x.

`$cos` *x* [Scheme Procedure]
: Return the cosine of x.

`$tan` *x* [Scheme Procedure]
: Return the tangent of *x*.

`$asin` *x* [Scheme Procedure]
: Return the arcsine of *x*.

`$acos` *x* [Scheme Procedure]
: Return the arccosine of *x*.

`$atan` *x* [Scheme Procedure]
: Return the arctangent of *x* in the range $-PI/2$ to $PI/2$.

`$atan2` *x y* [Scheme Procedure]
`scm_sys_atan2` (*x*, *y*) [C Function]
: Return the arc tangent of the two arguments *x* and *y*. This is similar to calculating the arc tangent of *x* / *y*, except that the signs of both arguments are used to determine the quadrant of the result. This procedure does not accept complex arguments.

`$exp` *x* [Scheme Procedure]
: Return e to the power of *x*, where e is the base of natural logarithms (2.71828. . .).

`$log` *x* [Scheme Procedure]
: Return the natural logarithm of *x*.

`$sinh` *x* [Scheme Procedure]
: Return the hyperbolic sine of *x*.

`$cosh` *x* [Scheme Procedure]
: Return the hyperbolic cosine of *x*.

`$tanh` *x* [Scheme Procedure]
: Return the hyperbolic tangent of *x*.

`$asinh` *x* [Scheme Procedure]
: Return the hyperbolic arcsine of *x*.

`$acosh` *x* [Scheme Procedure]
: Return the hyperbolic arccosine of *x*.

`$atanh` *x* [Scheme Procedure]
: Return the hyperbolic arctangent of *x*.

C functions for the above are provided by the standard mathematics library. Naturally these expect and return `double` arguments (see section “Mathematics” in *GNU C Library Reference Manual*).

Scheme Procedure	C Function
`$abs`	`fabs`
`$sqrt`	`sqrt`
`$sin`	`sin`
`$cos`	`cos`

`$tan`	`tan`
`$asin`	`asin`
`$acos`	`acos`
`$atan`	`atan`
`$atan2`	`atan2`
`$exp`	`exp`
`$expt`	`pow`
`$log`	`log`
`$sinh`	`sinh`
`$cosh`	`cosh`
`$tanh`	`tanh`
`$asinh`	`asinh`
`$acosh`	`acosh`
`$atanh`	`atanh`

`asinh`, `acosh` and `atanh` are C99 standard but might not be available on older systems. Guile provides the following equivalents (on all systems).

`double scm_asinh` (*double x*) [C Function]
`double scm_acosh` (*double x*) [C Function]
`double scm_atanh` (*double x*) [C Function]

Return the hyperbolic arcsine, arccosine or arctangent of *x* respectively.

5.5.2.14 Bitwise Operations

For the following bitwise functions, negative numbers are treated as infinite precision twos-complements. For instance −6 is bits ...111010, with infinitely many ones on the left. It can be seen that adding 6 (binary 110) to such a bit pattern gives all zeros.

`logand` *n1 n2 ...* [Scheme Procedure]
`scm_logand` (*n1*, *n2*) [C Function]

Return the bitwise AND of the integer arguments.

```
(logand) ⇒ -1
(logand 7) ⇒ 7
(logand #b111 #b011 #b001) ⇒ 1
```

`logior` *n1 n2 ...* [Scheme Procedure]
`scm_logior` (*n1*, *n2*) [C Function]

Return the bitwise OR of the integer arguments.

```
(logior) ⇒ 0
(logior 7) ⇒ 7
(logior #b000 #b001 #b011) ⇒ 3
```

`logxor` *n1 n2 ...* [Scheme Procedure]
`scm_loxor` (*n1*, *n2*) [C Function]

Return the bitwise XOR of the integer arguments. A bit is set in the result if it is set in an odd number of arguments.

```
(logxor) ⇒ 0
(logxor 7) ⇒ 7
```

```
(logxor #b000 #b001 #b011) ⇒ 2
(logxor #b000 #b001 #b011 #b011) ⇒ 1
```

`lognot` *n* [Scheme Procedure]
`scm_lognot` (*n*) [C Function]

Return the integer which is the ones-complement of the integer argument, ie. each 0 bit is changed to 1 and each 1 bit to 0.

```
(number->string (lognot #b10000000) 2)
   ⇒ "-10000001"
(number->string (lognot #b0) 2)
   ⇒ "-1"
```

`logtest` *j k* [Scheme Procedure]
`scm_logtest` (*j*, *k*) [C Function]

Test whether *j* and *k* have any 1 bits in common. This is equivalent to `(not (zero? (logand j k)))`, but without actually calculating the `logand`, just testing for non-zero.

```
(logtest #b0100 #b1011) ⇒ #f
(logtest #b0100 #b0111) ⇒ #t
```

`logbit?` *index j* [Scheme Procedure]
`scm_logbit_p` (*index*, *j*) [C Function]

Test whether bit number *index* in *j* is set. *index* starts from 0 for the least significant bit.

```
(logbit? 0 #b1101) ⇒ #t
(logbit? 1 #b1101) ⇒ #f
(logbit? 2 #b1101) ⇒ #t
(logbit? 3 #b1101) ⇒ #t
(logbit? 4 #b1101) ⇒ #f
```

`ash` *n cnt* [Scheme Procedure]
`scm_ash` (*n*, *cnt*) [C Function]

Return *n* shifted left by *cnt* bits, or shifted right if *cnt* is negative. This is an “arithmetic” shift.

This is effectively a multiplication by 2^{cnt}, and when *cnt* is negative it's a division, rounded towards negative infinity. (Note that this is not the same rounding as `quotient` does.)

With *n* viewed as an infinite precision twos complement, `ash` means a left shift introducing zero bits, or a right shift dropping bits.

```
(number->string (ash #b1 3) 2)     ⇒ "1000"
(number->string (ash #b1010 -1) 2) ⇒ "101"

;; -23 is bits ...11101001, -6 is bits ...111010
(ash -23 -2) ⇒ -6
```

`logcount` *n* [Scheme Procedure]

`scm_logcount` (*n*) [C Function]

Return the number of bits in integer *n*. If *n* is positive, the 1-bits in its binary representation are counted. If negative, the 0-bits in its two's-complement binary representation are counted. If zero, 0 is returned.

```
(logcount #b10101010)
   ⇒ 4
(logcount 0)
   ⇒ 0
(logcount -2)
   ⇒ 1
```

`integer-length` *n* [Scheme Procedure]
`scm_integer_length` (*n*) [C Function]

Return the number of bits necessary to represent *n*.

For positive *n* this is how many bits to the most significant one bit. For negative *n* it's how many bits to the most significant zero bit in twos complement form.

```
(integer-length #b10101010) ⇒ 8
(integer-length #b1111)     ⇒ 4
(integer-length 0)          ⇒ 0
(integer-length -1)         ⇒ 0
(integer-length -256)       ⇒ 8
(integer-length -257)       ⇒ 9
```

`integer-expt` *n k* [Scheme Procedure]
`scm_integer_expt` (*n, k*) [C Function]

Return *n* raised to the power *k*. *k* must be an exact integer, *n* can be any number.

Negative *k* is supported, and results in $1/n^|k|$ in the usual way. n^0 is 1, as usual, and that includes 0^0 is 1.

```
(integer-expt 2 5)   ⇒ 32
(integer-expt -3 3)  ⇒ -27
(integer-expt 5 -3)  ⇒ 1/125
(integer-expt 0 0)   ⇒ 1
```

`bit-extract` *n start end* [Scheme Procedure]
`scm_bit_extract` (*n, start, end*) [C Function]

Return the integer composed of the *start* (inclusive) through *end* (exclusive) bits of *n*. The *start*th bit becomes the 0-th bit in the result.

```
(number->string (bit-extract #b1101101010 0 4) 2)
   ⇒ "1010"
(number->string (bit-extract #b1101101010 4 9) 2)
   ⇒ "10110"
```

5.5.2.15 Random Number Generation

Pseudo-random numbers are generated from a random state object, which can be created with `seed->random-state`. The *state* parameter to the various functions below is optional, it defaults to the state object in the `*random-state*` variable.

`copy-random-state` [*state*] [Scheme Procedure]
`scm_copy_random_state` (*state*) [C Function]
Return a copy of the random state *state*.

`random` *n* [*state*] [Scheme Procedure]
`scm_random` (*n, state*) [C Function]
Return a number in [0, *n*).

Accepts a positive integer or real n and returns a number of the same type between zero (inclusive) and *n* (exclusive). The values returned have a uniform distribution.

`random:exp` [*state*] [Scheme Procedure]
`scm_random_exp` (*state*) [C Function]
Return an inexact real in an exponential distribution with mean 1. For an exponential distribution with mean *u* use `(* u (random:exp))`.

`random:hollow-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_hollow_sphere_x` (*vect, state*) [C Function]
Fills *vect* with inexact real random numbers the sum of whose squares is equal to 1.0. Thinking of *vect* as coordinates in space of dimension *n* = `(vector-length vect)`, the coordinates are uniformly distributed over the surface of the unit n-sphere.

`random:normal` [*state*] [Scheme Procedure]
`scm_random_normal` (*state*) [C Function]
Return an inexact real in a normal distribution. The distribution used has mean 0 and standard deviation 1. For a normal distribution with mean *m* and standard deviation *d* use `(+ m (* d (random:normal)))`.

`random:normal-vector!` *vect* [*state*] [Scheme Procedure]
`scm_random_normal_vector_x` (*vect, state*) [C Function]
Fills *vect* with inexact real random numbers that are independent and standard normally distributed (i.e., with mean 0 and variance 1).

`random:solid-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_solid_sphere_x` (*vect, state*) [C Function]
Fills *vect* with inexact real random numbers the sum of whose squares is less than 1.0. Thinking of *vect* as coordinates in space of dimension *n* = `(vector-length vect)`, the coordinates are uniformly distributed within the unit *n*-sphere.

`random:uniform` [*state*] [Scheme Procedure]
`scm_random_uniform` (*state*) [C Function]
Return a uniformly distributed inexact real random number in [0,1).

`seed->random-state` *seed* [Scheme Procedure]
`scm_seed_to_random_state` (*seed*) [C Function]
Return a new random state using *seed*.

`*random-state*` [Variable]
The global random state used by the above functions when the *state* parameter is not given.

5.5.3 Characters

In Scheme, a character literal is written as `#\name` where *name* is the name of the character that you want. Printable characters have their usual single character name; for example, `#\a` is a lower case `a`.

Most of the "control characters" (those below codepoint 32) in the ASCII character set, as well as the space, may be referred to by longer names: for example, `#\tab`, `#\esc`, `#\stx`, and so on. The following table describes the ASCII names for each character.

0 = `#\nul`	1 = `#\soh`	2 = `#\stx`	3 = `#\etx`
4 = `#\eot`	5 = `#\enq`	6 = `#\ack`	7 = `#\bel`
8 = `#\bs`	9 = `#\ht`	10 = `#\nl`	11 = `#\vt`
12 = `#\np`	13 = `#\cr`	14 = `#\so`	15 = `#\si`
16 = `#\dle`	17 = `#\dc1`	18 = `#\dc2`	19 = `#\dc3`
20 = `#\dc4`	21 = `#\nak`	22 = `#\syn`	23 = `#\etb`
24 = `#\can`	25 = `#\em`	26 = `#\sub`	27 = `#\esc`
28 = `#\fs`	29 = `#\gs`	30 = `#\rs`	31 = `#\us`
32 = `#\sp`			

The "delete" character (octal 177) may be referred to with the name `#\del`.

Several characters have more than one name:

Alias	Original
`#\space`	`#\sp`
`#\newline`	`#\nl`
`#\tab`	`#\ht`
`#\backspace`	`#\bs`
`#\return`	`#\cr`
`#\page`	`#\np`
`#\null`	`#\nul`

`char?` *x* [Scheme Procedure]
`scm_char_p` (*x*) [C Function]

Return `#t` iff *x* is a character, else `#f`.

`char=?` *x y* [Scheme Procedure]

Return `#t` iff *x* is the same character as *y*, else `#f`.

`char<?` *x y* [Scheme Procedure]

Return `#t` iff *x* is less than *y* in the ASCII sequence, else `#f`.

`char<=?` *x y* [Scheme Procedure]

Return `#t` iff *x* is less than or equal to *y* in the ASCII sequence, else `#f`.

`char>?` *x y* [Scheme Procedure]

Return `#t` iff *x* is greater than *y* in the ASCII sequence, else `#f`.

`char>=?` *x y* [Scheme Procedure]

Return `#t` iff *x* is greater than or equal to *y* in the ASCII sequence, else `#f`.

`char-ci=?` *x y* [Scheme Procedure]

Return `#t` iff *x* is the same character as *y* ignoring case, else `#f`.

`char-ci<?` *x y* [Scheme Procedure]
Return `#t` iff *x* is less than *y* in the ASCII sequence ignoring case, else `#f`.

`char-ci<=?` *x y* [Scheme Procedure]
Return `#t` iff *x* is less than or equal to *y* in the ASCII sequence ignoring case, else `#f`.

`char-ci>?` *x y* [Scheme Procedure]
Return `#t` iff *x* is greater than *y* in the ASCII sequence ignoring case, else `#f`.

`char-ci>=?` *x y* [Scheme Procedure]
Return `#t` iff *x* is greater than or equal to *y* in the ASCII sequence ignoring case, else `#f`.

`char-alphabetic?` *chr* [Scheme Procedure]
`scm_char_alphabetic_p` (*chr*) [C Function]
Return `#t` iff *chr* is alphabetic, else `#f`.

`char-numeric?` *chr* [Scheme Procedure]
`scm_char_numeric_p` (*chr*) [C Function]
Return `#t` iff *chr* is numeric, else `#f`.

`char-whitespace?` *chr* [Scheme Procedure]
`scm_char_whitespace_p` (*chr*) [C Function]
Return `#t` iff *chr* is whitespace, else `#f`.

`char-upper-case?` *chr* [Scheme Procedure]
`scm_char_upper_case_p` (*chr*) [C Function]
Return `#t` iff *chr* is uppercase, else `#f`.

`char-lower-case?` *chr* [Scheme Procedure]
`scm_char_lower_case_p` (*chr*) [C Function]
Return `#t` iff *chr* is lowercase, else `#f`.

`char-is-both?` *chr* [Scheme Procedure]
`scm_char_is_both_p` (*chr*) [C Function]
Return `#t` iff *chr* is either uppercase or lowercase, else `#f`.

`char->integer` *chr* [Scheme Procedure]
`scm_char_to_integer` (*chr*) [C Function]
Return the number corresponding to ordinal position of *chr* in the ASCII sequence.

`integer->char` *n* [Scheme Procedure]
`scm_integer_to_char` (*n*) [C Function]
Return the character at position *n* in the ASCII sequence.

`char-upcase` *chr* [Scheme Procedure]
`scm_char_upcase` (*chr*) [C Function]
Return the uppercase character version of *chr*.

`char-downcase` *chr* [Scheme Procedure]
`scm_char_downcase` (*chr*) [C Function]
Return the lowercase character version of *chr*.

5.5.4 Character Sets

The features described in this section correspond directly to SRFI-14.

The data type *charset* implements sets of characters (see Section 5.5.3 [Characters], page 121). Because the internal representation of character sets is not visible to the user, a lot of procedures for handling them are provided.

Character sets can be created, extended, tested for the membership of a characters and be compared to other character sets.

The Guile implementation of character sets currently deals only with 8-bit characters. In the future, when Guile gets support for international character sets, this will change, but the functions provided here will always then be able to efficiently cope with very large character sets.

5.5.4.1 Character Set Predicates/Comparison

Use these procedures for testing whether an object is a character set, or whether several character sets are equal or subsets of each other. `char-set-hash` can be used for calculating a hash value, maybe for usage in fast lookup procedures.

`char-set?` *obj* [Scheme Procedure]
`scm_char_set_p` (*obj*) [C Function]

Return `#t` if *obj* is a character set, `#f` otherwise.

`char-set=` . *char_sets* [Scheme Procedure]
`scm_char_set_eq` (*char_sets*) [C Function]

Return `#t` if all given character sets are equal.

`char-set<=` . *char_sets* [Scheme Procedure]
`scm_char_set_leq` (*char_sets*) [C Function]

Return `#t` if every character set *cs*i is a subset of character set *cs*i+1.

`char-set-hash` *cs* [*bound*] [Scheme Procedure]
`scm_char_set_hash` (*cs, bound*) [C Function]

Compute a hash value for the character set *cs*. If *bound* is given and non-zero, it restricts the returned value to the range 0 . . . *bound - 1*.

5.5.4.2 Iterating Over Character Sets

Character set cursors are a means for iterating over the members of a character sets. After creating a character set cursor with `char-set-cursor`, a cursor can be dereferenced with `char-set-ref`, advanced to the next member with `char-set-cursor-next`. Whether a cursor has passed past the last element of the set can be checked with `end-of-char-set?`.

Additionally, mapping and (un-)folding procedures for character sets are provided.

`char-set-cursor` *cs* [Scheme Procedure]
`scm_char_set_cursor` (*cs*) [C Function]

Return a cursor into the character set *cs*.

`char-set-ref` *cs cursor* [Scheme Procedure]
`scm_char_set_ref` (*cs, cursor*) [C Function]

Return the character at the current cursor position *cursor* in the character set *cs*. It is an error to pass a cursor for which `end-of-char-set?` returns true.

`char-set-cursor-next` *cs cursor* [Scheme Procedure]
`scm_char_set_cursor_next` (*cs, cursor*) [C Function]

Advance the character set cursor *cursor* to the next character in the character set *cs*. It is an error if the cursor given satisfies `end-of-char-set?`.

`end-of-char-set?` *cursor* [Scheme Procedure]
`scm_end_of_char_set_p` (*cursor*) [C Function]

Return `#t` if *cursor* has reached the end of a character set, `#f` otherwise.

`char-set-fold` *kons knil cs* [Scheme Procedure]
`scm_char_set_fold` (*kons, knil, cs*) [C Function]

Fold the procedure *kons* over the character set *cs*, initializing it with *knil*.

`char-set-unfold` *p f g seed* [*base_cs*] [Scheme Procedure]
`scm_char_set_unfold` (*p, f, g, seed, base_cs*) [C Function]

This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of the seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base_cs* to form the result; *base_cs* defaults to the empty set.

`char-set-unfold!` *p f g seed base_cs* [Scheme Procedure]
`scm_char_set_unfold_x` (*p, f, g, seed, base_cs*) [C Function]

This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of the seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base_cs* to form the result; *base_cs* defaults to the empty set.

`char-set-for-each` *proc cs* [Scheme Procedure]
`scm_char_set_for_each` (*proc, cs*) [C Function]

Apply *proc* to every character in the character set *cs*. The return value is not specified.

`char-set-map` *proc cs* [Scheme Procedure]
`scm_char_set_map` (*proc, cs*) [C Function]

Map the procedure *proc* over every character in *cs*. *proc* must be a character `->` character procedure.

5.5.4.3 Creating Character Sets

New character sets are produced with these procedures.

`char-set-copy` *cs* [Scheme Procedure]
`scm_char_set_copy` (*cs*) [C Function]

Return a newly allocated character set containing all characters in *cs*.

`char-set` *. rest* [Scheme Procedure]
`scm_char_set` (*rest*) [C Function]

Return a character set containing all given characters.

`list->char-set` *list* [*base_cs*] [Scheme Procedure]
`scm_list_to_char_set` (*list, base_cs*) [C Function]

Convert the character list *list* to a character set. If the character set *base_cs* is given, the character in this set are also included in the result.

`list->char-set!` *list base_cs* [Scheme Procedure]
`scm_list_to_char_set_x` (*list, base_cs*) [C Function]

Convert the character list *list* to a character set. The characters are added to *base_cs* and *base_cs* is returned.

`string->char-set` *str* [*base_cs*] [Scheme Procedure]
`scm_string_to_char_set` (*str, base_cs*) [C Function]

Convert the string *str* to a character set. If the character set *base_cs* is given, the characters in this set are also included in the result.

`string->char-set!` *str base_cs* [Scheme Procedure]
`scm_string_to_char_set_x` (*str, base_cs*) [C Function]

Convert the string *str* to a character set. The characters from the string are added to *base_cs*, and *base_cs* is returned.

`char-set-filter` *pred cs* [*base_cs*] [Scheme Procedure]
`scm_char_set_filter` (*pred, cs, base_cs*) [C Function]

Return a character set containing every character from *cs* so that it satisfies *pred*. If provided, the characters from *base_cs* are added to the result.

`char-set-filter!` *pred cs base_cs* [Scheme Procedure]
`scm_char_set_filter_x` (*pred, cs, base_cs*) [C Function]

Return a character set containing every character from *cs* so that it satisfies *pred*. The characters are added to *base_cs* and *base_cs* is returned.

`ucs-range->char-set` *lower upper* [*error* [*base_cs*]] [Scheme Procedure]
`scm_ucs_range_to_char_set` (*lower, upper, error, base_cs*) [C Function]

Return a character set containing all characters whose character codes lie in the half-open range [*lower*,*upper*).

If *error* is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If *error* is `#f`, these characters are silently left out of the resultung character set.

The characters in *base_cs* are added to the result, if given.

`ucs-range->char-set!` *lower upper error base_cs* [Scheme Procedure]
`scm_ucs_range_to_char_set_x` (*lower, upper, error, base_cs*) [C Function]

Return a character set containing all characters whose character codes lie in the half-open range [*lower*,*upper*).

If *error* is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If *error* is `#f`, these characters are silently left out of the resultung character set.

The characters are added to *base_cs* and *base_cs* is returned.

`->char-set` *x* [Scheme Procedure]
`scm_to_char_set` (*x*) [C Function]
Coerces x into a char-set. *x* may be a string, character or char-set. A string is converted to the set of its constituent characters; a character is converted to a singleton set; a char-set is returned as-is.

5.5.4.4 Querying Character Sets

Access the elements and other information of a character set with these procedures.

`char-set-size` *cs* [Scheme Procedure]
`scm_char_set_size` (*cs*) [C Function]
Return the number of elements in character set *cs*.

`char-set-count` *pred cs* [Scheme Procedure]
`scm_char_set_count` (*pred, cs*) [C Function]
Return the number of the elements int the character set *cs* which satisfy the predicate *pred*.

`char-set->list` *cs* [Scheme Procedure]
`scm_char_set_to_list` (*cs*) [C Function]
Return a list containing the elements of the character set *cs*.

`char-set->string` *cs* [Scheme Procedure]
`scm_char_set_to_string` (*cs*) [C Function]
Return a string containing the elements of the character set *cs*. The order in which the characters are placed in the string is not defined.

`char-set-contains?` *cs ch* [Scheme Procedure]
`scm_char_set_contains_p` (*cs, ch*) [C Function]
Return `#t` iff the character *ch* is contained in the character set *cs*.

`char-set-every` *pred cs* [Scheme Procedure]
`scm_char_set_every` (*pred, cs*) [C Function]
Return a true value if every character in the character set *cs* satisfies the predicate *pred*.

`char-set-any` *pred cs* [Scheme Procedure]
`scm_char_set_any` (*pred, cs*) [C Function]
Return a true value if any character in the character set *cs* satisfies the predicate *pred*.

5.5.4.5 Character-Set Algebra

Character sets can be manipulated with the common set algebra operation, such as union, complement, intersection etc. All of these procedures provide side-effecting variants, which modify their character set argument(s).

`char-set-adjoin` *cs . rest* [Scheme Procedure]
`scm_char_set_adjoin` (*cs, rest*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete` *cs . rest* [Scheme Procedure]
`scm_char_set_delete` (*cs, rest*) [C Function]

Delete all character arguments from the first argument, which must be a character set.

`char-set-adjoin!` *cs . rest* [Scheme Procedure]
`scm_char_set_adjoin_x` (*cs, rest*) [C Function]

Add all character arguments to the first argument, which must be a character set.

`char-set-delete!` *cs . rest* [Scheme Procedure]
`scm_char_set_delete_x` (*cs, rest*) [C Function]

Delete all character arguments from the first argument, which must be a character set.

`char-set-complement` *cs* [Scheme Procedure]
`scm_char_set_complement` (*cs*) [C Function]

Return the complement of the character set *cs*.

`char-set-union` *. rest* [Scheme Procedure]
`scm_char_set_union` (*rest*) [C Function]

Return the union of all argument character sets.

`char-set-intersection` *. rest* [Scheme Procedure]
`scm_char_set_intersection` (*rest*) [C Function]

Return the intersection of all argument character sets.

`char-set-difference` *cs1 . rest* [Scheme Procedure]
`scm_char_set_difference` (*cs1, rest*) [C Function]

Return the difference of all argument character sets.

`char-set-xor` *. rest* [Scheme Procedure]
`scm_char_set_xor` (*rest*) [C Function]

Return the exclusive-or of all argument character sets.

`char-set-diff+intersection` *cs1 . rest* [Scheme Procedure]
`scm_char_set_diff_plus_intersection` (*cs1, rest*) [C Function]

Return the difference and the intersection of all argument character sets.

`char-set-complement!` *cs* [Scheme Procedure]
`scm_char_set_complement_x` (*cs*) [C Function]

Return the complement of the character set *cs*.

`char-set-union!` *cs1 . rest* [Scheme Procedure]
`scm_char_set_union_x` (*cs1, rest*) [C Function]

Return the union of all argument character sets.

`char-set-intersection!` *cs1 . rest* [Scheme Procedure]
`scm_char_set_intersection_x` (*cs1, rest*) [C Function]

Return the intersection of all argument character sets.

`char-set-difference!` *cs1 . rest* [Scheme Procedure]
`scm_char_set_difference_x` (*cs1*, *rest*) [C Function]
Return the difference of all argument character sets.

`char-set-xor!` *cs1 . rest* [Scheme Procedure]
`scm_char_set_xor_x` (*cs1*, *rest*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection!` *cs1 cs2 . rest* [Scheme Procedure]
`scm_char_set_diff_plus_intersection_x` (*cs1*, *cs2*, *rest*) [C Function]
Return the difference and the intersection of all argument character sets.

5.5.4.6 Standard Character Sets

In order to make the use of the character set data type and procedures useful, several predefined character set variables exist.

Currently, the contents of these character sets are recomputed upon a successful `setlocale` call (see Section 6.2.13 [Locales], page 417) in order to reflect the characters available in the current locale's codeset. For instance, `char-set:letter` contains 52 characters under an ASCII locale (e.g., the default `C` locale) and 117 characters under an ISO-8859-1 ("Latin-1") locale.

`char-set:lower-case` [Scheme Variable]
`scm_char_set_lower_case` [C Variable]
All lower-case characters.

`char-set:upper-case` [Scheme Variable]
`scm_char_set_upper_case` [C Variable]
All upper-case characters.

`char-set:title-case` [Scheme Variable]
`scm_char_set_title_case` [C Variable]
This is empty, because ASCII has no titlecase characters.

`char-set:letter` [Scheme Variable]
`scm_char_set_letter` [C Variable]
All letters, e.g. the union of `char-set:lower-case` and `char-set:upper-case`.

`char-set:digit` [Scheme Variable]
`scm_char_set_digit` [C Variable]
All digits.

`char-set:letter+digit` [Scheme Variable]
`scm_char_set_letter_and_digit` [C Variable]
The union of `char-set:letter` and `char-set:digit`.

`char-set:graphic` [Scheme Variable]
`scm_char_set_graphic` [C Variable]
All characters which would put ink on the paper.

`char-set:printing` [Scheme Variable]
`scm_char_set_printing` [C Variable]

The union of `char-set:graphic` and `char-set:whitespace`.

`char-set:whitespace` [Scheme Variable]
`scm_char_set_whitespace` [C Variable]

All whitespace characters.

`char-set:blank` [Scheme Variable]
`scm_char_set_blank` [C Variable]

All horizontal whitespace characters, that is `#\space` and `#\tab`.

`char-set:iso-control` [Scheme Variable]
`scm_char_set_iso_control` [C Variable]

The ISO control characters with the codes 0–31 and 127.

`char-set:punctuation` [Scheme Variable]
`scm_char_set_punctuation` [C Variable]

The characters `!"#%&'()*,-./:;?@[\\]_{}`

`char-set:symbol` [Scheme Variable]
`scm_char_set_symbol` [C Variable]

The characters `$+<=>^`|~`.

`char-set:hex-digit` [Scheme Variable]
`scm_char_set_hex_digit` [C Variable]

The hexadecimal digits `0123456789abcdefABCDEF`.

`char-set:ascii` [Scheme Variable]
`scm_char_set_ascii` [C Variable]

All ASCII characters.

`char-set:empty` [Scheme Variable]
`scm_char_set_empty` [C Variable]

The empty character set.

`char-set:full` [Scheme Variable]
`scm_char_set_full` [C Variable]

This character set contains all possible characters.

5.5.5 Strings

Strings are fixed-length sequences of characters. They can be created by calling constructor procedures, but they can also literally get entered at the REPL or in Scheme source files.

Strings always carry the information about how many characters they are composed of with them, so there is no special end-of-string character, like in C. That means that Scheme strings can contain any character, even the ‘`#\nul`’ character ‘`\0`’.

To use strings efficiently, you need to know a bit about how Guile implements them. In Guile, a string consists of two parts, a head and the actual memory where the characters are stored. When a string (or a substring of it) is copied, only a new head gets created, the memory is usually not copied. The two heads start out pointing to the same memory.

When one of these two strings is modified, as with `string-set!`, their common memory does get copied so that each string has its own memory and modifying one does not accidently modify the other as well. Thus, Guile's strings are 'copy on write'; the actual copying of their memory is delayed until one string is written to.

This implementation makes functions like `substring` very efficient in the common case that no modifications are done to the involved strings.

If you do know that your strings are getting modified right away, you can use `substring/copy` instead of `substring`. This function performs the copy immediately at the time of creation. This is more efficient, especially in a multi-threaded program. Also, `substring/copy` can avoid the problem that a short substring holds on to the memory of a very large original string that could otherwise be recycled.

If you want to avoid the copy altogether, so that modifications of one string show up in the other, you can use `substring/shared`. The strings created by this procedure are called *mutation sharing substrings* since the substring and the original string share modifications to each other.

If you want to prevent modifications, use `substring/read-only`.

Guile provides all procedures of SRFI-13 and a few more.

5.5.5.1 String Read Syntax

The read syntax for strings is an arbitrarily long sequence of characters enclosed in double quotes (`"`).

Backslash is an escape character and can be used to insert the following special characters. `\"` and `\\` are R5RS standard, the rest are Guile extensions, notice they follow C string syntax.

`\\`	Backslash character.
`\"`	Double quote character (an unescaped `"` is otherwise the end of the string).
`\0`	NUL character (ASCII 0).
`\a`	Bell character (ASCII 7).
`\f`	Formfeed character (ASCII 12).
`\n`	Newline character (ASCII 10).
`\r`	Carriage return character (ASCII 13).
`\t`	Tab character (ASCII 9).
`\v`	Vertical tab character (ASCII 11).
`\xHH`	Character code given by two hexadecimal digits. For example `\x7f` for an ASCII DEL (127).

The following are examples of string literals:

```
"foo"
"bar plonk"
"Hello World"
"\"Hi\", he said."
```

5.5.5.2 String Predicates

The following procedures can be used to check whether a given string fulfills some specified property.

`string?` *obj* [Scheme Procedure]
`scm_string_p` (*obj*) [C Function]

Return `#t` if *obj* is a string, else `#f`.

`int scm_is_string` (*SCM obj*) [C Function]

Returns `1` if *obj* is a string, `0` otherwise.

`string-null?` *str* [Scheme Procedure]
`scm_string_null_p` (*str*) [C Function]

Return `#t` if *str*'s length is zero, and `#f` otherwise.

```
(string-null? "")  ⇒ #t
y                   ⇒ "foo"
(string-null? y)    ⇒ #f
```

`string-any` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_any` (*char_pred, s, start, end*) [C Function]

Check if *char_pred* is true for any character in string *s*.

char_pred can be a character to check for any equal to that, or a character set (see Section 5.5.4 [Character Sets], page 123) to check for any in that set, or a predicate procedure to call.

For a procedure, calls `(char_pred c)` are made successively on the characters from *start* to *end*. If *char_pred* returns true (ie. non-`#f`), `string-any` stops and that return value is the return from `string-any`. The call on the last character (ie. at *end* − 1), if that point is reached, is a tail call.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#f`.

`string-every` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_every` (*char_pred, s, start, end*) [C Function]

Check if *char_pred* is true for every character in string *s*.

char_pred can be a character to check for every character equal to that, or a character set (see Section 5.5.4 [Character Sets], page 123) to check for every character being in that set, or a predicate procedure to call.

For a procedure, calls `(char_pred c)` are made successively on the characters from *start* to *end*. If *char_pred* returns `#f`, `string-every` stops and returns `#f`. The call on the last character (ie. at *end* − 1), if that point is reached, is a tail call and the return from that call is the return from `string-every`.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#t`.

5.5.5.3 String Constructors

The string constructor procedures create new string objects, possibly initializing them with some specified character data. See also See Section 5.5.5.5 [String Selection], page 133, for ways to create strings from existing strings.

`string` *char…* [Scheme Procedure]

Return a newly allocated string made from the given character arguments.

```
(string #\x #\y #\z) ⇒ "xyz"
(string)             ⇒ ""
```

`list->string` *lst* [Scheme Procedure]
`scm_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters.

```
(list->string '(#\a #\b #\c)) ⇒ "abc"
```

`reverse-list->string` *lst* [Scheme Procedure]
`scm_reverse_list_to_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters, in reverse order.

```
(reverse-list->string '(#\a #\B #\c)) ⇒ "cBa"
```

`make-string` *k* [*chr*] [Scheme Procedure]
`scm_make_string` (*k, chr*) [C Function]

Return a newly allocated string of length *k*. If *chr* is given, then all elements of the string are initialized to *chr*, otherwise the contents of the *string* are unspecified.

`SCM scm_c_make_string` (*size_t len, SCM chr*) [C Function]

Like `scm_make_string`, but expects the length as a `size_t`.

`string-tabulate` *proc len* [Scheme Procedure]
`scm_string_tabulate` (*proc, len*) [C Function]

proc is an integer->char procedure. Construct a string of size *len* by applying *proc* to each index to produce the corresponding string element. The order in which *proc* is applied to the indices is not specified.

`string-join` *ls* [*delimiter* [*grammar*]] [Scheme Procedure]
`scm_string_join` (*ls, delimiter, grammar*) [C Function]

Append the string in the string list *ls*, using the string *delim* as a delimiter between the elements of *ls*. *grammar* is a symbol which specifies how the delimiter is placed between the strings, and defaults to the symbol `infix`.

`infix` Insert the separator between list elements. An empty string will produce an empty list.

`string-infix`
Like `infix`, but will raise an error if given the empty list.

`suffix` Insert the separator after every list element.

`prefix` Insert the separator before each list element.

5.5.5.4 List/String conversion

When processing strings, it is often convenient to first convert them into a list representation by using the procedure `string->list`, work with the resulting list, and then convert it back into a string. These procedures are useful for similar tasks.

`string->list` *str [start [end]]* [Scheme Procedure]
`scm_substring_to_list` (*str, start, end*) [C Function]
`scm_string_to_list` (*str*) [C Function]

Convert the string *str* into a list of characters.

`string-split` *str chr* [Scheme Procedure]
`scm_string_split` (*str, chr*) [C Function]

Split the string *str* into the a list of the substrings delimited by appearances of the character *chr*. Note that an empty substring between separator characters will result in an empty string in the result list.

```
(string-split "root:x:0:0:root:/root:/bin/bash" #\:)
⇒
("root" "x" "0" "0" "root" "/root" "/bin/bash")

(string-split "::" #\:)
⇒
("" "" "")

(string-split "" #\:)
⇒
("")
```

5.5.5.5 String Selection

Portions of strings can be extracted by these procedures. `string-ref` delivers individual characters whereas `substring` can be used to extract substrings from longer strings.

`string-length` *string* [Scheme Procedure]
`scm_string_length` (*string*) [C Function]

Return the number of characters in *string*.

`size_t scm_c_string_length` (*SCM str*) [C Function]

Return the number of characters in *str* as a `size_t`.

`string-ref` *str k* [Scheme Procedure]
`scm_string_ref` (*str, k*) [C Function]

Return character *k* of *str* using zero-origin indexing. *k* must be a valid index of *str*.

`SCM scm_c_string_ref` (*SCM str, size_t k*) [C Function]

Return character *k* of *str* using zero-origin indexing. *k* must be a valid index of *str*.

`string-copy` *str [start [end]]* [Scheme Procedure]
`scm_substring_copy` (*str, start, end*) [C Function]
`scm_string_copy` (*str*) [C Function]

Return a copy of the given string *str*.

The returned string shares storage with *str* initially, but it is copied as soon as one of the two strings is modified.

`substring` *str start [end]* [Scheme Procedure] `scm_substring` (*str, start, end*) [C Function]

Return a new string formed from the characters of *str* beginning with index *start* (inclusive) and ending with index *end* (exclusive). *str* must be a string, *start* and *end* must be exact integers satisfying:

0 <= *start* <= *end* <= `(string-length` *str*`)`.

The returned string shares storage with *str* initially, but it is copied as soon as one of the two strings is modified.

`substring/shared` *str start* [*end*] [Scheme Procedure]
`scm_substring_shared` (*str, start, end*) [C Function]

Like `substring`, but the strings continue to share their storage even if they are modified. Thus, modifications to *str* show up in the new string, and vice versa.

`substring/copy` *str start* [*end*] [Scheme Procedure]
`scm_substring_copy` (*str, start, end*) [C Function]

Like `substring`, but the storage for the new string is copied immediately.

`substring/read-only` *str start* [*end*] [Scheme Procedure]
`scm_substring_read_only` (*str, start, end*) [C Function]

Like `substring`, but the resulting string can not be modified.

`SCM scm_c_substring` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_shared` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_copy` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_read_only` (*SCM str, size_t start, size_t end*) [C Function]

Like `scm_substring`, etc. but the bounds are given as a `size_t`.

`string-take` *s n* [Scheme Procedure]
`scm_string_take` (*s, n*) [C Function]

Return the *n* first characters of *s*.

`string-drop` *s n* [Scheme Procedure]
`scm_string_drop` (*s, n*) [C Function]

Return all but the first *n* characters of *s*.

`string-take-right` *s n* [Scheme Procedure]
`scm_string_take_right` (*s, n*) [C Function]

Return the *n* last characters of *s*.

`string-drop-right` *s n* [Scheme Procedure]
`scm_string_drop_right` (*s, n*) [C Function]

Return all but the last *n* characters of *s*.

`string-pad` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`string-pad-right` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`scm_string_pad` (*s, len, chr, start, end*) [C Function]
`scm_string_pad_right` (*s, len, chr, start, end*) [C Function]

Take characters *start* to *end* from the string *s* and either pad with *char* or truncate them to give *len* characters.

`string-pad` pads or truncates on the left, so for example

```
(string-pad "x" 3)     ⇒ "  x"
(string-pad "abcde" 3) ⇒ "cde"
```

`string-pad-right` pads or truncates on the right, so for example

```
(string-pad-right "x" 3)     ⇒ "x  "
(string-pad-right "abcde" 3) ⇒ "abc"
```

`string-trim` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-right` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-both` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`scm_string_trim` (*s, char_pred, start, end*) [C Function]
`scm_string_trim_right` (*s, char_pred, start, end*) [C Function]
`scm_string_trim_both` (*s, char_pred, start, end*) [C Function]

Trim occurrances of *char_pred* from the ends of *s*.

`string-trim` trims *char_pred* characters from the left (start) of the string, `string-trim-right` trims them from the right (end) of the string, `string-trim-both` trims from both ends.

char_pred can be a character, a character set, or a predicate procedure to call on each character. If *char_pred* is not given the default is whitespace as per `char-set:whitespace` (see Section 5.5.4.6 [Standard Character Sets], page 128).

```
(string-trim " x ")              ⇒ "x "
(string-trim-right "banana" #\a) ⇒ "banan"
(string-trim-both ".,xy:;" char-set:punctuation)
                  ⇒ "xy"
(string-trim-both "xyzzy" (lambda (c)
                            (or (eqv? c #\x)
                                (eqv? c #\y))))
                  ⇒ "zz"
```

5.5.5.6 String Modification

These procedures are for modifying strings in-place. This means that the result of the operation is not a new string; instead, the original string's memory representation is modified.

`string-set!` *str k chr* [Scheme Procedure]
`scm_string_set_x` (*str, k, chr*) [C Function]

Store *chr* in element *k* of *str* and return an unspecified value. *k* must be a valid index of *str*.

`void scm_c_string_set_x` (*SCM str, size_t k, SCM chr*) [C Function]

Like `scm_string_set_x`, but the index is given as a `size_t`.

`string-fill!` *str chr* [*start* [*end*]] [Scheme Procedure]
`scm_substring_fill_x` (*str, chr, start, end*) [C Function]
`scm_string_fill_x` (*str, chr*) [C Function]

Stores *chr* in every element of the given *str* and returns an unspecified value.

`substring-fill!` *str start end fill* [Scheme Procedure]
`scm_substring_fill_x` (*str, start, end, fill*) [C Function]

Change every character in *str* between *start* and *end* to *fill*.

```
(define y "abcdefg")
(substring-fill! y 1 3 #\r)
y
⇒ "arrdefg"
```

`substring-move!` *str1 start1 end1 str2 start2* [Scheme Procedure]
`scm_substring_move_x` (*str1, start1, end1, str2, start2*) [C Function]

Copy the substring of *str1* bounded by *start1* and *end1* into *str2* beginning at position *start2*. *str1* and *str2* can be the same string.

`string-copy!` *target tstart s* [*start* [*end*]] [Scheme Procedure]
`scm_string_copy_x` (*target, tstart, s, start, end*) [C Function]

Copy the sequence of characters from index range [*start*, *end*) in string *s* to string *target*, beginning at index *tstart*. The characters are copied left-to-right or right-to-left as needed – the copy is guaranteed to work, even if *target* and *s* are the same string. It is an error if the copy operation runs off the end of the target string.

5.5.5.7 String Comparison

The procedures in this section are similar to the character ordering predicates (see Section 5.5.3 [Characters], page 121), but are defined on character sequences.

The first set is specified in R5RS and has names that end in `?`. The second set is specified in SRFI-13 and the names have no ending `?`. The predicates ending in `-ci` ignore the character case when comparing strings.

`string=?` *s1 s2* [Scheme Procedure]

Lexicographic equality predicate; return `#t` if the two strings are the same length and contain the same characters in the same positions, otherwise return `#f`.

The procedure `string-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

`string<?` *s1 s2* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if *s1* is lexicographically less than *s2*.

`string<=?` *s1 s2* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if *s1* is lexicographically less than or equal to *s2*.

`string>?` *s1 s2* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if *s1* is lexicographically greater than *s2*.

`string>=?` *s1 s2* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if *s1* is lexicographically greater than or equal to *s2*.

`string-ci=?` *s1 s2* [Scheme Procedure]

Case-insensitive string equality predicate; return `#t` if the two strings are the same length and their component characters match (ignoring case) at each position; otherwise return `#f`.

`string-ci<?` *s1 s2* [Scheme Procedure]
: Case insensitive lexicographic ordering predicate; return `#t` if *s1* is lexicographically less than *s2* regardless of case.

`string-ci<=?` *s1 s2* [Scheme Procedure]
: Case insensitive lexicographic ordering predicate; return `#t` if *s1* is lexicographically less than or equal to *s2* regardless of case.

`string-ci>?` *s1 s2* [Scheme Procedure]
: Case insensitive lexicographic ordering predicate; return `#t` if *s1* is lexicographically greater than *s2* regardless of case.

`string-ci>=?` *s1 s2* [Scheme Procedure]
: Case insensitive lexicographic ordering predicate; return `#t` if *s1* is lexicographically greater than or equal to *s2* regardless of case.

`string-compare` *s1 s2 proc_lt proc_eq proc_gt [start1 [end1 [start2 [end2]]]]* [Scheme Procedure]
`scm_string_compare` (*s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2*) [C Function]
: Apply *proc_lt*, *proc_eq*, *proc_gt* to the mismatch index, depending upon whether *s1* is less than, equal to, or greater than *s2*. The mismatch index is the largest index *i* such that for every 0 <= *j* < *i*, *s1*[*j*] = *s2*[*j*] – that is, *i* is the first position that does not match.

`string-compare-ci` *s1 s2 proc_lt proc_eq proc_gt [start1 [end1 [start2 [end2]]]]* [Scheme Procedure]
`scm_string_compare_ci` (*s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2*) [C Function]
: Apply *proc_lt*, *proc_eq*, *proc_gt* to the mismatch index, depending upon whether *s1* is less than, equal to, or greater than *s2*. The mismatch index is the largest index *i* such that for every 0 <= *j* < *i*, *s1*[*j*] = *s2*[*j*] – that is, *i* is the first position that does not match. The character comparison is done case-insensitively.

`string=` *s1 s2 [start1 [end1 [start2 [end2]]]]* [Scheme Procedure]
`scm_string_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]
: Return `#f` if *s1* and *s2* are not equal, a true value otherwise.

`string<>` *s1 s2 [start1 [end1 [start2 [end2]]]]* [Scheme Procedure]
`scm_string_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]
: Return `#f` if *s1* and *s2* are equal, a true value otherwise.

`string<` *s1 s2 [start1 [end1 [start2 [end2]]]]* [Scheme Procedure]
`scm_string_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]
: Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise.

`string>` *s1 s2 [start1 [end1 [start2 [end2]]]]* [Scheme Procedure]
`scm_string_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]
: Return `#f` if *s1* is less or equal to *s2*, a true value otherwise.

`string<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_le` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater to *s2*, a true value otherwise.

`string>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less to *s2*, a true value otherwise.

`string-ci=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are not equal, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are equal, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_le` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-hash` *s* [*bound* [*start* [*end*]]] [Scheme Procedure]
`scm_substring_hash` (*s, bound, start, end*) [C Function]

Compute a hash value for *S*. the optional argument *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

`string-hash-ci` *s* [*bound* [*start* [*end*]]] [Scheme Procedure]
`scm_substring_hash_ci` (*s, bound, start, end*) [C Function]

Compute a hash value for *S*. the optional argument *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

5.5.5.8 String Searching

`string-index` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index` (*s, char_pred, start, end*) [C Function]

Search through the string *s* from left to right, returning the index of the first occurence of a character which

- equals *char_pred*, if it is character,
- satisifies the predicate *char_pred*, if it is a procedure,
- is in the set *char_pred*, if it is a character set.

`string-rindex` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_rindex` (*s, char_pred, start, end*) [C Function]

Search through the string *s* from right to left, returning the index of the last occurence of a character which

- equals *char_pred*, if it is character,
- satisifies the predicate *char_pred*, if it is a procedure,
- is in the set if *char_pred* is a character set.

`string-prefix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the length of the longest common prefix of the two strings.

`string-prefix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the length of the longest common prefix of the two strings, ignoring character case.

`string-suffix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the length of the longest common suffix of the two strings.

`string-suffix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the length of the longest common suffix of the two strings, ignoring character case.

`string-prefix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a prefix of *s2*?

`string-prefix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a prefix of *s2*, ignoring character case?

`string-suffix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a suffix of *s2*?

`string-suffix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a suffix of *s2*, ignoring character case?

`string-index-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index_right` (*s, char_pred, start, end*) [C Function]

Search through the string *s* from right to left, returning the index of the last occurence of a character which

- equals *char_pred*, if it is character,
- satisifies the predicate *char_pred*, if it is a procedure,
- is in the set if *char_pred* is a character set.

`string-skip` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip` (*s, char_pred, start, end*) [C Function]

Search through the string *s* from left to right, returning the index of the first occurence of a character which

- does not equal *char_pred*, if it is character,
- does not satisify the predicate *char_pred*, if it is a procedure,
- is not in the set if *char_pred* is a character set.

`string-skip-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip_right` (*s, char_pred, start, end*) [C Function]

Search through the string *s* from right to left, returning the index of the last occurence of a character which

- does not equal *char_pred*, if it is character,
- does not satisfy the predicate *char_pred*, if it is a procedure,
- is not in the set if *char_pred* is a character set.

`string-count` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_count` (*s, char_pred, start, end*) [C Function]

Return the count of the number of characters in the string *s* which

- equals *char_pred*, if it is character,
- satisifies the predicate *char_pred*, if it is a procedure.
- is in the set *char_pred*, if it is a character set.

`string-contains` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_contains` (*s1, s2, start1, end1, start2, end2*) [C Function]

Does string *s1* contain string *s2*? Return the index in *s1* where *s2* occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings.

`string-contains-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_contains_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]

Does string *s1* contain string *s2*? Return the index in *s1* where *s2* occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings. Character comparison is done case-insensitively.

5.5.5.9 Alphabetic Case Mapping

These are procedures for mapping strings to their upper- or lower-case equivalents, respectively, or for capitalizing strings.

`string-upcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase` (*str, start, end*) [C Function]
`scm_string_upcase` (*str*) [C Function]

Upcase every character in `str`.

`string-upcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase_x` (*str, start, end*) [C Function]
`scm_string_upcase_x` (*str*) [C Function]

Destructively upcase every character in `str`.

```
(string-upcase! y)
⇒ "ARRDEFG"
y
⇒ "ARRDEFG"
```

`string-downcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase` (*str, start, end*) [C Function]
`scm_string_downcase` (*str*) [C Function]

Downcase every character in *str*.

`string-downcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase_x` (*str, start, end*) [C Function]
`scm_string_downcase_x` (*str*) [C Function]

Destructively downcase every character in *str*.

```
y
⇒ "ARRDEFG"
(string-downcase! y)
⇒ "arrdefg"
y
⇒ "arrdefg"
```

`string-capitalize` *str* [Scheme Procedure]
`scm_string_capitalize` (*str*) [C Function]

Return a freshly allocated string with the characters in *str*, where the first character of every word is capitalized.

`string-capitalize!` *str* [Scheme Procedure]
`scm_string_capitalize_x` (*str*) [C Function]

Upcase the first character of every word in *str* destructively and return *str*.

```
y                       ⇒ "hello world"
(string-capitalize! y)  ⇒ "Hello World"
y                       ⇒ "Hello World"
```

`string-titlecase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_titlecase` (*str, start, end*) [C Function]

Titlecase every first character in a word in *str*.

`string-titlecase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_titlecase_x` (*str*, *start*, *end*) [C Function]

Destructively titlecase every first character in a word in *str*.

5.5.5.10 Reversing and Appending Strings

`string-reverse` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_reverse` (*str*, *start*, *end*) [C Function]

Reverse the string *str*. The optional arguments *start* and *end* delimit the region of *str* to operate on.

`string-reverse!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_reverse_x` (*str*, *start*, *end*) [C Function]

Reverse the string *str* in-place. The optional arguments *start* and *end* delimit the region of *str* to operate on. The return value is unspecified.

`string-append` . *args* [Scheme Procedure]
`scm_string_append` (*args*) [C Function]

Return a newly allocated string whose characters form the concatenation of the given strings, *args*.

```
(let ((h "hello "))
  (string-append h "world"))
⇒ "hello world"
```

`string-append/shared` . *ls* [Scheme Procedure]
`scm_string_append_shared` (*ls*) [C Function]

Like `string-append`, but the result may share memory with the argument strings.

`string-concatenate` *ls* [Scheme Procedure]
`scm_string_concatenate` (*ls*) [C Function]

Append the elements of *ls* (which must be strings) together into a single string. Guaranteed to return a freshly allocated string.

`string-concatenate-reverse` *ls* [*final_string* [*end*]] [Scheme Procedure]
`scm_string_concatenate_reverse` (*ls*, *final_string*, *end*) [C Function]

Without optional arguments, this procedure is equivalent to

```
(string-concatenate (reverse ls))
```

If the optional argument *final_string* is specified, it is consed onto the beginning to *ls* before performing the list-reverse and string-concatenate operations. If *end* is given, only the characters of *final_string* up to index *end* are used.

Guaranteed to return a freshly allocated string.

`string-concatenate/shared` *ls* [Scheme Procedure]
`scm_string_concatenate_shared` (*ls*) [C Function]

Like `string-concatenate`, but the result may share memory with the strings in the list *ls*.

`string-concatenate-reverse/shared` *ls* [*final_string* [*end*]] [Scheme Procedure]
`scm_string_concatenate_reverse_shared` (*ls*, *final_string*, *end*) [C Function]

Like `string-concatenate-reverse`, but the result may share memory with the the strings in the *ls* arguments.

5.5.5.11 Mapping, Folding, and Unfolding

`string-map` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_map` (*proc, s, start, end*) [C Function]

proc is a char->char procedure, it is mapped over *s*. The order in which the procedure is applied to the string elements is not specified.

`string-map!` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_map_x` (*proc, s, start, end*) [C Function]

proc is a char->char procedure, it is mapped over *s*. The order in which the procedure is applied to the string elements is not specified. The string *s* is modified in-place, the return value is not specified.

`string-for-each` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_for_each` (*proc, s, start, end*) [C Function]

proc is mapped over *s* in left-to-right order. The return value is not specified.

`string-for-each-index` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_for_each_index` (*proc, s, start, end*) [C Function]

Call `(proc i)` for each index i in *s*, from left to right.

For example, to change characters to alternately upper and lower case,

```
(define str (string-copy "studly"))
(string-for-each-index (lambda (i)
                         (string-set! str i
                           ((if (even? i) char-upcase char-downcase)
                            (string-ref str i))))
                       str)
str ⇒ "StUdLy"
```

`string-fold` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold` (*kons, knil, s, start, end*) [C Function]

Fold *kons* over the characters of *s*, with *knil* as the terminating element, from left to right. *kons* must expect two arguments: The actual character and the last result of *kons*' application.

`string-fold-right` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold_right` (*kons, knil, s, start, end*) [C Function]

Fold *kons* over the characters of *s*, with *knil* as the terminating element, from right to left. *kons* must expect two arguments: The actual character and the last result of *kons*' application.

`string-unfold` *p f g seed* [*base* [*make_final*]] [Scheme Procedure]
`scm_string_unfold` (*p, f, g, seed, base, make_final*) [C Function]

- *g* is used to generate a series of *seed* values from the initial *seed*: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), . . .
- *p* tells us when to stop – when it returns true when applied to one of these seed values.
- *f* maps each seed value to the corresponding character in the result string. These chars are assembled into the string in a left-to-right order.

- *base* is the optional initial/leftmost portion of the constructed string; it default to the empty string.
- *make_final* is applied to the terminal seed value (on which *p* returns true) to produce the final/rightmost portion of the constructed string. It defaults to `(lambda (x) )`.

`string-unfold-right` *p f g seed* [*base* [*make_final*]] [Scheme Procedure]
`scm_string_unfold_right` (*p, f, g, seed, base, make_final*) [C Function]

- *g* is used to generate a series of *seed* values from the initial *seed*: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of these seed values.
- *f* maps each seed value to the corresponding character in the result string. These chars are assembled into the string in a right-to-left order.
- *base* is the optional initial/rightmost portion of the constructed string; it default to the empty string.
- *make_final* is applied to the terminal seed value (on which *p* returns true) to produce the final/leftmost portion of the constructed string. It defaults to `(lambda (x) )`.

5.5.5.12 Miscellaneous String Operations

`xsubstring` *s from* [*to* [*start* [*end*]]] [Scheme Procedure]
`scm_xsubstring` (*s, from, to, start, end*) [C Function]

This is the *extended substring* procedure that implements replicated copying of a substring of some string.

s is a string, *start* and *end* are optional arguments that demarcate a substring of *s*, defaulting to 0 and the length of *s*. Replicate this substring up and down index space, in both the positive and negative directions. `xsubstring` returns the substring of this string beginning at index *from*, and ending at *to*, which defaults to *from* + (*end* - *start*).

`string-xcopy!` *target tstart s sfrom* [*sto* [*start* [*end*]]] [Scheme Procedure]
`scm_string_xcopy_x` (*target, tstart, s, sfrom, sto, start, end*) [C Function]

Exactly the same as `xsubstring`, but the extracted text is written into the string *target* starting at index *tstart*. The operation is not defined if `(eq? target s)` or these arguments share storage – you cannot copy a string on top of itself.

`string-replace` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_replace` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the string *s1*, but with the characters *start1* ... *end1* replaced by the characters *start2* ... *end2* from *s2*.

`string-tokenize` *s* [*token_set* [*start* [*end*]]] [Scheme Procedure]
`scm_string_tokenize` (*s, token_set, start, end*) [C Function]

Split the string *s* into a list of substrings, where each substring is a maximal non-empty contiguous sequence of characters from the character set *token_set*, which defaults to `char-set:graphic`. If *start* or *end* indices are provided, they restrict `string-tokenize` to operating on the indicated substring of *s*.

`string-filter` *s char_pred [start [end]]* [Scheme Procedure]
`scm_string_filter` *(s, char_pred, start, end)* [C Function]

Filter the string *s*, retaining only those characters which satisfy *char_pred*.

If *char_pred* is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

`string-delete` *s char_pred [start [end]]* [Scheme Procedure]
`scm_string_delete` *(s, char_pred, start, end)* [C Function]

Delete characters satisfying *char_pred* from *s*.

If *char_pred* is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

5.5.5.13 Conversion to/from C

When creating a Scheme string from a C string or when converting a Scheme string to a C string, the concept of character encoding becomes important.

In C, a string is just a sequence of bytes, and the character encoding describes the relation between these bytes and the actual characters that make up the string. For Scheme strings, character encoding is not an issue (most of the time), since in Scheme you never get to see the bytes, only the characters.

Well, ideally, anyway. Right now, Guile simply equates Scheme characters and bytes, ignoring the possibility of multi-byte encodings completely. This will change in the future, where Guile will use Unicode codepoints as its characters and UTF-8 or some other encoding as its internal encoding. When you exclusively use the functions listed in this section, you are 'future-proof'.

Converting a Scheme string to a C string will often allocate fresh memory to hold the result. You must take care that this memory is properly freed eventually. In many cases, this can be achieved by using `scm_dynwind_free` inside an appropriate dynwind context, See Section 5.11.9 [Dynamic Wind], page 266.

`SCM scm_from_locale_string` *(const char *str)* [C Function]
`SCM scm_from_locale_stringn` *(const char *str, size_t len)* [C Function]

Creates a new Scheme string that has the same contents as *str* when interpreted in the current locale character encoding.

For `scm_from_locale_string`, *str* must be null-terminated.

For `scm_from_locale_stringn`, *len* specifies the length of *str* in bytes, and *str* does not need to be null-terminated. If *len* is `(size_t)-1`, then *str* does need to be null-terminated and the real length will be found with `strlen`.

`SCM scm_take_locale_string` *(char *str)* [C Function]
`SCM scm_take_locale_stringn` *(char *str, size_t len)* [C Function]

Like `scm_from_locale_string` and `scm_from_locale_stringn`, respectively, but also frees *str* with `free` eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.

`char * scm_to_locale_string` (*SCM str*) [C Function]
`char * scm_to_locale_stringn` (*SCM str, size_t *lenp*) [C Function]

Returns a C string in the current locale encoding with the same contents as *str*. The C string must be freed with `free` eventually, maybe by using `scm_dynwind_free`, See Section 5.11.9 [Dynamic Wind], page 266.

For `scm_to_locale_string`, the returned string is null-terminated and an error is signalled when *str* contains `#\nul` characters.

For `scm_to_locale_stringn` and *lenp* not `NULL`, *str* might contain `#\nul` characters and the length of the returned string in bytes is stored in `*lenp`. The returned string will not be null-terminated in this case. If *lenp* is `NULL`, `scm_to_locale_stringn` behaves like `scm_to_locale_string`.

`size_t scm_to_locale_stringbuf` (*SCM str, char *buf, size_t max_len*) [C Function]

Puts *str* as a C string in the current locale encoding into the memory pointed to by *buf*. The buffer at *buf* has room for *max_len* bytes and `scm_to_local_stringbuf` will never store more than that. No terminating `'\0'` will be stored.

The return value of `scm_to_locale_stringbuf` is the number of bytes that are needed for all of *str*, regardless of whether *buf* was large enough to hold them. Thus, when the return value is larger than *max_len*, only *max_len* bytes have been stored and you probably need to try again with a larger buffer.

5.5.6 Regular Expressions

A *regular expression* (or *regexp*) is a pattern that describes a whole class of strings. A full description of regular expressions and their syntax is beyond the scope of this manual; an introduction can be found in the Emacs manual (see section "Syntax of Regular Expressions" in *The GNU Emacs Manual*), or in many general Unix reference books.

If your system does not include a POSIX regular expression library, and you have not linked Guile with a third-party regexp library such as Rx, these functions will not be available. You can tell whether your Guile installation includes regular expression support by checking whether `(provided? 'regex)` returns true.

The following regexp and string matching features are provided by the `(ice-9 regex)` module. Before using the described functions, you should load this module by executing `(use-modules (ice-9 regex))`.

5.5.6.1 Regexp Functions

By default, Guile supports POSIX extended regular expressions. That means that the characters '`(`', '`)`', '`+`' and '`?`' are special, and must be escaped if you wish to match the literal characters.

This regular expression interface was modeled after that implemented by SCSH, the Scheme Shell. It is intended to be upwardly compatible with SCSH regular expressions.

Zero bytes (`#\nul`) cannot be used in regex patterns or input strings, since the underlying C functions treat that as the end of string. If there's a zero byte an error is thrown.

Patterns and input strings are treated as being in the locale character set if `setlocale` has been called (see Section 6.2.13 [Locales], page 417), and in a multibyte locale this includes treating multi-byte sequences as a single character. (Guile strings are currently merely bytes, though this may change in the future, See Section 5.5.5.13 [Conversion to/from C], page 145.)

`string-match` *pattern str* [*start*] [Scheme Procedure]

Compile the string *pattern* into a regular expression and compare it with *str*. The optional numeric argument *start* specifies the position of *str* at which to begin matching.

`string-match` returns a *match structure* which describes what, if anything, was matched by the regular expression. See Section 5.5.6.2 [Match Structures], page 150. If *str* does not match *pattern* at all, `string-match` returns `#f`.

Two examples of a match follow. In the first example, the pattern matches the four digits in the match string. In the second, the pattern matches nothing.

```
(string-match "[0-9][0-9][0-9][0-9]" "blah2002")
⇒ #("blah2002" (4 . 8))

(string-match "[A-Za-z]" "123456")
⇒ #f
```

Each time `string-match` is called, it must compile its *pattern* argument into a regular expression structure. This operation is expensive, which makes `string-match` inefficient if the same regular expression is used several times (for example, in a loop). For better performance, you can compile a regular expression in advance and then match strings against the compiled regexp.

`make-regexp` *pat flag...* [Scheme Procedure]
`scm_make_regexp` (*pat, flaglst*) [C Function]

Compile the regular expression described by *pat*, and return the compiled regexp structure. If *pat* does not describe a legal regular expression, `make-regexp` throws a `regular-expression-syntax` error.

The *flag* arguments change the behavior of the compiled regular expression. The following values may be supplied:

`regexp/icase` [Variable]

Consider uppercase and lowercase letters to be the same when matching.

`regexp/newline` [Variable]

If a newline appears in the target string, then permit the '`^`' and '`$`' operators to match immediately after or immediately before the newline, respectively. Also, the '`.`' and '`[^...]`' operators will never match a newline character. The intent of this flag is to treat the target string as a buffer containing many lines of text, and the regular expression as a pattern that may match a single one of those lines.

`regexp/basic` [Variable]

Compile a basic ("obsolete") regexp instead of the extended ("modern") regexps that are the default. Basic regexps do not consider '`|`', '`+`' or '`?`' to be special characters, and require the '`{...}`' and '`(...)`' metacharacters to be backslash-escaped (see Section 5.5.6.3 [Backslash Escapes], page 152). There are several other differences between basic and extended regular expressions, but these are the most significant.

`regexp/extended` [Variable]
Compile an extended regular expression rather than a basic regexp. This is the default behavior; this flag will not usually be needed. If a call to `make-regexp` includes both `regexp/basic` and `regexp/extended` flags, the one which comes last will override the earlier one.

`regexp-exec` *rx str* [*start* [*flags*]] [Scheme Procedure]
`scm_regexp_exec` (*rx, str, start, flags*) [C Function]
Match the compiled regular expression *rx* against `str`. If the optional integer *start* argument is provided, begin matching from that position in the string. Return a match structure describing the results of the match, or `#f` if no match could be found.

The *flags* argument changes the matching behavior. The following flag values may be supplied, use `logior` (see Section 5.5.2.14 [Bitwise Operations], page 117) to combine them,

`regexp/notbol` [Variable]
Consider that the *start* offset into *str* is not the beginning of a line and should not match operator '`^`'.

If *rx* was created with the `regexp/newline` option above, '`^`' will still match after a newline in *str*.

`regexp/noteol` [Variable]
Consider that the end of *str* is not the end of a line and should not match operator '`$`'.

If *rx* was created with the `regexp/newline` option above, '`$`' will still match before a newline in *str*.

```
;; Regexp to match uppercase letters
(define r (make-regexp "[A-Z]*"))

;; Regexp to match letters, ignoring case
(define ri (make-regexp "[A-Z]*" regexp/icase))

;; Search for bob using regexp r
(match:substring (regexp-exec r "bob"))
⇒ ""                  ; no match

;; Search for bob using regexp ri
(match:substring (regexp-exec ri "Bob"))
⇒ "Bob"               ; matched case insensitive
```

`regexp?` *obj* [Scheme Procedure]

`scm_regexp_p` (*obj*) [C Function]

Return `#t` if *obj* is a compiled regular expression, or `#f` otherwise.

`list-matches` *regexp str* [*flags*] [Scheme Procedure]

Return a list of match structures which are the non-overlapping matches of *regexp* in *str*. *regexp* can be either a pattern string or a compiled regexp. The *flags* argument is as per `regexp-exec` above.

```
(map match:substring (list-matches "[a-z]+" "abc 42 def 78"))
⇒ ("abc" "def")
```

`fold-matches` *regexp str init proc* [*flags*] [Scheme Procedure]

Apply *proc* to the non-overlapping matches of *regexp* in *str*, to build a result. *regexp* can be either a pattern string or a compiled regexp. The *flags* argument is as per `regexp-exec` above.

proc is called as `(`*proc* `match prev)` where *match* is a match structure and *prev* is the previous return from *proc*. For the first call *prev* is the given *init* parameter. `fold-matches` returns the final value from *proc*.

For example to count matches,

```
(fold-matches "[a-z][0-9]" "abc x1 def y2" 0
              (lambda (match count)
                (1+ count)))
⇒ 2
```

Regular expressions are commonly used to find patterns in one string and replace them with the contents of another string. The following functions are convenient ways to do this.

`regexp-substitute` *port match* [*item...*] [Scheme Procedure]

Write to *port* selected parts of the match structure *match*. Or if *port* is `#f` then form a string from those parts and return that.

Each *item* specifies a part to be written, and may be one of the following,

- A string. String arguments are written out verbatim.
- An integer. The submatch with that number is written (`match:substring`). Zero is the entire match.
- The symbol '`pre`'. The portion of the matched string preceding the regexp match is written (`match:prefix`).
- The symbol '`post`'. The portion of the matched string following the regexp match is written (`match:suffix`).

For example, changing a match and retaining the text before and after,

```
(regexp-substitute #f (string-match "[0-9]+" "number 25 is good")
                   'pre "37" 'post)
⇒ "number 37 is good"
```

Or matching a YYYYMMDD format date such as '`20020828`' and re-ordering and hyphenating the fields.

```
(define date-regex "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute #f (string-match date-regex s)
                   'pre 2 "-" 3 "-" 1 'post " (" 0 ")")
⇒ "Date 04-29-2002 12am. (20020429)"
```

`regexp-substitute/global` *port regexp target* [*item. . .*] [Scheme Procedure]

Write to *port* selected parts of matches of *regexp* in *target*. If *port* is `#f` then form a string from those parts and return that. *regexp* can be a string or a compiled regex.

This is similar to `regexp-substitute`, but allows global substitutions on *target*. Each *item* behaves as per `regexp-substitute`, with the following differences,

- A function. Called as `(`*`item`*` match)` with the match structure for the *regexp* match, it should return a string to be written to *port*.
- The symbol '`post`'. This doesn't output anything, but instead causes `regexp-substitute/global` to recurse on the unmatched portion of *target*.

  This *must* be supplied to perform a global search and replace on *target*; without it `regexp-substitute/global` returns after a single match and output.

For example, to collapse runs of tabs and spaces to a single hyphen each,

```
(regexp-substitute/global #f "[ \t]+"  "this   is   the text"
                          'pre "-" 'post)
⇒ "this-is-the-text"
```

Or using a function to reverse the letters in each word,

```
(regexp-substitute/global #f "[a-z]+"  "to do and not-do"
  'pre (lambda (m) (string-reverse (match:substring m))) 'post)
⇒ "ot od dna ton-od"
```

Without the `post` symbol, just one regexp match is made. For example the following is the date example from `regexp-substitute` above, without the need for the separate `string-match` call.

```
(define date-regex "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute/global #f date-regex s
                          'pre 2 "-" 3 "-" 1 'post " (" 0 ")")

⇒ "Date 04-29-2002 12am. (20020429)"
```

5.5.6.2 Match Structures

A *match structure* is the object returned by `string-match` and `regexp-exec`. It describes which portion of a string, if any, matched the given regular expression. Match structures include: a reference to the string that was checked for matches; the starting and ending positions of the regexp match; and, if the regexp included any parenthesized subexpressions, the starting and ending positions of each submatch.

In each of the regexp match functions described below, the `match` argument must be a match structure returned by a previous call to `string-match` or `regexp-exec`. Most of these functions return some information about the original target string that was matched against a regular expression; we will call that string *target* for easy reference.

`regexp-match?` *obj* [Scheme Procedure]
: Return `#t` if *obj* is a match structure returned by a previous call to `regexp-exec`, or `#f` otherwise.

`match:substring` *match* [*n*] [Scheme Procedure]
: Return the portion of *target* matched by subexpression number *n*. Submatch 0 (the default) represents the entire regexp match. If the regular expression as a whole matched, but the subexpression number *n* did not match, return `#f`.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:substring s)
⇒ "2002"

;; match starting at offset 6 in the string
(match:substring
  (string-match "[0-9][0-9][0-9][0-9]" "blah987654" 6))
⇒ "7654"
```

`match:start` *match* [*n*] [Scheme Procedure]
: Return the starting position of submatch number *n*.

In the following example, the result is 4, since the match starts at character index 4:

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:start s)
⇒ 4
```

`match:end` *match* [*n*] [Scheme Procedure]
: Return the ending position of submatch number *n*.

In the following example, the result is 8, since the match runs between characters 4 and 8 (i.e. the “2002”).

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:end s)
⇒ 8
```

`match:prefix` *match* [Scheme Procedure]
: Return the unmatched portion of *target* preceding the regexp match.

```
    (define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
    (match:prefix s)
    ⇒ "blah"
```

`match:suffix` *match* [Scheme Procedure]
: Return the unmatched portion of *target* following the regexp match.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:suffix s)
⇒ "foo"
```

`match:count` *match* [Scheme Procedure]
: Return the number of parenthesized subexpressions from *match*. Note that the entire regular expression match itself counts as a subexpression, and failed submatches are included in the count.

`match:string` *match* [Scheme Procedure]

Return the original *target* string.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:string s)
⇒ "blah2002foo"
```

5.5.6.3 Backslash Escapes

Sometimes you will want a regexp to match characters like '`*`' or '`$`' exactly. For example, to check whether a particular string represents a menu entry from an Info node, it would be useful to match it against a regexp like '`^* [^:]*::`'. However, this won't work; because the asterisk is a metacharacter, it won't match the '`*`' at the beginning of the string. In this case, we want to make the first asterisk un-magic.

You can do this by preceding the metacharacter with a backslash character '`\`'. (This is also called *quoting* the metacharacter, and is known as a *backslash escape*.) When Guile sees a backslash in a regular expression, it considers the following glyph to be an ordinary character, no matter what special meaning it would ordinarily have. Therefore, we can make the above example work by changing the regexp to '`^\* [^:]*::`'. The '`\*`' sequence tells the regular expression engine to match only a single asterisk in the target string.

Since the backslash is itself a metacharacter, you may force a regexp to match a backslash in the target string by preceding the backslash with itself. For example, to find variable references in a TEX program, you might want to find occurrences of the string '`\let\`' followed by any number of alphabetic characters. The regular expression '`\\let\\[A-Za-z]*`' would do this: the double backslashes in the regexp each match a single backslash in the target string.

`regexp-quote` *str* [Scheme Procedure]

Quote each special character found in *str* with a backslash, and return the resulting string.

Very important: Using backslash escapes in Guile source code (as in Emacs Lisp or C) can be tricky, because the backslash character has special meaning for the Guile reader. For example, if Guile encounters the character sequence '`\n`' in the middle of a string while processing Scheme code, it replaces those characters with a newline character. Similarly, the character sequence '`\t`' is replaced by a horizontal tab. Several of these *escape sequences* are processed by the Guile reader before your code is executed. Unrecognized escape sequences are ignored: if the characters '`\*`' appear in a string, they will be translated to the single character '`*`'.

This translation is obviously undesirable for regular expressions, since we want to be able to include backslashes in a string in order to escape regexp metacharacters. Therefore, to make sure that a backslash is preserved in a string in your Guile program, you must use *two* consecutive backslashes:

```
(define Info-menu-entry-pattern (make-regexp "^\\* [^:]*"))
```

The string in this example is preprocessed by the Guile reader before any code is executed. The resulting argument to `make-regexp` is the string '`^\* [^:]*`', which is what we really want.

This also means that in order to write a regular expression that matches a single backslash character, the regular expression string in the source code must include *four* backslashes. Each consecutive pair of backslashes gets translated by the Guile reader to a single backslash, and the resulting double-backslash is interpreted by the regexp engine as matching a single backslash character. Hence:

```
(define tex-variable-pattern (make-regexp "\\\\let\\\\=[A-Za-z]*"))
```

The reason for the unwieldiness of this syntax is historical. Both regular expression pattern matchers and Unix string processing systems have traditionally used backslashes with the special meanings described above. The POSIX regular expression specification and ANSI C standard both require these semantics. Attempting to abandon either convention would cause other kinds of compatibility problems, possibly more severe ones. Therefore, without extending the Scheme reader to support strings with different quoting conventions (an ungainly and confusing extension when implemented in other languages), we must adhere to this cumbersome escape syntax.

5.5.7 Symbols

Symbols in Scheme are widely used in three ways: as items of discrete data, as lookup keys for alists and hash tables, and to denote variable references.

A *symbol* is similar to a string in that it is defined by a sequence of characters. The sequence of characters is known as the symbol's *name*. In the usual case — that is, where the symbol's name doesn't include any characters that could be confused with other elements of Scheme syntax — a symbol is written in a Scheme program by writing the sequence of characters that make up the name, *without* any quotation marks or other special syntax. For example, the symbol whose name is "multiply-by-2" is written, simply:

```
multiply-by-2
```

Notice how this differs from a *string* with contents "multiply-by-2", which is written with double quotation marks, like this:

```
"multiply-by-2"
```

Looking beyond how they are written, symbols are different from strings in two important respects.

The first important difference is uniqueness. If the same-looking string is read twice from two different places in a program, the result is two *different* string objects whose contents just happen to be the same. If, on the other hand, the same-looking symbol is read twice from two different places in a program, the result is the *same* symbol object both times.

Given two read symbols, you can use `eq?` to test whether they are the same (that is, have the same name). `eq?` is the most efficient comparison operator in Scheme, and comparing two symbols like this is as fast as comparing, for example, two numbers. Given two strings, on the other hand, you must use `equal?` or `string=?`, which are much slower comparison operators, to determine whether the strings have the same contents.

```
(define sym1 (quote hello))
(define sym2 (quote hello))
(eq? sym1 sym2) ⇒ #t

(define str1 "hello")
```

```
(define str2 "hello")
(eq? str1 str2) ⇒ #f
(equal? str1 str2) ⇒ #t
```

The second important difference is that symbols, unlike strings, are not self-evaluating. This is why we need the `(quote ...)`s in the example above: `(quote hello)` evaluates to the symbol named "hello" itself, whereas an unquoted `hello` is *read* as the symbol named "hello" and evaluated as a variable reference . . . about which more below (see Section 5.5.7.3 [Symbol Variables], page 156).

5.5.7.1 Symbols as Discrete Data

Numbers and symbols are similar to the extent that they both lend themselves to `eq?` comparison. But symbols are more descriptive than numbers, because a symbol's name can be used directly to describe the concept for which that symbol stands.

For example, imagine that you need to represent some colours in a computer program. Using numbers, you would have to choose arbitrarily some mapping between numbers and colours, and then take care to use that mapping consistently:

```
;; 1=red, 2=green, 3=purple

(if (eq? (colour-of car) 1)
    ...)
```

You can make the mapping more explicit and the code more readable by defining constants:

```
(define red 1)
(define green 2)
(define purple 3)

(if (eq? (colour-of car) red)
    ...)
```

But the simplest and clearest approach is not to use numbers at all, but symbols whose names specify the colours that they refer to:

```
(if (eq? (colour-of car) 'red)
    ...)
```

The descriptive advantages of symbols over numbers increase as the set of concepts that you want to describe grows. Suppose that a car object can have other properties as well, such as whether it has or uses:

- automatic or manual transmission
- leaded or unleaded fuel
- power steering (or not).

Then a car's combined property set could be naturally represented and manipulated as a list of symbols:

```
(properties-of car1)
⇒
(red manual unleaded power-steering)

(if (memq 'power-steering (properties-of car1))
        (display "Unfit people can drive this car.\n")
        (display "You'll need strong arms to drive this car!\n"))
  ⊣
  Unfit people can drive this car.
```

Remember, the fundamental property of symbols that we are relying on here is that an occurrence of `'red` in one part of a program is an *indistinguishable* symbol from an occurrence of `'red` in another part of a program; this means that symbols can usefully be compared using `eq?`. At the same time, symbols have naturally descriptive names. This combination of efficiency and descriptive power makes them ideal for use as discrete data.

5.5.7.2 Symbols as Lookup Keys

Given their efficiency and descriptive power, it is natural to use symbols as the keys in an association list or hash table.

To illustrate this, consider a more structured representation of the car properties example from the preceding subsection. Rather than mixing all the properties up together in a flat list, we could use an association list like this:

```
(define car1-properties '((colour . red)
                          (transmission . manual)
                          (fuel . unleaded)
                          (steering . power-assisted)))
```

Notice how this structure is more explicit and extensible than the flat list. For example it makes clear that `manual` refers to the transmission rather than, say, the windows or the locking of the car. It also allows further properties to use the same symbols among their possible values without becoming ambiguous:

```
(define car1-properties '((colour . red)
                          (transmission . manual)
                          (fuel . unleaded)
                          (steering . power-assisted)
                          (seat-colour . red)
                          (locking . manual)))
```

With a representation like this, it is easy to use the efficient `assq-XXX` family of procedures (see Section 5.6.11 [Association Lists], page 210) to extract or change individual pieces of information:

```
(assq-ref car1-properties 'fuel) ⇒ unleaded
(assq-ref car1-properties 'transmission) ⇒ manual

(assq-set! car1-properties 'seat-colour 'black)
⇒
((colour . red)
 (transmission . manual)
 (fuel . unleaded)
 (steering . power-assisted)
 (seat-colour . black)
 (locking . manual)))
```

Hash tables also have keys, and exactly the same arguments apply to the use of symbols in hash tables as in association lists. The hash value that Guile uses to decide where to add a symbol-keyed entry to a hash table can be obtained by calling the `symbol-hash` procedure:

`symbol-hash` *symbol* [Scheme Procedure]
`scm_symbol_hash` (*symbol*) [C Function]

Return a hash value for *symbol*.

See Section 5.6.12 [Hash Tables], page 215 for information about hash tables in general, and for why you might choose to use a hash table rather than an association list.

5.5.7.3 Symbols as Denoting Variables

When an unquoted symbol in a Scheme program is evaluated, it is interpreted as a variable reference, and the result of the evaluation is the appropriate variable's value.

For example, when the expression `(string-length "abcd")` is read and evaluated, the sequence of characters `string-length` is read as the symbol whose name is "string-length". This symbol is associated with a variable whose value is the procedure that implements string length calculation. Therefore evaluation of the `string-length` symbol results in that procedure.

The details of the connection between an unquoted symbol and the variable to which it refers are explained elsewhere. See Section 5.10 [Binding Constructs], page 247, for how associations between symbols and variables are created, and Section 5.16 [Modules], page 303, for how those associations are affected by Guile's module system.

5.5.7.4 Operations Related to Symbols

Given any Scheme value, you can determine whether it is a symbol using the `symbol?` primitive:

`symbol?` *obj* [Scheme Procedure]
`scm_symbol_p` (*obj*) [C Function]

Return `#t` if *obj* is a symbol, otherwise return `#f`.

`int scm_is_symbol` (*SCM val*) [C Function]

Equivalent to `scm_is_true (scm_symbol_p (val))`.

Once you know that you have a symbol, you can obtain its name as a string by calling `symbol->string`. Note that Guile differs by default from R5RS on the details of `symbol->string` as regards case-sensitivity:

`symbol->string` *s* [Scheme Procedure]
`scm_symbol_to_string` (*s*) [C Function]

Return the name of symbol *s* as a string. By default, Guile reads symbols case-sensitively, so the string returned will have the same case variation as the sequence of characters that caused *s* to be created.

If Guile is set to read symbols case-insensitively (as specified by R5RS), and *s* comes into being as part of a literal expression (see section "Literal expressions" in *The*

Revised^5 Report on Scheme) or by a call to the `read` or `string-ci->symbol` procedures, Guile converts any alphabetic characters in the symbol's name to lower case before creating the symbol object, so the string returned here will be in lower case.

If *s* was created by `string->symbol`, the case of characters in the string returned will be the same as that in the string that was passed to `string->symbol`, regardless of Guile's case-sensitivity setting at the time *s* was created.

It is an error to apply mutation procedures like `string-set!` to strings returned by this procedure.

Most symbols are created by writing them literally in code. However it is also possible to create symbols programmatically using the following `string->symbol` and `string-ci->symbol` procedures:

`string->symbol` *string* [Scheme Procedure]
`scm_string_to_symbol` (*string*) [C Function]

Return the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves.

`string-ci->symbol` *str* [Scheme Procedure]
`scm_string_ci_to_symbol` (*str*) [C Function]

Return the symbol whose name is *str*. If Guile is currently reading symbols case-insensitively, *str* is converted to lowercase before the returned symbol is looked up or created.

The following examples illustrate Guile's detailed behaviour as regards the case-sensitivity of symbols:

```
(read-enable 'case-insensitive)   ; R5RS compliant behaviour

(symbol->string 'flying-fish)    ⇒ "flying-fish"
(symbol->string 'Martin)         ⇒ "martin"
(symbol->string
   (string->symbol "Malvina"))   ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi)  ⇒ #t
(string->symbol "mISSISSIppi")   ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #f
(eq? 'LolliPop
  (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D."))) ⇒ #t

(read-disable 'case-insensitive)   ; Guile default behaviour

(symbol->string 'flying-fish)    ⇒ "flying-fish"
```

```
(symbol->string 'Martin)         ⇒ "Martin"
(symbol->string
   (string->symbol "Malvina"))   ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi)  ⇒ #f
(string->symbol "mISSISSIppi")   ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #t
(eq? 'LolliPop
  (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D."))) ⇒ #t
```

From C, there are lower level functions that construct a Scheme symbol from a C string in the current locale encoding.

When you want to do more from C, you should convert between symbols and strings using `scm_symbol_to_string` and `scm_string_to_symbol` and work with the strings.

`scm_from_locale_symbol` (*const char *name*) [C Function]
`scm_from_locale_symboln` (*const char *name, size_t len*) [C Function]

> Construct and return a Scheme symbol whose name is specified by *name*. For `scm_from_locale_symbol`, *name* must be null terminated; for `scm_from_locale_symboln` the length of *name* is specified explicitly by *len*.

`SCM scm_take_locale_symbol` (*char *str*) [C Function]
`SCM scm_take_locale_symboln` (*char *str, size_t len*) [C Function]

> Like `scm_from_locale_symbol` and `scm_from_locale_symboln`, respectively, but also frees *str* with `free` eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.

Finally, some applications, especially those that generate new Scheme code dynamically, need to generate symbols for use in the generated code. The `gensym` primitive meets this need:

`gensym` [*prefix*] [Scheme Procedure]
`scm_gensym` (*prefix*) [C Function]

> Create a new symbol with a name constructed from a prefix and a counter value. The string *prefix* can be specified as an optional argument. Default prefix is ‘ `g`’. The counter is increased by 1 at each call. There is no provision for resetting the counter.

The symbols generated by `gensym` are *likely* to be unique, since their names begin with a space and it is only otherwise possible to generate such symbols if a programmer goes out of their way to do so. Uniqueness can be guaranteed by instead using uninterned symbols (see Section 5.5.7.7 [Symbol Uninterned], page 160), though they can't be usefully written out and read back in.

5.5.7.5 Function Slots and Property Lists

In traditional Lisp dialects, symbols are often understood as having three kinds of value at once:

- a *variable* value, which is used when the symbol appears in code in a variable reference context
- a *function* value, which is used when the symbol appears in code in a function name position (i.e. as the first element in an unquoted list)
- a *property list* value, which is used when the symbol is given as the first argument to Lisp's `put` or `get` functions.

Although Scheme (as one of its simplifications with respect to Lisp) does away with the distinction between variable and function namespaces, Guile currently retains some elements of the traditional structure in case they turn out to be useful when implementing translators for other languages, in particular Emacs Lisp.

Specifically, Guile symbols have two extra slots. for a symbol's property list, and for its "function value." The following procedures are provided to access these slots.

`symbol-fref` *symbol* [Scheme Procedure]
`scm_symbol_fref` (*symbol*) [C Function]
Return the contents of *symbol*'s *function slot*.

`symbol-fset!` *symbol value* [Scheme Procedure]
`scm_symbol_fset_x` (*symbol, value*) [C Function]
Set the contents of *symbol*'s function slot to *value*.

`symbol-pref` *symbol* [Scheme Procedure]
`scm_symbol_pref` (*symbol*) [C Function]
Return the *property list* currently associated with *symbol*.

`symbol-pset!` *symbol value* [Scheme Procedure]
`scm_symbol_pset_x` (*symbol, value*) [C Function]
Set *symbol*'s property list to *value*.

`symbol-property` *sym prop* [Scheme Procedure]
From *sym*'s property list, return the value for property *prop*. The assumption is that *sym*'s property list is an association list whose keys are distinguished from each other using `equal?`; *prop* should be one of the keys in that list. If the property list has no entry for *prop*, `symbol-property` returns `#f`.

`set-symbol-property!` *sym prop val* [Scheme Procedure]
In *sym*'s property list, set the value for property *prop* to *val*, or add a new entry for *prop*, with value *val*, if none already exists. For the structure of the property list, see `symbol-property`.

`symbol-property-remove!` *sym prop* [Scheme Procedure]
From *sym*'s property list, remove the entry for property *prop*, if there is one. For the structure of the property list, see `symbol-property`.

Support for these extra slots may be removed in a future release, and it is probably better to avoid using them. For a more modern and Schemely approach to properties, see Section 5.9.2 [Object Properties], page 238.

5.5.7.6 Extended Read Syntax for Symbols

The read syntax for a symbol is a sequence of letters, digits, and *extended alphabetic characters*, beginning with a character that cannot begin a number. In addition, the special cases of `+`, `-`, and `...` are read as symbols even though numbers can begin with `+`, `-` or `.`.

Extended alphabetic characters may be used within identifiers as if they were letters. The set of extended alphabetic characters is:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

In addition to the standard read syntax defined above (which is taken from R5RS (see section "Formal syntax" in *The Revised^5 Report on Scheme*)), Guile provides an extended symbol read syntax that allows the inclusion of unusual characters such as space characters, newlines and parentheses. If (for whatever reason) you need to write a symbol containing characters not mentioned above, you can do so as follows.

- Begin the symbol with the characters `#{`,
- write the characters of the symbol and
- finish the symbol with the characters `}#`.

Here are a few examples of this form of read syntax. The first symbol needs to use extended syntax because it contains a space character, the second because it contains a line break, and the last because it looks like a number.

```
#{foo bar}#

#{what
ever}#

#{4242}#
```

Although Guile provides this extended read syntax for symbols, widespread usage of it is discouraged because it is not portable and not very readable.

5.5.7.7 Uninterned Symbols

What makes symbols useful is that they are automatically kept unique. There are no two symbols that are distinct objects but have the same name. But of course, there is no rule without exception. In addition to the normal symbols that have been discussed up to now, you can also create special *uninterned* symbols that behave slightly differently.

To understand what is different about them and why they might be useful, we look at how normal symbols are actually kept unique.

Whenever Guile wants to find the symbol with a specific name, for example during `read` or when executing `string->symbol`, it first looks into a table of all existing symbols to find out whether a symbol with the given name already exists. When this is the case, Guile just returns that symbol. When not, a new symbol with the name is created and entered into the table so that it can be found later.

Sometimes you might want to create a symbol that is guaranteed 'fresh', i.e. a symbol that did not exist previously. You might also want to somehow guarantee that no one else will ever unintentionally stumble across your symbol in the future. These properties of a symbol are often needed when generating code during macro expansion. When introducing new temporary variables, you want to guarantee that they don't conflict with variables in other people's code.

The simplest way to arrange for this is to create a new symbol but not enter it into the global table of all symbols. That way, no one will ever get access to your symbol by chance. Symbols that are not in the table are called *uninterned*. Of course, symbols that *are* in the table are called *interned*.

You create new uninterned symbols with the function `make-symbol`. You can test whether a symbol is interned or not with `symbol-interned?`.

Uninterned symbols break the rule that the name of a symbol uniquely identifies the symbol object. Because of this, they can not be written out and read back in like interned symbols. Currently, Guile has no support for reading uninterned symbols. Note that the function `gensym` does not return uninterned symbols for this reason.

`make-symbol` *name* [Scheme Procedure]
`scm_make_symbol` (*name*) [C Function]

> Return a new uninterned symbol with the name *name*. The returned symbol is guaranteed to be unique and future calls to `string->symbol` will not return it.

`symbol-interned?` *symbol* [Scheme Procedure]
`scm_symbol_interned_p` (*symbol*) [C Function]

> Return `#t` if *symbol* is interned, otherwise return `#f`.

For example:

```
(define foo-1 (string->symbol "foo"))
(define foo-2 (string->symbol "foo"))
(define foo-3 (make-symbol "foo"))
(define foo-4 (make-symbol "foo"))

(eq? foo-1 foo-2)
⇒ #t
; Two interned symbols with the same name are the same object,

(eq? foo-1 foo-3)
⇒ #f
; but a call to make-symbol with the same name returns a
; distinct object.

(eq? foo-3 foo-4)
⇒ #f
; A call to make-symbol always returns a new object, even for
; the same name.

foo-3
⇒ #<uninterned-symbol foo 8085290>
; Uninterned symbols print differently from interned symbols,

(symbol? foo-3)
```

```
⇒ #t
; but they are still symbols,

(symbol-interned? foo-3)
⇒ #f
; just not interned.
```

5.5.8 Keywords

Keywords are self-evaluating objects with a convenient read syntax that makes them easy to type.

Guile's keyword support conforms to R5RS, and adds a (switchable) read syntax extension to permit keywords to begin with `:` as well as `#:`.

5.5.8.1 Why Use Keywords?

Keywords are useful in contexts where a program or procedure wants to be able to accept a large number of optional arguments without making its interface unmanageable.

To illustrate this, consider a hypothetical `make-window` procedure, which creates a new window on the screen for drawing into using some graphical toolkit. There are many parameters that the caller might like to specify, but which could also be sensibly defaulted, for example:

- color depth – Default: the color depth for the screen
- background color – Default: white
- width – Default: 600
- height – Default: 400

If `make-window` did not use keywords, the caller would have to pass in a value for each possible argument, remembering the correct argument order and using a special value to indicate the default value for that argument:

```
(make-window 'default              ;; Color depth
             'default              ;; Background color
             800                   ;; Width
             100                   ;; Height
             ...)                  ;; More make-window arguments
```

With keywords, on the other hand, defaulted arguments are omitted, and non-default arguments are clearly tagged by the appropriate keyword. As a result, the invocation becomes much clearer:

```
(make-window #:width 800 #:height 100)
```

On the other hand, for a simpler procedure with few arguments, the use of keywords would be a hindrance rather than a help. The primitive procedure `cons`, for example, would not be improved if it had to be invoked as

```
(cons #:car x #:cdr y)
```

So the decision whether to use keywords or not is purely pragmatic: use them if they will clarify the procedure invocation at point of call.

5.5.8.2 Coding With Keywords

If a procedure wants to support keywords, it should take a rest argument and then use whatever means is convenient to extract keywords and their corresponding arguments from the contents of that rest argument.

The following example illustrates the principle: the code for `make-window` uses a helper procedure called `get-keyword-value` to extract individual keyword arguments from the rest argument.

```
(define (get-keyword-value args keyword default)
  (let ((kv (memq keyword args)))
    (if (and kv (>= (length kv) 2))
        (cadr kv)
        default)))

(define (make-window . args)
  (let ((depth  (get-keyword-value args #:depth  screen-depth))
        (bg     (get-keyword-value args #:bg     "white"))
        (width  (get-keyword-value args #:width  800))
        (height (get-keyword-value args #:height 100))
        ...)
    ...))
```

But you don't need to write `get-keyword-value`. The `(ice-9 optargs)` module provides a set of powerful macros that you can use to implement keyword-supporting procedures like this:

```
(use-modules (ice-9 optargs))

(define (make-window . args)
  (let-keywords args #f ((depth  screen-depth)
                         (bg     "white")
                         (width  800)
                         (height 100))
    ...))
```

Or, even more economically, like this:

```
(use-modules (ice-9 optargs))

(define* (make-window #:key (depth  screen-depth)
                            (bg     "white")
                            (width  800)
                            (height 100))
  ...)
```

For further details on `let-keywords`, `define*` and other facilities provided by the `(ice-9 optargs)` module, see Section 5.8.3 [Optional Arguments], page 226.

5.5.8.3 Keyword Read Syntax

Guile, by default, only recognizes a keyword syntax that is compatible with R5RS. A token of the form `#:NAME`, where `NAME` has the same syntax as a Scheme symbol (see Section 5.5.7.6

[Symbol Read Syntax], page 160), is the external representation of the keyword named `NAME`. Keyword objects print using this syntax as well, so values containing keyword objects can be read back into Guile. When used in an expression, keywords are self-quoting objects.

If the `keyword` read option is set to `'prefix`, Guile also recognizes the alternative read syntax `:NAME`. Otherwise, tokens of the form `:NAME` are read as symbols, as required by R5RS.

To enable and disable the alternative non-R5RS keyword syntax, you use the `read-set!` procedure documented in Section 5.18.3.2 [User level options interfaces], page 339 and Section 5.18.3.3 [Reader options], page 339.

```
(read-set! keywords 'prefix)

#:type
⇒
#:type

:type
⇒
#:type

(read-set! keywords #f)

#:type
⇒
#:type

:type
⊣
ERROR: In expression :type:
ERROR: Unbound variable: :type
ABORT: (unbound-variable)
```

5.5.8.4 Keyword Procedures

`keyword?` *obj* [Scheme Procedure]
`scm_keyword_p` (*obj*) [C Function]
  Return `#t` if the argument *obj* is a keyword, else `#f`.

`keyword->symbol` *keyword* [Scheme Procedure]
`scm_keyword_to_symbol` (*keyword*) [C Function]
  Return the symbol with the same name as *keyword*.

`symbol->keyword` *symbol* [Scheme Procedure]
`scm_symbol_to_keyword` (*symbol*) [C Function]
  Return the keyword with the same name as *symbol*.

`int scm_is_keyword` (*SCM obj*) [C Function]
  Equivalent to `scm_is_true (scm_keyword_p (obj))`.

`SCM scm_from_locale_keyword` (*const char *str*) [C Function]
`SCM scm_from_locale_keywordn` (*const char *str, size_t len*) [C Function]
  Equivalent to `scm_symbol_to_keyword (scm_from_locale_symbol (str))` and `scm_symbol_to_keyword (scm_from_locale_symboln (str, len))`, respectively.

5.5.9 "Functionality-Centric" Data Types

Procedures and macros are documented in their own chapter: see Section 5.8 [Procedures and Macros], page 225.

Variable objects are documented as part of the description of Guile's module system: see Section 5.16.5 [Variables], page 321.

Asyncs, dynamic roots and fluids are described in the chapter on scheduling: see Section 5.17 [Scheduling], page 323.

Hooks are documented in the chapter on general utility functions: see Section 5.9.6 [Hooks], page 241.

Ports are described in the chapter on I/O: see Section 5.12 [Input and Output], page 271.

5.6 Compound Data Types

This chapter describes Guile's compound data types. By *compound* we mean that the primary purpose of these data types is to act as containers for other kinds of data (including other compound objects). For instance, a (non-uniform) vector with length 5 is a container that can hold five arbitrary Scheme objects.

The various kinds of container object differ from each other in how their memory is allocated, how they are indexed, and how particular values can be looked up within them.

5.6.1 Pairs

Pairs are used to combine two Scheme objects into one compound object. Hence the name: A pair stores a pair of objects.

The data type *pair* is extremely important in Scheme, just like in any other Lisp dialect. The reason is that pairs are not only used to make two values available as one object, but that pairs are used for constructing lists of values. Because lists are so important in Scheme, they are described in a section of their own (see Section 5.6.2 [Lists], page 168).

Pairs can literally get entered in source code or at the REPL, in the so-called *dotted list* syntax. This syntax consists of an opening parentheses, the first element of the pair, a dot, the second element and a closing parentheses. The following example shows how a pair consisting of the two numbers 1 and 2, and a pair containing the symbols `foo` and `bar` can be entered. It is very important to write the whitespace before and after the dot, because otherwise the Scheme parser would not be able to figure out where to split the tokens.

```
(1 . 2)
(foo . bar)
```

But beware, if you want to try out these examples, you have to *quote* the expressions. More information about quotation is available in the section Section 5.13.1.1 [Expression Syntax], page 288. The correct way to try these examples is as follows.

```
'(1 . 2)
⇒
(1 . 2)
'(foo . bar)
⇒
(foo . bar)
```

A new pair is made by calling the procedure `cons` with two arguments. Then the argument values are stored into a newly allocated pair, and the pair is returned. The name `cons` stands for "construct". Use the procedure `pair?` to test whether a given Scheme object is a pair or not.

`cons` *x y* [Scheme Procedure]
`scm_cons` (*x*, *y*) [C Function]

Return a newly allocated pair whose car is *x* and whose cdr is *y*. The pair is guaranteed to be different (in the sense of `eq?`) from every previously existing object.

`pair?` *x* [Scheme Procedure]
`scm_pair_p` (*x*) [C Function]

Return `#t` if *x* is a pair; otherwise return `#f`.

`int scm_is_pair` (*SCM x*) [C Function]

Return 1 when `x` is a pair; otherwise return 0.

The two parts of a pair are traditionally called *car* and *cdr*. They can be retrieved with procedures of the same name (`car` and `cdr`), and can be modified with the procedures `set-car!` and `set-cdr!`. Since a very common operation in Scheme programs is to access the car of a car of a pair, or the car of the cdr of a pair, etc., the procedures called `caar`, `cadr` and so on are also predefined.

`car` *pair* [Scheme Procedure]
`cdr` *pair* [Scheme Procedure]
`scm_car` (*pair*) [C Function]
`scm_cdr` (*pair*) [C Function]

Return the car or the cdr of *pair*, respectively.

`SCM SCM_CAR` (*SCM pair*) [C Macro]
`SCM SCM_CDR` (*SCM pair*) [C Macro]

These two macros are the fastest way to access the car or cdr of a pair; they can be thought of as compiling into a single memory reference.

These macros do no checking at all. The argument *pair* must be a valid pair.

`cddr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]
`caddr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]
`caaar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]
`cdddar` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]

`caaaar` *pair* [Scheme Procedure]
`scm_cddr` (*pair*) [C Function]
`scm_cdar` (*pair*) [C Function]
`scm_cadr` (*pair*) [C Function]
`scm_caar` (*pair*) [C Function]
`scm_cdddr` (*pair*) [C Function]
`scm_cddar` (*pair*) [C Function]
`scm_cdadr` (*pair*) [C Function]
`scm_cdaar` (*pair*) [C Function]
`scm_caddr` (*pair*) [C Function]
`scm_cadar` (*pair*) [C Function]
`scm_caadr` (*pair*) [C Function]
`scm_caaar` (*pair*) [C Function]
`scm_cddddr` (*pair*) [C Function]
`scm_cdddar` (*pair*) [C Function]
`scm_cddadr` (*pair*) [C Function]
`scm_cddaar` (*pair*) [C Function]
`scm_cdaddr` (*pair*) [C Function]
`scm_cdadar` (*pair*) [C Function]
`scm_cdaadr` (*pair*) [C Function]
`scm_cdaaar` (*pair*) [C Function]
`scm_cadddr` (*pair*) [C Function]
`scm_caddar` (*pair*) [C Function]
`scm_cadadr` (*pair*) [C Function]
`scm_cadaar` (*pair*) [C Function]
`scm_caaddr` (*pair*) [C Function]
`scm_caadar` (*pair*) [C Function]
`scm_caaadr` (*pair*) [C Function]
`scm_caaaar` (*pair*) [C Function]

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

`cadr`, `caddr` and `cadddr` pick out the second, third or fourth elements of a list, respectively. SRFI-1 provides the same under the names `second`, `third` and `fourth` (see Section 6.4.3.3 [SRFI-1 Selectors], page 426).

`set-car!` *pair value* [Scheme Procedure]
`scm_set_car_x` (*pair*, *value*) [C Function]

Stores *value* in the car field of *pair*. The value returned by `set-car!` is unspecified.

`set-cdr!` *pair value* [Scheme Procedure]
`scm_set_cdr_x` (*pair*, *value*) [C Function]

Stores *value* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

5.6.2 Lists

A very important data type in Scheme—as well as in all other Lisp dialects—is the data type *list*.[1]

This is the short definition of what a list is:

- Either the empty list `()`,
- or a pair which has a list in its cdr.

5.6.2.1 List Read Syntax

The syntax for lists is an opening parentheses, then all the elements of the list (separated by whitespace) and finally a closing parentheses.[2].

```
(1 2 3)            ; a list of the numbers 1, 2 and 3
("foo" bar 3.1415) ; a string, a symbol and a real number
()                 ; the empty list
```

The last example needs a bit more explanation. A list with no elements, called the *empty list*, is special in some ways. It is used for terminating lists by storing it into the cdr of the last pair that makes up a list. An example will clear that up:

```
(car '(1))
⇒
1
(cdr '(1))
⇒
()
```

This example also shows that lists have to be quoted when written (see Section 5.13.1.1 [Expression Syntax], page 288), because they would otherwise be mistakingly taken as procedure applications (see Section 3.1.2.2 [Simple Invocation], page 16).

5.6.2.2 List Predicates

Often it is useful to test whether a given Scheme object is a list or not. List-processing procedures could use this information to test whether their input is valid, or they could do different things depending on the datatype of their arguments.

`list?` *x* [Scheme Procedure]
`scm_list_p` (*x*) [C Function]

Return `#t` iff *x* is a proper list, else `#f`.

The predicate `null?` is often used in list-processing code to tell whether a given list has run out of elements. That is, a loop somehow deals with the elements of a list until the list satisfies `null?`. Then, the algorithm terminates.

`null?` *x* [Scheme Procedure]
`scm_null_p` (*x*) [C Function]

Return `#t` iff *x* is the empty list, else `#f`.

[1] Strictly speaking, Scheme does not have a real datatype *list*. Lists are made up of *chained pairs*, and only exist by definition—a list is a chain of pairs which looks like a list.

[2] Note that there is no separation character between the list elements, like a comma or a semicolon.

`int scm_is_null` (*SCM x*) [C Function]

Return 1 when `x` is the empty list; otherwise return 0.

5.6.2.3 List Constructors

This section describes the procedures for constructing new lists. `list` simply returns a list where the elements are the arguments, `cons*` is similar, but the last argument is stored in the cdr of the last pair of the list.

`list` *elem1 . . . elemN* [Scheme Procedure]
`scm_list_1` (*elem1*) [C Function]
`scm_list_2` (*elem1, elem2*) [C Function]
`scm_list_3` (*elem1, elem2, elem3*) [C Function]
`scm_list_4` (*elem1, elem2, elem3, elem4*) [C Function]
`scm_list_5` (*elem1, elem2, elem3, elem4, elem5*) [C Function]
`scm_list_n` (*elem1, . . ., elemN,* `SCM_UNDEFINED`) [C Function]

Return a new list containing elements *elem1* to *elemN*.

`scm_list_n` takes a variable number of arguments, terminated by the special `SCM_UNDEFINED`. That final `SCM_UNDEFINED` is not included in the list. None of *elem1* to *elemN* can themselves be `SCM_UNDEFINED`, or `scm_list_n` will terminate at that point.

`cons*` *arg1 arg2 . . .* [Scheme Procedure]

Like `list`, but the last arg provides the tail of the constructed list, returning `(cons` ***arg1*** `(cons` ***arg2*** `(cons ...` ***argn*** `)))`. Requires at least one argument. If given one argument, that argument is returned as result. This function is called `list*` in some other Schemes and in Common LISP.

`list-copy` *lst* [Scheme Procedure]
`scm_list_copy` (*lst*) [C Function]

Return a (newly-created) copy of *lst*.

`make-list` *n* [*init*] [Scheme Procedure]

Create a list containing of *n* elements, where each element is initialized to *init*. *init* defaults to the empty list `()` if not given.

Note that `list-copy` only makes a copy of the pairs which make up the spine of the lists. The list elements are not copied, which means that modifying the elements of the new list also modifies the elements of the old list. On the other hand, applying procedures like `set-cdr!` or `delv!` to the new list will not alter the old list. If you also need to copy the list elements (making a deep copy), use the procedure `copy-tree` (see Section 5.9.4 [Copying], page 241).

5.6.2.4 List Selection

These procedures are used to get some information about a list, or to retrieve one or more elements of a list.

`length` *lst* [Scheme Procedure]
`scm_length` (*lst*) [C Function]

Return the number of elements in list *lst*.

`last-pair` *lst* [Scheme Procedure]
`scm_last_pair` (*lst*) [C Function]

Return the last pair in *lst*, signalling an error if *lst* is circular.

`list-ref` *list k* [Scheme Procedure]
`scm_list_ref` (*list*, *k*) [C Function]

Return the *k*th element from *list*.

`list-tail` *lst k* [Scheme Procedure]
`list-cdr-ref` *lst k* [Scheme Procedure]
`scm_list_tail` (*lst*, *k*) [C Function]

Return the "tail" of *lst* beginning with its *k*th element. The first element of the list is considered to be element 0.

`list-tail` and `list-cdr-ref` are identical. It may help to think of `list-cdr-ref` as accessing the *k*th cdr of the list, or returning the results of cdring *k* times down *lst*.

`list-head` *lst k* [Scheme Procedure]
`scm_list_head` (*lst*, *k*) [C Function]

Copy the first *k* elements from *lst* into a new list, and return it.

5.6.2.5 Append and Reverse

`append` and `append!` are used to concatenate two or more lists in order to form a new list. `reverse` and `reverse!` return lists with the same elements as their arguments, but in reverse order. The procedure variants with an `!` directly modify the pairs which form the list, whereas the other procedures create new pairs. This is why you should be careful when using the side-effecting variants.

`append` *lst1 ... lstN* [Scheme Procedure]
`append!` *lst1 ... lstN* [Scheme Procedure]
`scm_append` (*lstlst*) [C Function]
`scm_append_x` (*lstlst*) [C Function]

Return a list comprising all the elements of lists *lst1* to *lstN*.

```
(append '(x) '(y))          ⇒  (x y)
(append '(a) '(b c d))      ⇒  (a b c d)
(append '(a (b)) '((c)))    ⇒  (a (b) (c))
```

The last argument *lstN* may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))    ⇒  (a b c . d)
(append '() 'a)             ⇒  a
```

`append` doesn't modify the given lists, but the return may share structure with the final *lstN*. `append!` modifies the given lists to form its return.

For `scm_append` and `scm_append_x`, *lstlst* is a list of the list operands *lst1 ... lstN*. That *lstlst* itself is not modified or used in the return.

`reverse` *lst* [Scheme Procedure]
`reverse!` *lst* [*newtail*] [Scheme Procedure]

`scm_reverse` (*lst*) [C Function]
`scm_reverse_x` (*lst, newtail*) [C Function]

Return a list comprising the elements of *lst*, in reverse order.

`reverse` constructs a new list, `reverse!` modifies *lst* in constructing its return.

For `reverse!`, the optional *newtail* is appended to to the result. *newtail* isn't reversed, it simply becomes the list tail. For `scm_reverse_x`, the *newtail* parameter is mandatory, but can be `SCM_EOL` if no further tail is required.

5.6.2.6 List Modification

The following procedures modify an existing list, either by changing elements of the list, or by changing the list structure itself.

`list-set!` *list k val* [Scheme Procedure]
`scm_list_set_x` (*list, k, val*) [C Function]

Set the *k*th element of *list* to *val*.

`list-cdr-set!` *list k val* [Scheme Procedure]
`scm_list_cdr_set_x` (*list, k, val*) [C Function]

Set the *k*th cdr of *list* to *val*.

`delq` *item lst* [Scheme Procedure]
`scm_delq` (*item, lst*) [C Function]

Return a newly-created copy of *lst* with elements `eq?` to *item* removed. This procedure mirrors `memq`: `delq` compares elements of *lst* against *item* with `eq?`.

`delv` *item lst* [Scheme Procedure]
`scm_delv` (*item, lst*) [C Function]

Return a newly-created copy of *lst* with elements `eqv?` to *item* removed. This procedure mirrors `memv`: `delv` compares elements of *lst* against *item* with `eqv?`.

`delete` *item lst* [Scheme Procedure]
`scm_delete` (*item, lst*) [C Function]

Return a newly-created copy of *lst* with elements `equal?` to *item* removed. This procedure mirrors `member`: `delete` compares elements of *lst* against *item* with `equal?`.

See also SRFI-1 which has an extended `delete` (Section 6.4.3.8 [SRFI-1 Deleting], page 433), and also an `lset-difference` which can delete multiple *item*s in one call (Section 6.4.3.10 [SRFI-1 Set Operations], page 435).

`delq!` *item lst* [Scheme Procedure]
`delv!` *item lst* [Scheme Procedure]
`delete!` *item lst* [Scheme Procedure]
`scm_delq_x` (*item, lst*) [C Function]
`scm_delv_x` (*item, lst*) [C Function]
`scm_delete_x` (*item, lst*) [C Function]

These procedures are destructive versions of `delq`, `delv` and `delete`: they modify the pointers in the existing *lst* rather than creating a new list. Caveat evaluator: Like other destructive list functions, these functions cannot modify the binding of *lst*, and so cannot be used to delete the first element of *lst* destructively.

`delq1!` *item lst* [Scheme Procedure]
`scm_delq1_x` (*item*, *lst*) [C Function]

Like `delq!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eq?`. See also `delv1!` and `delete1!`.

`delv1!` *item lst* [Scheme Procedure]
`scm_delv1_x` (*item*, *lst*) [C Function]

Like `delv!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eqv?`. See also `delq1!` and `delete1!`.

`delete1!` *item lst* [Scheme Procedure]
`scm_delete1_x` (*item*, *lst*) [C Function]

Like `delete!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `equal?`. See also `delq1!` and `delv1!`.

`filter` *pred lst* [Scheme Procedure]
`filter!` *pred lst* [Scheme Procedure]

Return a list containing all elements from *lst* which satisfy the predicate *pred*. The elements in the result list have the same order as in *lst*. The order in which *pred* is applied to the list elements is not specified.

`filter` does not change *lst*, but the result may share a tail with it. `filter!` may modify *lst* to construct its return.

5.6.2.7 List Searching

The following procedures search lists for particular elements. They use different comparison predicates for comparing list elements with the object to be searched. When they fail, they return `#f`, otherwise they return the sublist whose car is equal to the search object, where equality depends on the equality predicate used.

`memq` *x lst* [Scheme Procedure]
`scm_memq` (*x*, *lst*) [C Function]

Return the first sublist of *lst* whose car is `eq?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

`memv` *x lst* [Scheme Procedure]
`scm_memv` (*x*, *lst*) [C Function]

Return the first sublist of *lst* whose car is `eqv?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

`member` *x lst* [Scheme Procedure]
`scm_member` (*x*, *lst*) [C Function]

Return the first sublist of *lst* whose car is `equal?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

See also SRFI-1 which has an extended `member` function (Section 6.4.3.7 [SRFI-1 Searching], page 432).

5.6.2.8 List Mapping

List processing is very convenient in Scheme because the process of iterating over the elements of a list can be highly abstracted. The procedures in this section are the most basic iterating procedures for lists. They take a procedure and one or more lists as arguments, and apply the procedure to each element of the list. They differ in their return value.

`map` *proc arg1 arg2 . . .* [Scheme Procedure]
`map-in-order` *proc arg1 arg2 . . .* [Scheme Procedure]
`scm_map` (*proc, arg1, args*) [C Function]

> Apply *proc* to each element of the list *arg1* (if only two arguments are given), or to the corresponding elements of the argument lists (if more than two arguments are given). The result(s) of the procedure applications are saved and returned in a list. For `map`, the order of procedure applications is not specified, `map-in-order` applies the procedure from left to right to the list elements.

`for-each` *proc arg1 arg2 . . .* [Scheme Procedure]

> Like `map`, but the procedure is always applied from left to right, and the result(s) of the procedure applications are thrown away. The return value is not specified.

See also SRFI-1 which extends these functions to take lists of unequal lengths (Section 6.4.3.5 [SRFI-1 Fold and Map], page 428).

5.6.3 Vectors

Vectors are sequences of Scheme objects. Unlike lists, the length of a vector, once the vector is created, cannot be changed. The advantage of vectors over lists is that the time required to access one element of a vector given its *position* (synonymous with *index*), a zero-origin number, is constant, whereas lists have an access time linear to the position of the accessed element in the list.

Vectors can contain any kind of Scheme object; it is even possible to have different types of objects in the same vector. For vectors containing vectors, you may wish to use arrays, instead. Note, too, that vectors are the special case of one dimensional non-uniform arrays and that most array procedures operate happily on vectors (see Section 5.6.7 [Arrays], page 191).

5.6.3.1 Read Syntax for Vectors

Vectors can literally be entered in source code, just like strings, characters or some of the other data types. The read syntax for vectors is as follows: A sharp sign (`#`), followed by an opening parentheses, all elements of the vector in their respective read syntax, and finally a closing parentheses. The following are examples of the read syntax for vectors; where the first vector only contains numbers and the second three different object types: a string, a symbol and a number in hexadecimal notation.

```
#(1 2 3)
#("Hello" foo #xdeadbeef)
```

Like lists, vectors have to be quoted:

```
'#(a b c) ⇒ #(a b c)
```

5.6.3.2 Dynamic Vector Creation and Validation

Instead of creating a vector implicitly by using the read syntax just described, you can create a vector dynamically by calling one of the `vector` and `list->vector` primitives with the list of Scheme values that you want to place into a vector. The size of the vector thus created is determined implicitly by the number of arguments given.

`vector` *. l* [Scheme Procedure]
`list->vector` *l* [Scheme Procedure]
`scm_vector` (*l*) [C Function]

Return a newly allocated vector composed of the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c) ⇒ #(a b c)
```

The inverse operation is `vector->list`:

`vector->list` *v* [Scheme Procedure]
`scm_vector_to_list` (*v*) [C Function]

Return a newly allocated list composed of the elements of *v*.

```
(vector->list '#(dah dah didah)) ⇒  (dah dah didah)
(list->vector '(dididit dah)) ⇒  #(dididit dah)
```

To allocate a vector with an explicitly specified size, use `make-vector`. With this primitive you can also specify an initial value for the vector elements (the same value for all elements, that is):

`make-vector` *len* [*fill*] [Scheme Procedure]
`scm_make_vector` (*len, fill*) [C Function]

Return a newly allocated vector of *len* elements. If a second argument is given, then each position is initialized to *fill*. Otherwise the initial contents of each position is unspecified.

`SCM scm_c_make_vector` (*size_t k, SCM fill*) [C Function]

Like `scm_make_vector`, but the length is given as a `size_t`.

To check whether an arbitrary Scheme value *is* a vector, use the `vector?` primitive:

`vector?` *obj* [Scheme Procedure]
`scm_vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a vector, otherwise return `#f`.

`int scm_is_vector` (*SCM obj*) [C Function]

Return non-zero when *obj* is a vector, otherwise return `zero`.

5.6.3.3 Accessing and Modifying Vector Contents

`vector-length` and `vector-ref` return information about a given vector, respectively its size and the elements that are contained in the vector.

`vector-length` *vector* [Scheme Procedure]
`scm_vector_length` *vector* [C Function]

Return the number of elements in *vector* as an exact integer.

`size_t scm_c_vector_length` (*SCM v*) [C Function]

Return the number of elements in *vector* as a `size_t`.

`vector-ref` *vector k* [Scheme Procedure]
`scm_vector_ref` *vector k* [C Function]

Return the contents of position *k* of *vector*. *k* must be a valid index of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5) ⇒ 8
(vector-ref '#(1 1 2 3 5 8 13 21)
    (let ((i (round (* 2 (acos -1)))))
      (if (inexact? i)
        (inexact->exact i)
           i))) ⇒ 13
```

`SCM scm_c_vector_ref` (*SCM v, size_t k*) [C Function]

Return the contents of position *k* (a `size_t`) of *vector*.

A vector created by one of the dynamic vector constructor procedures (see Section 5.6.3.2 [Vector Creation], page 175) can be modified using the following procedures.

NOTE: According to R5RS, it is an error to use any of these procedures on a literally read vector, because such vectors should be considered as constants. Currently, however, Guile does not detect this error.

`vector-set!` *vector k obj* [Scheme Procedure]
`scm_vector_set_x` *vector k obj* [C Function]

Store *obj* in position *k* of *vector*. *k* must be a valid index of *vector*. The value returned by '`vector-set!`' is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec) ⇒  #(0 ("Sue" "Sue") "Anna")
```

`void scm_c_vector_set_x` (*SCM v, size_t k, SCM obj*) [C Function]

Store *obj* in position *k* (a `size_t`) of *v*.

`vector-fill!` *v fill* [Scheme Procedure]
`scm_vector_fill_x` (*v, fill*) [C Function]

Store *fill* in every position of *vector*. The value returned by `vector-fill!` is unspecified.

`vector-copy` *vec* [Scheme Procedure]
`scm_vector_copy` (*vec*) [C Function]

Return a copy of *vec*.

`vector-move-left!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_left_x` (*vec1, start1, end1, vec2, start2*) [C Function]

Copy elements from *vec1*, positions *start1* to *end1*, to *vec2* starting at position *start2*. *start1* and *start2* are inclusive indices; *end1* is exclusive.

`vector-move-left!` copies elements in leftmost order. Therefore, in the case where *vec1* and *vec2* refer to the same vector, `vector-move-left!` is usually appropriate when *start1* is greater than *start2*.

`vector-move-right!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_right_x` (*vec1, start1, end1, vec2, start2*) [C Function]

Copy elements from *vec1*, positions *start1* to *end1*, to *vec2* starting at position *start2*. *start1* and *start2* are inclusive indices; *end1* is exclusive.

`vector-move-right!` copies elements in rightmost order. Therefore, in the case where *vec1* and *vec2* refer to the same vector, `vector-move-right!` is usually appropriate when *start1* is less than *start2*.

5.6.3.4 Vector Accessing from C

A vector can be read and modified from C with the functions `scm_c_vector_ref` and `scm_c_vector_set_x`, for example. In addition to these functions, there are two more ways to access vectors from C that might be more efficient in certain situations: you can restrict yourself to *simple vectors* and then use the very fast *simple vector macros*; or you can use the very general framework for accessing all kinds of arrays (see Section 5.6.7.4 [Accessing Arrays from C], page 198), which is more verbose, but can deal efficiently with all kinds of vectors (and arrays). For vectors, you can use the `scm_vector_elements` and `scm_vector_writable_elements` functions as shortcuts.

`int scm_is_simple_vector` (*SCM obj*) [C Function]

Return non-zero if *obj* is a simple vector, else return zero. A simple vector is a vector that can be used with the `SCM_SIMPLE_*` macros below.

The following functions are guaranteed to return simple vectors: `scm_make_vector`, `scm_c_make_vector`, `scm_vector`, `scm_list_to_vector`.

`size_t SCM_SIMPLE_VECTOR_LENGTH` (*SCM vec*) [C Macro]

Evaluates to the length of the simple vector *vec*. No type checking is done.

`SCM SCM_SIMPLE_VECTOR_REF` (*SCM vec, size_t idx*) [C Macro]

Evaluates to the element at position *idx* in the simple vector *vec*. No type or range checking is done.

`void SCM_SIMPLE_VECTOR_SET` (*SCM vec, size_t idx, SCM val*) [C Macro]

Sets the element at position *idx* in the simple vector *vec* to *val*. No type or range checking is done.

`const SCM * scm_vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Acquirea handle for the vector *vec* and return a pointer to the elements of it. This pointer can only be used to read the elements of *vec*. When *vec* is not a vector, an error is signaled. The handle mustr eventually be released with `scm_array_handle_release`.

The variables pointed to by *lenp* and *incp* are filled with the number of elements of the vector and the increment (number of elements) between successive elements, respectively. Successive elements of *vec* need not be contiguous in their underlying "root vector" returned here; hence the increment is not necessarily equal to 1 and may well be negative too (see Section 5.6.7.3 [Shared Arrays], page 196).

The following example shows the typical way to use this function. It creates a list of all elements of *vec* (in reverse order).

```
          scm_t_array_handle handle;
          size_t i, len;
          ssize_t inc;
          const SCM *elt;
          SCM list;

          elt = scm_vector_elements (vec, &handle, &len, &inc);
          list = SCM_EOL;
          for (i = 0; i < len; i++, elt += inc)
            list = scm_cons (*elt, list);
          scm_array_handle_release (&handle);
```

`SCM * scm_vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` but the pointer can be used to modify the vector.

The following example shows the typical way to use this function. It fills a vector with `#t`.

```
          scm_t_array_handle handle;
          size_t i, len;
          ssize_t inc;
          SCM *elt;

          elt = scm_vector_writable_elements (vec, &handle, &len, &inc);
          for (i = 0; i < len; i++, elt += inc)
            *elt = SCM_BOOL_T;
          scm_array_handle_release (&handle);
```

5.6.4 Uniform Numeric Vectors

A uniform numeric vector is a vector whose elements are all of a single numeric type. Guile offers uniform numeric vectors for signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers, two sizes of floating point values, and complex floating-point numbers of these two sizes.

Strings could be regarded as uniform vectors of characters, See Section 5.5.5 [Strings], page 129. Likewise, bit vectors could be regarded as uniform vectors of bits, See Section 5.6.5 [Bit Vectors], page 187. Both are sufficiently different from uniform numeric vectors that the procedures described here do not apply to these two data types. However, both strings and bit vectors are generalized vectors, See Section 5.6.6 [Generalized Vectors], page 190, and arrays, See Section 5.6.7 [Arrays], page 191.

Uniform numeric vectors are the special case of one dimensional uniform numeric arrays.

Uniform numeric vectors can be useful since they consume less memory than the non-uniform, general vectors. Also, since the types they can store correspond directly to C types, it is easier to work with them efficiently on a low level. Consider image processing as an example, where you want to apply a filter to some image. While you could store the pixels of an image in a general vector and write a general convolution function, things are much more efficient with uniform vectors: the convolution function knows that all pixels are unsigned 8-bit values (say), and can use a very tight inner loop.

That is, when it is written in C. Functions for efficiently working with uniform numeric vectors from C are listed at the end of this section.

Procedures similar to the vector procedures (see Section 5.6.3 [Vectors], page 174) are provided for handling these uniform vectors, but they are distinct datatypes and the two cannot be inter-mixed. If you want to work primarily with uniform numeric vectors, but want to offer support for general vectors as a convenience, you can use one of the `scm_any_to_*` functions. They will coerce lists and vectors to the given type of uniform vector. Alternatively, you can write two versions of your code: one that is fast and works only with uniform numeric vectors, and one that works with any kind of vector but is slower.

One set of the procedures listed below is a generic one: it works with all types of uniform numeric vectors. In addition to that, there is a set of procedures for each type that only works with that type. Unless you really need to the generality of the first set, it is best to use the more specific functions. They might not be that much faster, but their use can serve as a kind of declaration and makes it easier to optimize later on.

The generic set of procedures uses `uniform` in its names, the specific ones use the tag from the following table.

`u8`	unsigned 8-bit integers
`s8`	signed 8-bit integers
`u16`	unsigned 16-bit integers
`s16`	signed 16-bit integers
`u32`	unsigned 32-bit integers
`s32`	signed 32-bit integers
`u64`	unsigned 64-bit integers
`s64`	signed 64-bit integers
`f32`	the C type `float`
`f64`	the C type `double`
`c32`	complex numbers in rectangular form with the real and imaginary part being a `float`
`c64`	complex numbers in rectangular form with the real and imaginary part being a `double`

The external representation (ie. read syntax) for these vectors is similar to normal Scheme vectors, but with an additional tag from the table above indiciating the vector's type. For example,

```
#u16(1 2 3)
#f64(3.1415 2.71)
```

Note that the read syntax for floating-point here conflicts with `#f` for false. In Standard Scheme one can write `(1 #f3)` for a three element list `(1 #f 3)`, but for Guile `(1 #f3)` is invalid. `(1 #f 3)` is almost certainly what one should write anyway to make the intention clear, so this is rarely a problem.

`uniform-vector?` *obj* [Scheme Procedure]
`u8vector?` *obj* [Scheme Procedure]
`s8vector?` *obj* [Scheme Procedure]
`u16vector?` *obj* [Scheme Procedure]
`s16vector?` *obj* [Scheme Procedure]
`u32vector?` *obj* [Scheme Procedure]
`s32vector?` *obj* [Scheme Procedure]
`u64vector?` *obj* [Scheme Procedure]
`s64vector?` *obj* [Scheme Procedure]
`f32vector?` *obj* [Scheme Procedure]
`f64vector?` *obj* [Scheme Procedure]
`c32vector?` *obj* [Scheme Procedure]
`c64vector?` *obj* [Scheme Procedure]
`scm_uniform_vector_p` (*obj*) [C Function]
`scm_u8vector_p` (*obj*) [C Function]
`scm_s8vector_p` (*obj*) [C Function]
`scm_u16vector_p` (*obj*) [C Function]
`scm_s16vector_p` (*obj*) [C Function]
`scm_u32vector_p` (*obj*) [C Function]
`scm_s32vector_p` (*obj*) [C Function]
`scm_u64vector_p` (*obj*) [C Function]
`scm_s64vector_p` (*obj*) [C Function]
`scm_f32vector_p` (*obj*) [C Function]
`scm_f64vector_p` (*obj*) [C Function]
`scm_c32vector_p` (*obj*) [C Function]
`scm_c64vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a homogeneous numeric vector of the indicated type.

`make-u8vector` *n* [*value*] [Scheme Procedure]
`make-s8vector` *n* [*value*] [Scheme Procedure]
`make-u16vector` *n* [*value*] [Scheme Procedure]
`make-s16vector` *n* [*value*] [Scheme Procedure]
`make-u32vector` *n* [*value*] [Scheme Procedure]
`make-s32vector` *n* [*value*] [Scheme Procedure]
`make-u64vector` *n* [*value*] [Scheme Procedure]
`make-s64vector` *n* [*value*] [Scheme Procedure]
`make-f32vector` *n* [*value*] [Scheme Procedure]
`make-f64vector` *n* [*value*] [Scheme Procedure]
`make-c32vector` *n* [*value*] [Scheme Procedure]
`make-c64vector` *n* [*value*] [Scheme Procedure]
`scm_make_u8vector` *n* [*value*] [C Function]
`scm_make_s8vector` *n* [*value*] [C Function]
`scm_make_u16vector` *n* [*value*] [C Function]
`scm_make_s16vector` *n* [*value*] [C Function]
`scm_make_u32vector` *n* [*value*] [C Function]
`scm_make_s32vector` *n* [*value*] [C Function]
`scm_make_u64vector` *n* [*value*] [C Function]

`scm_make_s64vector` *n* [*value*] [C Function]
`scm_make_f32vector` *n* [*value*] [C Function]
`scm_make_f64vector` *n* [*value*] [C Function]
`scm_make_c32vector` *n* [*value*] [C Function]
`scm_make_c64vector` *n* [*value*] [C Function]

Return a newly allocated homogeneous numeric vector holding *n* elements of the indicated type. If *value* is given, the vector is initialized with that value, otherwise the contents are unspecified.

`u8vector` *value* ... [Scheme Procedure]
`s8vector` *value* ... [Scheme Procedure]
`u16vector` *value* ... [Scheme Procedure]
`s16vector` *value* ... [Scheme Procedure]
`u32vector` *value* ... [Scheme Procedure]
`s32vector` *value* ... [Scheme Procedure]
`u64vector` *value* ... [Scheme Procedure]
`s64vector` *value* ... [Scheme Procedure]
`f32vector` *value* ... [Scheme Procedure]
`f64vector` *value* ... [Scheme Procedure]
`c32vector` *value* ... [Scheme Procedure]
`c64vector` *value* ... [Scheme Procedure]
`scm_u8vector` (*values*) [C Function]
`scm_s8vector` (*values*) [C Function]
`scm_u16vector` (*values*) [C Function]
`scm_s16vector` (*values*) [C Function]
`scm_u32vector` (*values*) [C Function]
`scm_s32vector` (*values*) [C Function]
`scm_u64vector` (*values*) [C Function]
`scm_s64vector` (*values*) [C Function]
`scm_f32vector` (*values*) [C Function]
`scm_f64vector` (*values*) [C Function]
`scm_c32vector` (*values*) [C Function]
`scm_c64vector` (*values*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, holding the given parameter *values*. The vector length is the number of parameters given.

`uniform-vector-length` *vec* [Scheme Procedure]
`u8vector-length` *vec* [Scheme Procedure]
`s8vector-length` *vec* [Scheme Procedure]
`u16vector-length` *vec* [Scheme Procedure]
`s16vector-length` *vec* [Scheme Procedure]
`u32vector-length` *vec* [Scheme Procedure]
`s32vector-length` *vec* [Scheme Procedure]
`u64vector-length` *vec* [Scheme Procedure]
`s64vector-length` *vec* [Scheme Procedure]
`f32vector-length` *vec* [Scheme Procedure]
`f64vector-length` *vec* [Scheme Procedure]
`c32vector-length` *vec* [Scheme Procedure]

c64vector-length *vec* [Scheme Procedure]
scm_uniform_vector_length (*vec*) [C Function]
scm_u8vector_length (*vec*) [C Function]
scm_s8vector_length (*vec*) [C Function]
scm_u16vector_length (*vec*) [C Function]
scm_s16vector_length (*vec*) [C Function]
scm_u32vector_length (*vec*) [C Function]
scm_s32vector_length (*vec*) [C Function]
scm_u64vector_length (*vec*) [C Function]
scm_s64vector_length (*vec*) [C Function]
scm_f32vector_length (*vec*) [C Function]
scm_f64vector_length (*vec*) [C Function]
scm_c32vector_length (*vec*) [C Function]
scm_c64vector_length (*vec*) [C Function]

Return the number of elements in *vec*.

uniform-vector-ref *vec i* [Scheme Procedure]
u8vector-ref *vec i* [Scheme Procedure]
s8vector-ref *vec i* [Scheme Procedure]
u16vector-ref *vec i* [Scheme Procedure]
s16vector-ref *vec i* [Scheme Procedure]
u32vector-ref *vec i* [Scheme Procedure]
s32vector-ref *vec i* [Scheme Procedure]
u64vector-ref *vec i* [Scheme Procedure]
s64vector-ref *vec i* [Scheme Procedure]
f32vector-ref *vec i* [Scheme Procedure]
f64vector-ref *vec i* [Scheme Procedure]
c32vector-ref *vec i* [Scheme Procedure]
c64vector-ref *vec i* [Scheme Procedure]
scm_uniform_vector_ref (*vec i*) [C Function]
scm_u8vector_ref (*vec i*) [C Function]
scm_s8vector_ref (*vec i*) [C Function]
scm_u16vector_ref (*vec i*) [C Function]
scm_s16vector_ref (*vec i*) [C Function]
scm_u32vector_ref (*vec i*) [C Function]
scm_s32vector_ref (*vec i*) [C Function]
scm_u64vector_ref (*vec i*) [C Function]
scm_s64vector_ref (*vec i*) [C Function]
scm_f32vector_ref (*vec i*) [C Function]
scm_f64vector_ref (*vec i*) [C Function]
scm_c32vector_ref (*vec i*) [C Function]
scm_c64vector_ref (*vec i*) [C Function]

Return the element at index *i* in *vec*. The first element in *vec* is index 0.

uniform-vector-set! *vec i value* [Scheme Procedure]
u8vector-set! *vec i value* [Scheme Procedure]
s8vector-set! *vec i value* [Scheme Procedure]
u16vector-set! *vec i value* [Scheme Procedure]

`s16vector-set!` *vec i value* [Scheme Procedure]
`u32vector-set!` *vec i value* [Scheme Procedure]
`s32vector-set!` *vec i value* [Scheme Procedure]
`u64vector-set!` *vec i value* [Scheme Procedure]
`s64vector-set!` *vec i value* [Scheme Procedure]
`f32vector-set!` *vec i value* [Scheme Procedure]
`f64vector-set!` *vec i value* [Scheme Procedure]
`c32vector-set!` *vec i value* [Scheme Procedure]
`c64vector-set!` *vec i value* [Scheme Procedure]
`scm_uniform_vector_set_x` (*vec i value*) [C Function]
`scm_u8vector_set_x` (*vec i value*) [C Function]
`scm_s8vector_set_x` (*vec i value*) [C Function]
`scm_u16vector_set_x` (*vec i value*) [C Function]
`scm_s16vector_set_x` (*vec i value*) [C Function]
`scm_u32vector_set_x` (*vec i value*) [C Function]
`scm_s32vector_set_x` (*vec i value*) [C Function]
`scm_u64vector_set_x` (*vec i value*) [C Function]
`scm_s64vector_set_x` (*vec i value*) [C Function]
`scm_f32vector_set_x` (*vec i value*) [C Function]
`scm_f64vector_set_x` (*vec i value*) [C Function]
`scm_c32vector_set_x` (*vec i value*) [C Function]
`scm_c64vector_set_x` (*vec i value*) [C Function]

Set the element at index *i* in *vec* to *value*. The first element in *vec* is index 0. The return value is unspecified.

`uniform-vector->list` *vec* [Scheme Procedure]
`u8vector->list` *vec* [Scheme Procedure]
`s8vector->list` *vec* [Scheme Procedure]
`u16vector->list` *vec* [Scheme Procedure]
`s16vector->list` *vec* [Scheme Procedure]
`u32vector->list` *vec* [Scheme Procedure]
`s32vector->list` *vec* [Scheme Procedure]
`u64vector->list` *vec* [Scheme Procedure]
`s64vector->list` *vec* [Scheme Procedure]
`f32vector->list` *vec* [Scheme Procedure]
`f64vector->list` *vec* [Scheme Procedure]
`c32vector->list` *vec* [Scheme Procedure]
`c64vector->list` *vec* [Scheme Procedure]
`scm_uniform_vector_to_list` (*vec*) [C Function]
`scm_u8vector_to_list` (*vec*) [C Function]
`scm_s8vector_to_list` (*vec*) [C Function]
`scm_u16vector_to_list` (*vec*) [C Function]
`scm_s16vector_to_list` (*vec*) [C Function]
`scm_u32vector_to_list` (*vec*) [C Function]
`scm_s32vector_to_list` (*vec*) [C Function]
`scm_u64vector_to_list` (*vec*) [C Function]
`scm_s64vector_to_list` (*vec*) [C Function]

`scm_f32vector_to_list` (*vec*) [C Function]
`scm_f64vector_to_list` (*vec*) [C Function]
`scm_c32vector_to_list` (*vec*) [C Function]
`scm_c64vector_to_list` (*vec*) [C Function]

Return a newly allocated list holding all elements of *vec*.

`list->u8vector` *lst* [Scheme Procedure]
`list->s8vector` *lst* [Scheme Procedure]
`list->u16vector` *lst* [Scheme Procedure]
`list->s16vector` *lst* [Scheme Procedure]
`list->u32vector` *lst* [Scheme Procedure]
`list->s32vector` *lst* [Scheme Procedure]
`list->u64vector` *lst* [Scheme Procedure]
`list->s64vector` *lst* [Scheme Procedure]
`list->f32vector` *lst* [Scheme Procedure]
`list->f64vector` *lst* [Scheme Procedure]
`list->c32vector` *lst* [Scheme Procedure]
`list->c64vector` *lst* [Scheme Procedure]
`scm_list_to_u8vector` (*lst*) [C Function]
`scm_list_to_s8vector` (*lst*) [C Function]
`scm_list_to_u16vector` (*lst*) [C Function]
`scm_list_to_s16vector` (*lst*) [C Function]
`scm_list_to_u32vector` (*lst*) [C Function]
`scm_list_to_s32vector` (*lst*) [C Function]
`scm_list_to_u64vector` (*lst*) [C Function]
`scm_list_to_s64vector` (*lst*) [C Function]
`scm_list_to_f32vector` (*lst*) [C Function]
`scm_list_to_f64vector` (*lst*) [C Function]
`scm_list_to_c32vector` (*lst*) [C Function]
`scm_list_to_c64vector` (*lst*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, initialized with the elements of the list *lst*.

`any->u8vector` *obj* [Scheme Procedure]
`any->s8vector` *obj* [Scheme Procedure]
`any->u16vector` *obj* [Scheme Procedure]
`any->s16vector` *obj* [Scheme Procedure]
`any->u32vector` *obj* [Scheme Procedure]
`any->s32vector` *obj* [Scheme Procedure]
`any->u64vector` *obj* [Scheme Procedure]
`any->s64vector` *obj* [Scheme Procedure]
`any->f32vector` *obj* [Scheme Procedure]
`any->f64vector` *obj* [Scheme Procedure]
`any->c32vector` *obj* [Scheme Procedure]
`any->c64vector` *obj* [Scheme Procedure]
`scm_any_to_u8vector` (*obj*) [C Function]
`scm_any_to_s8vector` (*obj*) [C Function]
`scm_any_to_u16vector` (*obj*) [C Function]

`scm_any_to_s16vector` (*obj*) [C Function]
`scm_any_to_u32vector` (*obj*) [C Function]
`scm_any_to_s32vector` (*obj*) [C Function]
`scm_any_to_u64vector` (*obj*) [C Function]
`scm_any_to_s64vector` (*obj*) [C Function]
`scm_any_to_f32vector` (*obj*) [C Function]
`scm_any_to_f64vector` (*obj*) [C Function]
`scm_any_to_c32vector` (*obj*) [C Function]
`scm_any_to_c64vector` (*obj*) [C Function]

Return a (maybe newly allocated) uniform numeric vector of the indicated type, initialized with the elements of *obj*, which must be a list, a vector, or a uniform vector. When *obj* is already a suitable uniform numeric vector, it is returned unchanged.

`int scm_is_uniform_vector` (*SCM uvec*) [C Function]

Return non-zero when *uvec* is a uniform numeric vector, zero otherwise.

`SCM scm_take_u8vector` (*const scm_t_uint8 *data, size_t len*) [C Function]
`SCM scm_take_s8vector` (*const scm_t_int8 *data, size_t len*) [C Function]
`SCM scm_take_u16vector` (*const scm_t_uint16 *data, size_t len*) [C Function]
`SCM scm_take_s168vector` (*const scm_t_int16 *data, size_t len*) [C Function]
`SCM scm_take_u32vector` (*const scm_t_uint32 *data, size_t len*) [C Function]
`SCM scm_take_s328vector` (*const scm_t_int32 *data, size_t len*) [C Function]
`SCM scm_take_u64vector` (*const scm_t_uint64 *data, size_t len*) [C Function]
`SCM scm_take_s64vector` (*const scm_t_int64 *data, size_t len*) [C Function]
`SCM scm_take_f32vector` (*const float *data, size_t len*) [C Function]
`SCM scm_take_f64vector` (*const double *data, size_t len*) [C Function]
`SCM scm_take_c32vector` (*const float *data, size_t len*) [C Function]
`SCM scm_take_c64vector` (*const double *data, size_t len*) [C Function]

Return a new uniform numeric vector of the indicated type and length that uses the memory pointed to by *data* to store its elements. This memory will eventually be freed with `free`. The argument *len* specifies the number of elements in *data*, not its size in bytes.

The `c32` and `c64` variants take a pointer to a C array of `float`s or `double`s. The real parts of the complex numbers are at even indices in that array, the corresponding imaginary parts are at the following odd index.

`size_t scm_c_uniform_vector_length` (*SCM uvec*) [C Function]

Return the number of elements of *uvec* as a `size_t`.

`const void * scm_uniform_vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`const scm_t_uint8 * scm_u8vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`const scm_t_int8 * scm_s8vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`const scm_t_uint16 * scm_u16vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const scm_t_int16 * scm_s16vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const scm_t_uint32 * scm_u32vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const scm_t_int32 * scm_s32vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const scm_t_uint64 * scm_u64vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const scm_t_int64 * scm_s64vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const float * scm_f23vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const double * scm_f64vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const float * scm_c32vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`const double * scm_c64vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` (see Section 5.6.3.4 [Vector Accessing from C], page 177), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

`void * scm_uniform_vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_uint8 * scm_u8vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_int8 * scm_s8vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_uint16 * scm_u16vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_int16 * scm_s16vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_uint32 * scm_u32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_int32 * scm_s32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_uint64 * scm_u64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`scm_t_int64 * scm_s64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`float * scm_f23vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`double * scm_f64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`float * scm_c32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function] `double * scm_c64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_writable_elements` (see Section 5.6.3.4 [Vector Accessing from C], page 177), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

`uniform-vector-read!` *uvec* [*port_or_fd* [*start* [*end*]]] [Scheme Procedure]
`scm_uniform_vector_read_x` (*uvec, port_or_fd, start, end*) [C Function]

Fill the elements of *uvec* by reading raw bytes from *port-or-fdes*, using host byte order.

The optional arguments *start* (inclusive) and *end* (exclusive) allow a specified region to be read, leaving the remainder of the vector unchanged.

When *port-or-fdes* is a port, all specified elements of *uvec* are attempted to be read, potentially blocking while waiting formore input or end-of-file. When *port-or-fd* is an integer, a single call to read(2) is made.

An error is signalled when the last element has only been partially filled before reaching end-of-file or in the single call to read(2).

`uniform-vector-read!` returns the number of elements read.

port-or-fdes may be omitted, in which case it defaults to the value returned by `(current-input-port)`.

`uniform-vector-write` *uvec* [*port_or_fd* [*start* [*end*]]] [Scheme Procedure]
`scm_uniform_vector_write` (*uvec, port_or_fd, start, end*) [C Function]

Write the elements of *uvec* as raw bytes to *port-or-fdes*, in the host byte order.

The optional arguments *start* (inclusive) and *end* (exclusive) allow a specified region to be written.

When *port-or-fdes* is a port, all specified elements of *uvec* are attempted to be written, potentially blocking while waiting for more room. When *port-or-fd* is an integer, a single call to write(2) is made.

An error is signalled when the last element has only been partially written in the single call to write(2).

The number of objects actually written is returned. *port-or-fdes* may be omitted, in which case it defaults to the value returned by `(current-output-port)`.

5.6.5 Bit Vectors

Bit vectors are zero-origin, one-dimensional arrays of booleans. They are displayed as a sequence of `0`s and `1`s prefixed by `#*`, e.g.,

```
(make-bitvector 8 #f) ⇒
#*00000000
```

Bit vectors are are also generalized vectors, See Section 5.6.6 [Generalized Vectors], page 190, and can thus be used with the array procedures, See Section 5.6.7 [Arrays], page 191. Bit vectors are the special case of one dimensional bit arrays.

`bitvector?` *obj* [Scheme Procedure]
`scm_bitvector_p` (*obj*) [C Function]

Return `#t` when *obj* is a bitvector, else return `#f`.

`int scm_is_bitvector` (*SCM obj*) [C Function]

Return `1` when *obj* is a bitvector, else return `0`.

`make-bitvector` *len* [*fill*] [Scheme Procedure]
`scm_make_bitvector` (*len, fill*) [C Function]

Create a new bitvector of length *len* and optionally initialize all elements to *fill*.

`SCM scm_c_make_bitvector` (*size_t len, SCM fill*) [C Function]

Like `scm_make_bitvector`, but the length is given as a `size_t`.

`bitvector` *. bits* [Scheme Procedure]
`scm_bitvector` (*bits*) [C Function]

Create a new bitvector with the arguments as elements.

`bitvector-length` *vec* [Scheme Procedure]
`scm_bitvector_length` (*vec*) [C Function]

Return the length of the bitvector *vec*.

`size_t scm_c_bitvector_length` (*SCM vec*) [C Function]

Like `scm_bitvector_length`, but the length is returned as a `size_t`.

`bitvector-ref` *vec idx* [Scheme Procedure]
`scm_bitvector_ref` (*vec, idx*) [C Function]

Return the element at index *idx* of the bitvector *vec*.

`SCM scm_c_bitvector_ref` (*SCM obj, size_t idx*) [C Function]

Return the element at index *idx* of the bitvector *vec*.

`bitvector-set!` *vec idx val* [Scheme Procedure]
`scm_bitvector_set_x` (*vec, idx, val*) [C Function]

Set the element at index *idx* of the bitvector *vec* when *val* is true, else clear it.

`SCM scm_c_bitvector_set_x` (*SCM obj, size_t idx, SCM val*) [C Function]

Set the element at index *idx* of the bitvector *vec* when *val* is true, else clear it.

`bitvector-fill!` *vec val* [Scheme Procedure]
`scm_bitvector_fill_x` (*vec, val*) [C Function]

Set all elements of the bitvector *vec* when *val* is true, else clear them.

`list->bitvector` *list* [Scheme Procedure]
`scm_list_to_bitvector` (*list*) [C Function]

Return a new bitvector initialized with the elements of *list*.

`bitvector->list` *vec* [Scheme Procedure]
`scm_bitvector_to_list` (*vec*) [C Function]

Return a new list initialized with the elements of the bitvector *vec*.

`bit-count` *bool bitvector* [Scheme Procedure]
`scm_bit_count` (*bool, bitvector*) [C Function]

Return a count of how many entries in *bitvector* are equal to *bool*. For example,

```
(bit-count #f #*000111000)  ⇒ 6
```

`bit-position` *bool bitvector start* [Scheme Procedure]
`scm_bit_position` (*bool, bitvector, start*) [C Function]

Return the index of the first occurrance of *bool* in *bitvector*, starting from *start*. If there is no *bool* entry between *start* and the end of *bitvector*, then return `#f`. For example,

```
(bit-position #t #*000101 0)  ⇒ 3
(bit-position #f #*0001111 3) ⇒ #f
```

`bit-invert!` *bitvector* [Scheme Procedure]
`scm_bit_invert_x` (*bitvector*) [C Function]

Modify *bitvector* by replacing each element with its negation.

`bit-set*!` *bitvector uvec bool* [Scheme Procedure]
`scm_bit_set_star_x` (*bitvector, uvec, bool*) [C Function]

Set entries of *bitvector* to *bool*, with *uvec* selecting the entries to change. The return value is unspecified.

If *uvec* is a bit vector, then those entries where it has `#t` are the ones in *bitvector* which are set to *bool*. *uvec* and *bitvector* must be the same length. When *bool* is `#t` it's like *uvec* is OR'ed into *bitvector*. Or when *bool* is `#f` it can be seen as an ANDNOT.

```
(define bv #*01000010)
(bit-set*! bv #*10010001 #t)
bv
⇒ #*11010011
```

If *uvec* is a uniform vector of unsigned long integers, then they're indexes into *bitvector* which are set to *bool*.

```
(define bv #*01000010)
(bit-set*! bv #u(5 2 7) #t)
bv
⇒ #*01100111
```

`bit-count*` *bitvector uvec bool* [Scheme Procedure]
`scm_bit_count_star` (*bitvector, uvec, bool*) [C Function]

Return a count of how many entries in *bitvector* are equal to *bool*, with *uvec* selecting the entries to consider.

uvec is interpreted in the same way as for `bit-set*!` above. Namely, if *uvec* is a bit vector then entries which have `#t` there are considered in *bitvector*. Or if *uvec* is a uniform vector of unsigned long integers then it's the indexes in *bitvector* to consider.

For example,

```
(bit-count* #*01110111 #*11001101 #t) ⇒ 3
(bit-count* #*01110111 #u(7 0 4) #f)  ⇒ 2
```

`const scm_t_uint32 * scm_bitvector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` (see Section 5.6.3.4 [Vector Accessing from C], page 177), but for bitvectors. The variable pointed to by *offp* is set to the value returned by `scm_array_handle_bit_elements_offset`. See `scm_array_handle_bit_elements` for how to use the returned pointer and the offset.

`scm_t_uint32 * scm_bitvector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_bitvector_elements`, but the pointer is good for reading and writing.

5.6.6 Generalized Vectors

Guile has a number of data types that are generally vector-like: strings, uniform numeric vectors, bitvectors, and of course ordinary vectors of arbitrary Scheme values. These types are disjoint: a Scheme value belongs to at most one of the four types listed above.

If you want to gloss over this distinction and want to treat all four types with common code, you can use the procedures in this section. They work with the *generalized vector* type, which is the union of the four vector-like types.

`generalized-vector?` *obj* [Scheme Procedure]
`scm_generalized_vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a vector, string, bitvector, or uniform numeric vector.

`generalized-vector-length` *v* [Scheme Procedure]
`scm_generalized_vector_length` (*v*) [C Function]

Return the length of the generalized vector *v*.

`generalized-vector-ref` *v idx* [Scheme Procedure]
`scm_generalized_vector_ref` (*v, idx*) [C Function]

Return the element at index *idx* of the generalized vector *v*.

`generalized-vector-set!` *v idx val* [Scheme Procedure]
`scm_generalized_vector_set_x` (*v, idx, val*) [C Function]

Set the element at index *idx* of the generalized vector *v* to *val*.

`generalized-vector->list` *v* [Scheme Procedure]
`scm_generalized_vector_to_list` (*v*) [C Function]

Return a new list whose elements are the elements of the generalized vector *v*.

`int scm_is_generalized_vector` (*SCM obj*) [C Function]

Return `1` if *obj* is a vector, string, bitvector, or uniform numeric vector; else return `0`.

`size_t scm_c_generalized_vector_length` (*SCM v*) [C Function]

Return the length of the generalized vector *v*.

`SCM scm_c_generalized_vector_ref` (*SCM v, size_t idx*) [C Function]

Return the element at index *idx* of the generalized vector *v*.

`void scm_c_generalized_vector_set_x` (*SCM v, size_t idx, SCM val*) [C Function]

Set the element at index *idx* of the generalized vector *v* to *val*.

`void scm_generalized_vector_get_handle` (*SCM v, scm_t_array_handle *handle*) [C Function]

Like `scm_array_get_handle` but an error is signalled when *v* is not of rank one. You can use `scm_array_handle_ref` and `scm_array_handle_set` to read and write the elements of *v*, or you can use functions like `scm_array_handle_<foo>_elements` to deal with specific types of vectors.

5.6.7 Arrays

Arrays are a collection of cells organized into an arbitrary number of dimensions. Each cell can be accessed in constant time by supplying an index for each dimension.

In the current implementation, an array uses a generalized vector for the actual storage of its elements. Any kind of generalized vector will do, so you can have arrays of uniform numeric values, arrays of characters, arrays of bits, and of course, arrays of arbitrary Scheme values. For example, arrays with an underlying `c64vector` might be nice for digital signal processing, while arrays made from a `u8vector` might be used to hold gray-scale images.

The number of dimensions of an array is called its *rank*. Thus, a matrix is an array of rank 2, while a vector has rank 1. When accessing an array element, you have to specify one exact integer for each dimension. These integers are called the *indices* of the element. An array specifies the allowed range of indices for each dimension via an inclusive lower and upper bound. These bounds can well be negative, but the upper bound must be greater than or equal to the lower bound minus one. When all lower bounds of an array are zero, it is called a *zero-origin* array.

Arrays can be of rank 0, which could be interpreted as a scalar. Thus, a zero-rank array can store exactly one object and the list of indices of this element is the empty list.

Arrays contain zero elements when one of their dimensions has a zero length. These empty arrays maintain information about their shape: a matrix with zero columns and 3 rows is different from a matrix with 3 columns and zero rows, which again is different from a vector of length zero.

Generalized vectors, such as strings, uniform numeric vectors, bit vectors and ordinary vectors, are the special case of one dimensional arrays.

5.6.7.1 Array Syntax

An array is displayed as `#` followed by its rank, followed by a tag that describes the underlying vector, optionally followed by information about its shape, and finally followed by the cells, organized into dimensions using parentheses.

In more words, the array tag is of the form

```
#<rank><vectag><@lower><:len><@lower><:len>...
```

where `<rank>` is a positive integer in decimal giving the rank of the array. It is omitted when the rank is 1 and the array is non-shared and has zero-origin (see below). For shared arrays and for a non-zero origin, the rank is always printed even when it is 1 to dinstinguish them from ordinary vectors.

The `<vectag>` part is the tag for a uniform numeric vector, like `u8`, `s16`, etc, `b` for bitvectors, or `a` for strings. It is empty for ordinary vectors.

The `<@lower>` part is a '`@`' character followed by a signed integer in decimal giving the lower bound of a dimension. There is one `<@lower>` for each dimension. When all lower bounds are zero, all `<@lower>` parts are omitted.

The `<:len>` part is a '`:`' character followed by an unsigned integer in decimal giving the length of a dimension. Like for the lower bounds, there is one `<:len>` for each dimension, and the `<:len>` part always follows the `<@lower>` part for a dimension. Lengths are only then printed when they can't be deduced from the nested lists of elements of the array literal, which can happen when at least one length is zero.

As a special case, an array of rank 0 is printed as `#0<vectag>(<scalar>)`, where `<scalar>` is the result of printing the single element of the array.

Thus,

`#(1 2 3)` is an ordinary array of rank 1 with lower bound 0 in dimension 0. (I.e., a regular vector.)

`#@2(1 2 3)`
is an ordinary array of rank 1 with lower bound 2 in dimension 0.

`#2((1 2 3) (4 5 6))`
is a non-uniform array of rank 2; a 3×3 matrix with index ranges 0..2 and 0..2.

`#u32(0 1 2)`
is a uniform u8 array of rank 1.

`#2u32@2@3((1 2) (2 3))`
is a uniform u8 array of rank 2 with index ranges 2..3 and 3..4.

`#2()` is a two-dimensional array with index ranges 0..-1 and 0..-1, i.e. both dimensions have length zero.

`#2:0:2()` is a two-dimensional array with index ranges 0..-1 and 0..1, i.e. the first dimension has length zero, but the second has length 2.

`#0(12)` is a rank-zero array with contents 12.

5.6.7.2 Array Procedures

When an array is created, the range of each dimension must be specified, e.g., to create a 2×3 array with a zero-based index:

```
(make-array 'ho 2 3) ⇒ #2((ho ho ho) (ho ho ho))
```

The range of each dimension can also be given explicitly, e.g., another way to create the same array:

```
(make-array 'ho '(0 1) '(0 2)) ⇒ #2((ho ho ho) (ho ho ho))
```

The following procedures can be used with arrays (or vectors). An argument shown as *idx*... means one parameter for each dimension in the array. A *idxlist* argument means a list of such values, one for each dimension.

`array?` *obj* [Scheme Procedure]
`scm_array_p` (*obj*, *unused*) [C Function]

Return `#t` if the *obj* is an array, and `#f` if not.

The second argument to scm_array_p is there for historical reasons, but it is not used. You should always pass `SCM_UNDEFINED` as its value.

`typed-array?` *obj type* [Scheme Procedure]
`scm_typed_array_p` (*obj, type*) [C Function]

Return `#t` if the *obj* is an array of type *type*, and `#f` if not.

`int scm_is_array` (*SCM obj*) [C Function]

Return `1` if the *obj* is an array and `0` if not.

`int scm_is_typed_array` (*SCM obj, SCM type*) [C Function]

Return `0` if the *obj* is an array of type *type*, and `1` if not.

`make-array` *fill bound . . .* [Scheme Procedure]
`scm_make_array` (*fill, bounds*) [C Function]

Equivalent to `(make-typed-array #t` ***`fill bound`*** `...)`.

`make-typed-array` *type fill bound . . .* [Scheme Procedure]
`scm_make_typed_array` (*type, fill, bounds*) [C Function]

Create and return an array that has as many dimensions as there are *bound*s and (maybe) fill it with *fill*.

The underlaying storage vector is created according to *type*, which must be a symbol whose name is the 'vectag' of the array as explained above, or `#t` for ordinary, non-specialized arrays.

For example, using the symbol `f64` for *type* will create an array that uses a `f64vector` for storing its elements, and `a` will use a string.

When *fill* is not the special *unspecified* value, the new array is filled with *fill*. Otherwise, the initial contents of the array is unspecified. The special *unspecified* value is stored in the variable `*unspecified*` so that for example `(make-typed-array 'u32 *unspecified* 4)` creates a uninitialized `u32` vector of length 4.

Each *bound* may be a positive non-zero integer *N*, in which case the index for that dimension can range from 0 through *N-1*; or an explicit index range specifier in the form `(LOWER UPPER)`, where both *lower* and *upper* are integers, possibly less than zero, and possibly the same number (however, *lower* cannot be greater than *upper*).

`list->array` *dimspec list* [Scheme Procedure]

Equivalent to `(list->typed-array #t` ***`dimspec list`***`)`.

`list->typed-array` *type dimspec list* [Scheme Procedure]
`scm_list_to_typed_array` (*type, dimspec, list*) [C Function]

Return an array of the type indicated by *type* with elements the same as those of *list*.

The argument *dimspec* determines the number of dimensions of the array and their lower bounds. When *dimspec* is an exact integer, it gives the number of dimensions directly and all lower bounds are zero. When it is a list of exact integers, then each element is the lower index bound of a dimension, and there will be as many dimensions as elements in the list.

`array-type` *array* [Scheme Procedure]

Return the type of *array*. This is the 'vectag' used for printing *array* (or `#t` for ordinary arrays) and can be used with `make-typed-array` to create an array of the same kind as *array*.

`array-ref` *array idx . . .* [Scheme Procedure]

Return the element at `(idx ...)` in *array*.

```
(define a (make-array 999 '(1 2) '(3 4)))
(array-ref a 2 4) ⇒ 999
```

`array-in-bounds?` *array idx . . .* [Scheme Procedure]
`scm_array_in_bounds_p` *(array, idxlist)* [C Function]

Return `#t` if the given index would be acceptable to `array-ref`.

```
(define a (make-array #f '(1 2) '(3 4)))
(array-in-bounds? a 2 3) ⇒ #t
(array-in-bounds? a 0 0) ⇒ #f
```

`array-set!` *array obj idx . . .* [Scheme Procedure]
`scm_array_set_x` *(array, obj, idxlist)* [C Function]

Set the element at `(idx ...)` in *array* to *obj*. The return value is unspecified.

```
(define a (make-array #f '(0 1) '(0 1)))
(array-set! a #t 1 1)
a ⇒ #2((#f #f) (#f #t))
```

`enclose-array` *array dim1 . . .* [Scheme Procedure]
`scm_enclose_array` *(array, dimlist)* [C Function]

dim1, *dim2* . . . should be nonnegative integers less than the rank of *array*. `enclose-array` returns an array resembling an array of shared arrays. The dimensions of each shared array are the same as the *dim*th dimensions of the original array, the dimensions of the outer array are the same as those of the original array that did not match a *dim*.

An enclosed array is not a general Scheme array. Its elements may not be set using `array-set!`. Two references to the same element of an enclosed array will be `equal?` but will not in general be `eq?`. The value returned by `array-prototype` when given an enclosed array is unspecified.

For example,

```
(enclose-array '#3(((a b c) (d e f)) ((1 2 3) (4 5 6))) 1)
⇒
#<enclosed-array (#1(a d) #1(b e) #1(c f)) (#1(1 4) #1(2 5) #1(3 6))>

(enclose-array '#3(((a b c) (d e f)) ((1 2 3) (4 5 6))) 1 0)
⇒
#<enclosed-array #2((a 1) (d 4)) #2((b 2) (e 5)) #2((c 3) (f 6))>
```

`array-shape` *array* [Scheme Procedure]
`array-dimensions` *array* [Scheme Procedure]
`scm_array_dimensions` *(array)* [C Function]

Return a list of the bounds for each dimenson of *array*.

`array-shape` gives `(lower upper)` for each dimension. `array-dimensions` instead returns just *upper* $+ 1$ for dimensions with a 0 lower bound. Both are suitable as input to `make-array`.

For example,

```
(define a (make-array 'foo '(-1 3) 5))
(array-shape a)      ⇒ ((-1 3) (0 4))
(array-dimensions a) ⇒ ((-1 3) 5)
```

`array-rank` *obj* [Scheme Procedure]
`scm_array_rank` (*obj*) [C Function]
Return the rank of *array*.

`size_t scm_c_array_rank` (*SCM array*) [C Function]
Return the rank of *array* as a `size_t`.

`array->list` *array* [Scheme Procedure]
`scm_array_to_list` (*array*) [C Function]
Return a list consisting of all the elements, in order, of *array*.

`array-copy!` *src dst* [Scheme Procedure]
`array-copy-in-order!` *src dst* [Scheme Procedure]
`scm_array_copy_x` (*src, dst*) [C Function]
Copy every element from vector or array *src* to the corresponding element of *dst*. *dst* must have the same rank as *src*, and be at least as large in each dimension. The return value is unspecified.

`array-fill!` *array fill* [Scheme Procedure]
`scm_array_fill_x` (*array, fill*) [C Function]
Store *fill* in every element of *array*. The value returned is unspecified.

`array-equal?` *array1 array2 . . .* [Scheme Procedure]
Return `#t` if all arguments are arrays with the same shape, the same type, and have corresponding elements which are either `equal?` or `array-equal?`. This function differs from `equal?` (see Section 5.9.1 [Equality], page 236) in that a one dimensional shared array may be `array-equal?` but not `equal?` to a vector or uniform vector.

`array-map!` *dst proc src1 . . . srcN* [Scheme Procedure]
`array-map-in-order!` *dst proc src1 . . . srcN* [Scheme Procedure]
`scm_array_map_x` (*dst, proc, srclist*) [C Function]
Set each element of the *dst* array to values obtained from calls to *proc*. The value returned is unspecified.

Each call is `(`*`proc elem1 . . . elemN`*`)`, where each *elem* is from the corresponding *src* array, at the *dst* index. `array-map-in-order!` makes the calls in row-major order, `array-map!` makes them in an unspecified order.

The *src* arrays must have the same number of dimensions as *dst*, and must have a range for each dimension which covers the range in *dst*. This ensures all *dst* indices are valid in each *src*.

`array-for-each` *proc src1 . . . srcN* [Scheme Procedure]
`scm_array_for_each` (*proc, src1, srclist*) [C Function]
Apply *proc* to each tuple of elements of *src1 . . . srcN*, in row-major order. The value returned is unspecified.

`array-index-map!` *dst proc* [Scheme Procedure]
`scm_array_index_map_x` (*dst, proc*) [C Function]

Set each element of the *dst* array to values returned by calls to *proc*. The value returned is unspecified.

Each call is `(`***proc i1 ... iN***`)`, where *i1 . . . iN* is the destination index, one parameter for each dimension. The order in which the calls are made is unspecified.

For example, to create a 4×4 matrix representing a cyclic group,

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

```
(define a (make-array #f 4 4))
(array-index-map! a (lambda (i j)
                      (modulo (+ i j) 4)))
```

`uniform-array-read!` *ra* [*port_or_fd* [*start* [*end*]]] [Scheme Procedure]
`scm_uniform_array_read_x` (*ra, port_or_fd, start, end*) [C Function]

Attempt to read all elements of *ura*, in lexicographic order, as binary objects from *port-or-fdes*. If an end of file is encountered, the objects up to that point are put into *ura* (starting at the beginning) and the remainder of the array is unchanged.

The optional arguments *start* and *end* allow a specified region of a vector (or linearized array) to be read, leaving the remainder of the vector unchanged.

`uniform-array-read!` returns the number of objects read. *port-or-fdes* may be omitted, in which case it defaults to the value returned by `(current-input-port)`.

`uniform-array-write` *v* [*port_or_fd* [*start* [*end*]]] [Scheme Procedure]
`scm_uniform_array_write` (*v, port_or_fd, start, end*) [C Function]

Writes all elements of *ura* as binary objects to *port-or-fdes*.

The optional arguments *start* and *end* allow a specified region of a vector (or linearized array) to be written.

The number of objects actually written is returned. *port-or-fdes* may be omitted, in which case it defaults to the value returned by `(current-output-port)`.

5.6.7.3 Shared Arrays

`make-shared-array` *oldarray mapfunc bound . . .* [Scheme Procedure]
`scm_make_shared_array` (*oldarray, mapfunc, boundlist*) [C Function]

Return a new array which shares the storage of *oldarray*. Changes made through either affect the same underlying storage. The *bound. . .* arguments are the shape of the new array, the same as `make-array` (see Section 5.6.7.2 [Array Procedures], page 192).

mapfunc translates coordinates from the new array to the *oldarray*. It's called as `(`***mapfunc***` newidx1 ...)` with one parameter for each dimension of the new array, and should return a list of indices for *oldarray*, one for each dimension of *oldarray*.

mapfunc must be affine linear, meaning that each *oldarray* index must be formed by adding integer multiples (possibly negative) of some or all of *newidx1* etc, plus a possible integer offset. The multiples and offset must be the same in each call.

One good use for a shared array is to restrict the range of some dimensions, so as to apply say `array-for-each` or `array-fill!` to only part of an array. The plain `list` function can be used for *mapfunc* in this case, making no changes to the index values. For example,

```
(make-shared-array #2((a b c) (d e f) (g h i)) list 3 2)
⇒ #2((a b) (d e) (g h))
```

The new array can have fewer dimensions than *oldarray*, for example to take a column from an array.

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i) (list i 2))
                   '(0 2))
⇒ #1(c f i)
```

A diagonal can be taken by using the single new array index for both row and column in the old array. For example,

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i) (list i i))
                   '(0 2))
⇒ #1(a e i)
```

Dimensions can be increased by for instance considering portions of a one dimensional array as rows in a two dimensional array. (`array-contents` below can do the opposite, flattening an array.)

```
(make-shared-array #1(a b c d e f g h i j k l)
                   (lambda (i j) (list (+ (* i 3) j)))
                   4 3)
⇒ #2((a b c) (d e f) (g h i) (j k l))
```

By negating an index the order that elements appear can be reversed. The following just reverses the column order,

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i j) (list i (- 2 j)))
                   3 3)
⇒ #2((c b a) (f e d) (i h g))
```

A fixed offset on indexes allows for instance a change from a 0 based to a 1 based array,

```
(define x #2((a b c) (d e f) (g h i)))
(define y (make-shared-array x
                             (lambda (i j) (list (1- i) (1- j)))
                             '(1 3) '(1 3)))
(array-ref x 0 0) ⇒ a
(array-ref y 1 1) ⇒ a
```

A multiple on an index allows every Nth element of an array to be taken. The following is every third element,

```
(make-shared-array #1(a b c d e f g h i j k l)
                   (lambda (i) (list (* i 3)))
                   4)
```

```
⇒ #1(a d g j)
```

The above examples can be combined to make weird and wonderful selections from an array, but it's important to note that because *mapfunc* must be affine linear, arbitrary permutations are not possible.

In the current implementation, *mapfunc* is not called for every access to the new array but only on some sample points to establish a base and stride for new array indices in *oldarray* data. A few sample points are enough because *mapfunc* is linear.

`shared-array-increments` *array* [Scheme Procedure]
`scm_shared_array_increments` (*array*) [C Function]

For each dimension, return the distance between elements in the root vector.

`shared-array-offset` *array* [Scheme Procedure]
`scm_shared_array_offset` (*array*) [C Function]

Return the root vector index of the first element in the array.

`shared-array-root` *array* [Scheme Procedure]
`scm_shared_array_root` (*array*) [C Function]

Return the root vector of a shared array.

`array-contents` *array* [*strict*] [Scheme Procedure]
`scm_array_contents` (*array, strict*) [C Function]

If *array* may be *unrolled* into a one dimensional shared array without changing their order (last subscript changing fastest), then `array-contents` returns that shared array, otherwise it returns `#f`. All arrays made by `make-array` and `make-typed-array` may be unrolled, some arrays made by `make-shared-array` may not be.

If the optional argument *strict* is provided, a shared array will be returned only if its elements are stored internally contiguous in memory.

`transpose-array` *array dim1 . . .* [Scheme Procedure]
`scm_transpose_array` (*array, dimlist*) [C Function]

Return an array sharing contents with *array*, but with dimensions arranged in a different order. There must be one *dim* argument for each dimension of *array*. *dim1*, *dim2*, . . . should be integers between 0 and the rank of the array to be returned. Each integer in that range must appear at least once in the argument list.

The values of *dim1*, *dim2*, . . . correspond to dimensions in the array to be returned, and their positions in the argument list to dimensions of *array*. Several *dim*s may have the same value, in which case the returned array will have smaller rank than *array*.

```
(transpose-array '#2((a b) (c d)) 1 0) ⇒ #2((a c) (b d))
(transpose-array '#2((a b) (c d)) 0 0) ⇒ #1(a d)
(transpose-array '#3(((a b c) (d e f)) ((1 2 3) (4 5 6))) 1 1 0) ⇒
                 #2((a 4) (b 5) (c 6))
```

5.6.7.4 Accessing Arrays from C

Arrays, especially uniform numeric arrays, are useful to efficiently represent large amounts of rectangularily organized information, such as matrices, images, or generally blobs of binary data. It is desirable to access these blobs in a C like manner so that they can be handed to external C code such as linear algebra libraries or image processing routines.

While pointers to the elements of an array are in use, the array itself must be protected so that the pointer remains valid. Such a protected array is said to be *reserved*. A reserved array can be read but modifications to it that would cause the pointer to its elements to become invalid are prevented. When you attempt such a modification, an error is signalled.

(This is similar to locking the array while it is in use, but without the danger of a deadlock. In a multi-threaded program, you will need additional synchronization to avoid modifying reserved arrays.)

You must take care to always unreserve an array after reserving it, also in the presence of non-local exits. To simplify this, reserving and unreserving work like a dynwind context (see Section 5.11.9 [Dynamic Wind], page 266): a call to `scm_array_get_handle` can be thought of as beginning a dynwind context and `scm_array_handle_release` as ending it. When a non-local exit happens between these two calls, the array is implicitely unreserved.

That is, you need to properly pair reserving and unreserving in your code, but you don't need to worry about non-local exits.

These calls and other pairs of calls that establish dynwind contexts need to be properly nested. If you begin a context prior to reserving an array, you need to unreserve the array before ending the context. Likewise, when reserving two or more arrays in a certain order, you need to unreserve them in the opposite order.

Once you have reserved an array and have retrieved the pointer to its elements, you must figure out the layout of the elements in memory. Guile allows slices to be taken out of arrays without actually making a copy, such as making an alias for the diagonal of a matrix that can be treated as a vector. Arrays that result from such an operation are not stored contiguously in memory and when working with their elements directly, you need to take this into account.

The layout of array elements in memory can be defined via a *mapping function* that computes a scalar position from a vector of indices. The scalar position then is the offset of the element with the given indices from the start of the storage block of the array.

In Guile, this mapping function is restricted to be *affine*: all mapping functions of Guile arrays can be written as `p = b + c[0]*i[0] + c[1]*i[1] + ... + c[n-1]*i[n-1]` where `i[k]` is the `k`th index and `n` is the rank of the array. For example, a matrix of size 3x3 would have `b == 0`, `c[0] == 3` and `c[1] == 1`. When you transpose this matrix (with `transpose-array`, say), you will get an array whose mapping function has `b == 0`, `c[0] == 1` and `c[1] == 3`.

The function `scm_array_handle_dims` gives you (indirect) access to the coefficients `c[k]`.

Note that there are no functions for accessing the elements of a character array yet. Once the string implementation of Guile has been changed to use Unicode, we will provide them.

`scm_t_array_handle` [C Type]

> This is a structure type that holds all information necessary to manage the reservation of arrays as explained above. Structures of this type must be allocated on the stack and must only be accessed by the functions listed below.

`void scm_array_get_handle` (*SCM array, scm_t_array_handle *handle*) [C Function]

Reserve *array*, which must be an array, and prepare *handle* to be used with the functions below. You must eventually call `scm_array_handle_release` on *handle*, and do this in a properly nested fashion, as explained above. The structure pointed to by *handle* does not need to be initialized before calling this function.

`void scm_array_handle_release` (*scm_t_array_handle *handle*) [C Function]

End the array reservation represented by *handle*. After a call to this function, *handle* might be used for another reservation.

`size_t scm_array_handle_rank` (*scm_t_array_handle *handle*) [C Function]

Return the rank of the array represented by *handle*.

`scm_t_array_dim` [C Type]

This structure type holds information about the layout of one dimension of an array. It includes the following fields:

`ssize_t lbnd`
`ssize_t ubnd`

The lower and upper bounds (both inclusive) of the permissible index range for the given dimension. Both values can be negative, but *lbnd* is always less than or equal to *ubnd*.

`ssize_t inc`

The distance from one element of this dimension to the next. Note, too, that this can be negative.

`const scm_t_array_dim * scm_array_handle_dims` (*scm_t_array_handle *handle*) [C Function]

Return a pointer to a C vector of information about the dimensions of the array represented by *handle*. This pointer is valid as long as the array remains reserved. As explained above, the `scm_t_array_dim` structures returned by this function can be used calculate the position of an element in the storage block of the array from its indices.

This position can then be used as an index into the C array pointer returned by the various `scm_array_handle_<foo>_elements` functions, or with `scm_array_handle_ref` and `scm_array_handle_set`.

Here is how one can compute the position *pos* of an element given its indices in the vector *indices*:

```
ssize_t indices[RANK];
scm_t_array_dim *dims;
ssize_t pos;
size_t i;

pos = 0;
for (i = 0; i < RANK; i++)
  {
    if (indices[i] < dims[i].lbnd || indices[i] > dims[i].ubnd)
```

```
            out_of_range ();
          pos += (indices[i] - dims[i].lbnd) * dims[i].inc;
        }
```

ssize_t **scm_array_handle_pos** (*scm_t_array_handle *handle, SCM indices*) [C Function]

Compute the position corresponding to *indices*, a list of indices. The position is computed as described above for `scm_array_handle_dims`. The number of the indices and their range is checked and an approrpiate error is signalled for invalid indices.

SCM **scm_array_handle_ref** (*scm_t_array_handle *handle, ssize_t pos*) [C Function]

Return the element at position *pos* in the storage block of the array represented by *handle*. Any kind of array is acceptable. No range checking is done on *pos*.

void **scm_array_handle_set** (*scm_t_array_handle *handle, ssize_t pos, SCM val*) [C Function]

Set the element at position *pos* in the storage block of the array represented by *handle* to *val*. Any kind of array is acceptable. No range checking is done on *pos*. An error is signalled when the array can not store *val*.

const SCM * **scm_array_handle_elements** (*scm_t_array_handle *handle*) [C Function]

Return a pointer to the elements of a ordinary array of general Scheme values (i.e., a non-uniform array) for reading. This pointer is valid as long as the array remains reserved.

SCM * **scm_array_handle_writable_elements** (*scm_t_array_handle *handle*) [C Function]

Like `scm_array_handle_elements`, but the pointer is good for reading and writing.

const void * **scm_array_handle_uniform_elements** (*scm_t_array_handle *handle*) [C Function]

Return a pointer to the elements of a uniform numeric array for reading. This pointer is valid as long as the array remains reserved. The size of each element is given by `scm_array_handle_uniform_element_size`.

void * **scm_array_handle_uniform_writable_elements** (*scm_t_array_handle *handle*) [C Function]

Like `scm_array_handle_uniform_elements`, but the pointer is good reading and writing.

size_t **scm_array_handle_uniform_element_size** (*scm_t_array_handle *handle*) [C Function]

Return the size of one element of the uniform numeric array represented by *handle*.

const scm_t_uint8 * **scm_array_handle_u8_elements** (*scm_t_array_handle *handle*) [C Function]

const scm_t_int8 * **scm_array_handle_s8_elements** (*scm_t_array_handle *handle*) [C Function]

`const scm_t_uint16 * scm_array_handle_u16_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int16 * scm_array_handle_s16_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint32 * scm_array_handle_u32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int32 * scm_array_handle_s32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint64 * scm_array_handle_u64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int64 * scm_array_handle_s64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_f32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const double * scm_array_handle_f64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_c32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const double * scm_array_handle_c64_elements` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to the elements of a uniform numeric array of the indicated kind for reading. This pointer is valid as long as the array remains reserved.

The pointers for `c32` and `c64` uniform numeric arrays point to pairs of floating point numbers. The even index holds the real part, the odd index the imaginary part of the complex number.

`scm_t_uint8 * scm_array_handle_u8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int8 * scm_array_handle_s8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint16 * scm_array_handle_u16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int16 * scm_array_handle_s16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint32 * scm_array_handle_u32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int32 * scm_array_handle_s32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint64 * scm_array_handle_u64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int64 * scm_array_handle_s64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`float * scm_array_handle_f32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`double * scm_array_handle_f64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`float * scm_array_handle_c32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`double * scm_array_handle_c64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_<kind>_elements`, but the pointer is good for reading and writing.

`const scm_t_uint32 * scm_array_handle_bit_elements` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to the words that store the bits of the represented array, which must be a bit array.

Unlike other arrays, bit arrays have an additional offset that must be figured into index calculations. That offset is returned by `scm_array_handle_bit_elements_offset`.

To find a certain bit you first need to calculate its position as explained above for `scm_array_handle_dims` and then add the offset. This gives the absolute position of the bit, which is always a non-negative integer.

Each word of the bit array storage block contains exactly 32 bits, with the least significant bit in that word having the lowest absolute position number. The next word contains the next 32 bits.

Thus, the following code can be used to access a bit whose position according to `scm_array_handle_dims` is given in *pos*:

```
SCM bit_array;
scm_t_array_handle handle;
scm_t_uint32 *bits;
ssize_t pos;
size_t abs_pos;
size_t word_pos, mask;

scm_array_get_handle (&bit_array, &handle);
bits = scm_array_handle_bit_elements (&handle);

pos = ...
abs_pos = pos + scm_array_handle_bit_elements_offset (&handle);
word_pos = abs_pos / 32;
mask = 1L << (abs_pos % 32);

if (bits[word_pos] & mask)
  /* bit is set. */

scm_array_handle_release (&handle);
```

`scm_t_uint32 * scm_array_handle_bit_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_bit_elements` but the pointer is good for reading and writing. You must take care not to modify bits outside of the allowed index range of the array, even for contiguous arrays.

5.6.8 Records

A *record type* is a first class object representing a user-defined data type. A *record* is an instance of a record type.

`record?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a record of any type and `#f` otherwise.

Note that `record?` may be true of any Scheme value; there is no promise that records are disjoint with other Scheme types.

`make-record-type` *type-name field-names* [Scheme Procedure]

Return a *record-type descriptor*, a value representing a new data type disjoint from all others. The *type-name* argument must be a string, but is only used for debugging purposes (such as the printed representation of a record of the new type). The *field-names* argument is a list of symbols naming the *fields* of a record of the new type. It is an error if the list contains any duplicates. It is unspecified how record-type descriptors are represented.

`record-constructor` *rtd* [*field-names*] [Scheme Procedure]

Return a procedure for constructing new members of the type represented by *rtd*. The returned procedure accepts exactly as many arguments as there are symbols in the given list, *field-names*; these are used, in order, as the initial values of those fields in a new record, which is returned by the constructor procedure. The values of any fields not named in that list are unspecified. The *field-names* argument defaults to the list of field names in the call to `make-record-type` that created the type represented by *rtd*; if the *field-names* argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

`record-predicate` *rtd* [Scheme Procedure]

Return a procedure for testing membership in the type represented by *rtd*. The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type; it returns a false value otherwise.

`record-accessor` *rtd field-name* [Scheme Procedure]

Return a procedure for reading the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol *field-name* in that record. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

`record-modifier` *rtd field-name* [Scheme Procedure]

Return a procedure for writing the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol *field-name* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-descriptor` *record* [Scheme Procedure]

Return a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to `record-predicate`, the resulting predicate would return a true value when passed the given record. Note that it is not necessarily the case that the returned descriptor is the one that was passed to `record-constructor` in the call that created the constructor procedure that created the given record.

`record-type-name` *rtd* [Scheme Procedure]

Return the type-name associated with the type represented by rtd. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-fields` *rtd* [Scheme Procedure]

Return a list of the symbols naming the fields in members of the type represented by *rtd*. The returned value is `equal?` to the field-names argument given in the call to `make-record-type` that created the type represented by *rtd*.

5.6.9 Structures

[FIXME: this is pasted in from Tom Lord's original guile.texi and should be reviewed]

A *structure type* is a first class user-defined data type. A *structure* is an instance of a structure type. A structure type is itself a structure.

Structures are less abstract and more general than traditional records. In fact, in Guile Scheme, records are implemented using structures.

5.6.9.1 Structure Concepts

A structure object consists of a handle, structure data, and a vtable. The handle is a Scheme value which points to both the vtable and the structure's data. Structure data is a dynamically allocated region of memory, private to the structure, divided up into typed fields. A vtable is another structure used to hold type-specific data. Multiple structures can share a common vtable.

When applied to structures, the `equal?` predicate (see Section 5.9.1 [Equality], page 236) returns `#t` if the two structures share a common vtable *and* all their fields satisfy `equal?`.

Three concepts are key to understanding structures.

- *layout specifications*

  Layout specifications determine how memory allocated to structures is divided up into fields. Programmers must write a layout specification whenever a new type of structure is defined.

- *structural accessors*

  Structure access is by field number. There is only one set of accessors common to all structure objects.

- *vtables*

  Vtables, themselves structures, are first class representations of disjoint sub-types of structures in general. In most cases, when a new structure is created, programmers must specify a vtable for the new structure. Each vtable has a field describing the layout of its instances. Vtables can have additional, user-defined fields as well.

5.6.9.2 Structure Layout

When a structure is created, a region of memory is allocated to hold its state. The *layout* of the structure's type determines how that memory is divided into fields.

Each field has a specified type. There are only three types allowed, each corresponding to a one letter code. The allowed types are:

- 'u' – unprotected

  The field holds binary data that is not GC protected.
- 'p' – protected

  The field holds a Scheme value and is GC protected.
- 's' – self

  The field holds a Scheme value and is GC protected. When a structure is created with this type of field, the field is initialized to refer to the structure's own handle. This kind of field is mainly useful when mixing Scheme and C code in which the C code may need to compute a structure's handle given only the address of its malloc'd data.

Each field also has an associated access protection. There are only three kinds of protection, each corresponding to a one letter code. The allowed protections are:

- 'w' – writable

  The field can be read and written.
- 'r' – readable

  The field can be read, but not written.
- 'o' – opaque

  The field can be neither read nor written. This kind of protection is for fields useful only to built-in routines.

A layout specification is described by stringing together pairs of letters: one to specify a field type and one to specify a field protection. For example, a traditional cons pair type object could be described as:

```
; cons pairs have two writable fields of Scheme data
"pwpw"
```

A pair object in which the first field is held constant could be:

```
"prpw"
```

Binary fields, (fields of type "u"), hold one *word* each. The size of a word is a machine dependent value defined to be equal to the value of the C expression: `sizeof (long)`.

The last field of a structure layout may specify a tail array. A tail array is indicated by capitalizing the field's protection code ('W', 'R' or 'O'). A tail-array field is replaced by a read-only binary data field containing an array size. The array size is determined at the time the structure is created. It is followed by a corresponding number of fields of the type specified for the tail array. For example, a conventional Scheme vector can be described as:

```
; A vector is an arbitrary number of writable fields holding Scheme
; values:
"pW"
```

In the above example, field 0 contains the size of the vector and fields beginning at 1 contain the vector elements.

A kind of tagged vector (a constant tag followed by conventional vector elements) might be:

```
"prpW"
```

Structure layouts are represented by specially interned symbols whose name is a string of type and protection codes. To create a new structure layout, use this procedure:

`make-struct-layout` *fields* [Scheme Procedure]
`scm_make_struct_layout` (*fields*) [C Function]

Return a new structure layout object.

fields must be a string made up of pairs of characters strung together. The first character of each pair describes a field type, the second a field protection. Allowed types are 'p' for GC-protected Scheme data, 'u' for unprotected binary data, and 's' for a field that points to the structure itself. Allowed protections are 'w' for mutable fields, 'r' for read-only fields, and 'o' for opaque fields. The last field protection specification may be capitalized to indicate that the field is a tail-array.

5.6.9.3 Structure Basics

This section describes the basic procedures for creating and accessing structures.

`make-struct` *vtable tail_array_size . init* [Scheme Procedure]
`scm_make_struct` (*vtable, tail_array_size, init*) [C Function]

Create a new structure.

type must be a vtable structure (see Section 5.6.9.4 [Vtables], page 208).

tail-elts must be a non-negative integer. If the layout specification indicated by *type* includes a tail-array, this is the number of elements allocated to that array.

The *init1*, . . . are optional arguments describing how successive fields of the structure should be initialized. Only fields with protection 'r' or 'w' can be initialized, except for fields of type 's', which are automatically initialized to point to the new structure itself; fields with protection 'o' can not be initialized by Scheme programs.

If fewer optional arguments than initializable fields are supplied, fields of type 'p' get default value #f while fields of type 'u' are initialized to 0.

Structs are currently the basic representation for record-like data structures in Guile. The plan is to eventually replace them with a new representation which will at the same time be easier to use and more powerful.

For more information, see the documentation for `make-vtable-vtable`.

`struct?` *x* [Scheme Procedure]
`scm_struct_p` (*x*) [C Function]

Return `#t` iff *x* is a structure object, else `#f`.

`struct-ref` *handle pos* [Scheme Procedure]
`struct-set!` *struct n value* [Scheme Procedure]
`scm_struct_ref` (*handle, pos*) [C Function]
`scm_struct_set_x` (*struct, n, value*) [C Function]

Access (or modify) the *n*th field of *struct*.

If the field is of type 'p', then it can be set to an arbitrary value.

If the field is of type 'u', then it can only be set to a non-negative integer value small enough to fit in one machine word.

5.6.9.4 Vtables

Vtables are structures that are used to represent structure types. Each vtable contains a layout specification in field `vtable-index-layout` – instances of the type are laid out according to that specification. Vtables contain additional fields which are used only internally to libguile. The variable `vtable-offset-user` is bound to a field number. Vtable fields at that position or greater are user definable.

`struct-vtable` *handle* [Scheme Procedure]
`scm_struct_vtable` (*handle*) [C Function]
Return the vtable structure that describes the type of *struct*.

`struct-vtable?` *x* [Scheme Procedure]
`scm_struct_vtable_p` (*x*) [C Function]
Return `#t` iff *x* is a vtable structure.

If you have a vtable structure, `V`, you can create an instance of the type it describes by using `(make-struct V ...)`. But where does `V` itself come from? One possibility is that `V` is an instance of a user-defined vtable type, `V'`, so that `V` is created by using `(make-struct V' ...)`. Another possibility is that `V` is an instance of the type it itself describes. Vtable structures of the second sort are created by this procedure:

`make-vtable-vtable` *user_fields tail_array_size . init* [Scheme Procedure]
`scm_make_vtable_vtable` (*user_fields*, *tail_array_size*, *init*) [C Function]
Return a new, self-describing vtable structure.

user-fields is a string describing user defined fields of the vtable beginning at index `vtable-offset-user` (see `make-struct-layout`).

tail-size specifies the size of the tail-array (if any) of this vtable.

init1, . . . are the optional initializers for the fields of the vtable.

Vtables have one initializable system field—the struct printer. This field comes before the user fields in the initializers passed to `make-vtable-vtable` and `make-struct`, and thus works as a third optional argument to `make-vtable-vtable` and a fourth to `make-struct` when creating vtables:

If the value is a procedure, it will be called instead of the standard printer whenever a struct described by this vtable is printed. The procedure will be called with arguments STRUCT and PORT.

The structure of a struct is described by a vtable, so the vtable is in essence the type of the struct. The vtable is itself a struct with a vtable. This could go on forever if it weren't for the vtable-vtables which are self-describing vtables, and thus terminate the chain.

There are several potential ways of using structs, but the standard one is to use three kinds of structs, together building up a type sub-system: one vtable-vtable working as the root and one or several "types", each with a set of "instances". (The vtable-vtable should be compared to the class `<class>` which is the class of itself.)

```
(define ball-root (make-vtable-vtable "pr" 0))

(define (make-ball-type ball-color)
  (make-struct ball-root 0
       (make-struct-layout "pw")
               (lambda (ball port)
                 (format port "#<a ~A ball owned by ~A>"
                         (color ball)
                         (owner ball)))
               ball-color))
(define (color ball) (struct-ref (struct-vtable ball) vtable-offset-user))
(define (owner ball) (struct-ref ball 0))

(define red (make-ball-type 'red))
(define green (make-ball-type 'green))

(define (make-ball type owner) (make-struct type 0 owner))

(define ball (make-ball green 'Nisse))
ball ⇒ #<a green ball owned by Nisse>
```

`struct-vtable-name` *vtable* [Scheme Procedure]
`scm_struct_vtable_name` (*vtable*) [C Function]
    Return the name of the vtable *vtable*.

`set-struct-vtable-name!` *vtable name* [Scheme Procedure]
`scm_set_struct_vtable_name_x` (*vtable*, *name*) [C Function]
    Set the name of the vtable *vtable* to *name*.

`struct-vtable-tag` *handle* [Scheme Procedure]
`scm_struct_vtable_tag` (*handle*) [C Function]
    Return the vtable tag of the structure *handle*.

5.6.10 Dictionary Types

A *dictionary* object is a data structure used to index information in a user-defined way. In standard Scheme, the main aggregate data types are lists and vectors. Lists are not really indexed at all, and vectors are indexed only by number (e.g. `(vector-ref foo 5)`). Often you will find it useful to index your data on some other type; for example, in a library catalog you might want to look up a book by the name of its author. Dictionaries are used to help you organize information in such a way.

An *association list* (or *alist* for short) is a list of key-value pairs. Each pair represents a single quantity or object; the `car` of the pair is a key which is used to identify the object, and the `cdr` is the object's value.

A *hash table* also permits you to index objects with arbitrary keys, but in a way that makes looking up any one object extremely fast. A well-designed hash system makes hash table lookups almost as fast as conventional array or vector references.

Alists are popular among Lisp programmers because they use only the language's primitive operations (lists, *car*, *cdr* and the equality primitives). No changes to the language core are necessary. Therefore, with Scheme's built-in list manipulation facilities, it is very convenient to handle data stored in an association list. Also, alists are highly portable and can be easily implemented on even the most minimal Lisp systems.

However, alists are inefficient, especially for storing large quantities of data. Because we want Guile to be useful for large software systems as well as small ones, Guile provides a rich set of tools for using either association lists or hash tables.

5.6.11 Association Lists

An association list is a conventional data structure that is often used to implement simple key-value databases. It consists of a list of entries in which each entry is a pair. The *key* of each entry is the `car` of the pair and the *value* of each entry is the `cdr`.

```
ASSOCIATION LIST ::=  '( (KEY1 . VALUE1)
                         (KEY2 . VALUE2)
                         (KEY3 . VALUE3)
                         ...
                       )
```

Association lists are also known, for short, as *alists*.

The structure of an association list is just one example of the infinite number of possible structures that can be built using pairs and lists. As such, the keys and values in an association list can be manipulated using the general list structure procedures `cons`, `car`, `cdr`, `set-car!`, `set-cdr!` and so on. However, because association lists are so useful, Guile also provides specific procedures for manipulating them.

5.6.11.1 Alist Key Equality

All of Guile's dedicated association list procedures, apart from `acons`, come in three flavours, depending on the level of equality that is required to decide whether an existing key in the association list is the same as the key that the procedure call uses to identify the required entry.

- Procedures with *assq* in their name use `eq?` to determine key equality.
- Procedures with *assv* in their name use `eqv?` to determine key equality.
- Procedures with *assoc* in their name use `equal?` to determine key equality.

`acons` is an exception because it is used to build association lists which do not require their entries' keys to be unique.

5.6.11.2 Adding or Setting Alist Entries

`acons` adds a new entry to an association list and returns the combined association list. The combined alist is formed by consing the new entry onto the head of the alist specified in the `acons` procedure call. So the specified alist is not modified, but its contents become shared with the tail of the combined alist that `acons` returns.

In the most common usage of `acons`, a variable holding the original association list is updated with the combined alist:

```
(set! address-list (acons name address address-list))
```

In such cases, it doesn't matter that the old and new values of `address-list` share some of their contents, since the old value is usually no longer independently accessible.

Note that `acons` adds the specified new entry regardless of whether the alist may already contain entries with keys that are, in some sense, the same as that of the new entry. Thus `acons` is ideal for building alists where there is no concept of key uniqueness.

```
(set! task-list (acons 3 "pay gas bill" '()))
task-list
⇒
((3 . "pay gas bill"))

(set! task-list (acons 3 "tidy bedroom" task-list))
task-list
⇒
((3 . "tidy bedroom") (3 . "pay gas bill"))
```

`assq-set!`, `assv-set!` and `assoc-set!` are used to add or replace an entry in an association list where there *is* a concept of key uniqueness. If the specified association list already contains an entry whose key is the same as that specified in the procedure call, the existing entry is replaced by the new one. Otherwise, the new entry is consed onto the head of the old association list to create the combined alist. In all cases, these procedures return the combined alist.

`assq-set!` and friends *may* destructively modify the structure of the old association list in such a way that an existing variable is correctly updated without having to `set!` it to the value returned:

```
address-list
⇒
(("mary" . "34 Elm Road") ("james" . "16 Bow Street"))

(assoc-set! address-list "james" "1a London Road")
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

Or they may not:

```
(assoc-set! address-list "bob" "11 Newington Avenue")
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

The only safe way to update an association list variable when adding or replacing an entry like this is to `set!` the variable to the returned value:

```
(set! address-list
      (assoc-set! address-list "bob" "11 Newington Avenue"))
```

```
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))
```

Because of this slight inconvenience, you may find it more convenient to use hash tables to store dictionary data. If your application will not be modifying the contents of an alist very often, this may not make much difference to you.

If you need to keep the old value of an association list in a form independent from the list that results from modification by `acons`, `assq-set!`, `assv-set!` or `assoc-set!`, use `list-copy` to copy the old association list before modifying it.

`acons` *key value alist* [Scheme Procedure]
`scm_acons` (*key, value, alist*) [C Function]

Add a new key-value pair to *alist*. A new pair is created whose car is *key* and whose cdr is *value*, and the pair is consed onto *alist*, and the new list is returned. This function is *not* destructive; *alist* is not modified.

`assq-set!` *alist key val* [Scheme Procedure]
`assv-set!` *alist key value* [Scheme Procedure]
`assoc-set!` *alist key value* [Scheme Procedure]
`scm_assq_set_x` (*alist, key, val*) [C Function]
`scm_assv_set_x` (*alist, key, val*) [C Function]
`scm_assoc_set_x` (*alist, key, val*) [C Function]

Reassociate *key* in *alist* with *value*: find any existing *alist* entry for *key* and associate it with the new *value*. If *alist* does not contain an entry for *key*, add a new one. Return the (possibly new) alist.

These functions do not attempt to verify the structure of *alist*, and so may cause unusual results if passed an object that is not an association list.

5.6.11.3 Retrieving Alist Entries

`assq`, `assv` and `assoc` find the entry in an alist for a given key, and return the `(`*`key`*` . `*`value`*`)` pair. `assq-ref`, `assv-ref` and `assoc-ref` do a similar lookup, but return just the *value*.

`assq` *key alist* [Scheme Procedure]
`assv` *key alist* [Scheme Procedure]
`assoc` *key alist* [Scheme Procedure]
`scm_assq` (*key, alist*) [C Function]
`scm_assv` (*key, alist*) [C Function]
`scm_assoc` (*key, alist*) [C Function]

Return the first entry in *alist* with the given *key*. The return is the pair `(KEY . VALUE)` from *alist*. If there's no matching entry the return is `#f`.

`assq` compares keys with `eq?`, `assv` uses `eqv?` and `assoc` uses `equal?`. See also SRFI-1 which has an extended `assoc` (Section 6.4.3.9 [SRFI-1 Association Lists], page 434).

`assq-ref` *alist key* [Scheme Procedure]
`assv-ref` *alist key* [Scheme Procedure]

`assoc-ref` *alist key* [Scheme Procedure]
`scm_assq_ref` (*alist, key*) [C Function]
`scm_assv_ref` (*alist, key*) [C Function]
`scm_assoc_ref` (*alist, key*) [C Function]

Return the value from the first entry in *alist* with the given *key*, or `#f` if there's no such entry.

`assq-ref` compares keys with `eq?`, `assv-ref` uses `eqv?` and `assoc-ref` uses `equal?`.

Notice these functions have the *key* argument last, like other `-ref` functions, but this is opposite to what what `assq` etc above use.

When the return is `#f` it can be either *key* not found, or an entry which happens to have value `#f` in the `cdr`. Use `assq` etc above if you need to differentiate these cases.

5.6.11.4 Removing Alist Entries

To remove the element from an association list whose key matches a specified key, use `assq-remove!`, `assv-remove!` or `assoc-remove!` (depending, as usual, on the level of equality required between the key that you specify and the keys in the association list).

As with `assq-set!` and friends, the specified alist may or may not be modified destructively, and the only safe way to update a variable containing the alist is to `set!` it to the value that `assq-remove!` and friends return.

```
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("james" . "1a London Road"))
```

Note that, when `assq/v/oc-remove!` is used to modify an association list that has been constructed only using the corresponding `assq/v/oc-set!`, there can be at most one matching entry in the alist, so the question of multiple entries being removed in one go does not arise. If `assq/v/oc-remove!` is applied to an association list that has been constructed using `acons`, or an `assq/v/oc-set!` with a different level of equality, or any mixture of these, it removes only the first matching entry from the alist, even if the alist might contain further matching entries. For example:

```
(define address-list '())
(set! address-list (assq-set! address-list "mary" "11 Elm Street"))
(set! address-list (assq-set! address-list "mary" "57 Pine Drive"))
address-list
⇒
(("mary" . "57 Pine Drive") ("mary" . "11 Elm Street"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
```

```
(("mary" . "11 Elm Street"))
```

In this example, the two instances of the string "mary" are not the same when compared using `eq?`, so the two `assq-set!` calls add two distinct entries to `address-list`. When compared using `equal?`, both "mary"s in `address-list` are the same as the "mary" in the `assoc-remove!` call, but `assoc-remove!` stops after removing the first matching entry that it finds, and so one of the "mary" entries is left in place.

`assq-remove!` *alist key* [Scheme Procedure]
`assv-remove!` *alist key* [Scheme Procedure]
`assoc-remove!` *alist key* [Scheme Procedure]
`scm_assq_remove_x` (*alist, key*) [C Function]
`scm_assv_remove_x` (*alist, key*) [C Function]
`scm_assoc_remove_x` (*alist, key*) [C Function]

Delete the first entry in *alist* associated with *key*, and return the resulting alist.

5.6.11.5 Sloppy Alist Functions

`sloppy-assq`, `sloppy-assv` and `sloppy-assoc` behave like the corresponding non-`sloppy-` procedures, except that they return `#f` when the specified association list is not well-formed, where the non-`sloppy-` versions would signal an error.

Specifically, there are two conditions for which the non-`sloppy-` procedures signal an error, which the `sloppy-` procedures handle instead by returning `#f`. Firstly, if the specified alist as a whole is not a proper list:

```
(assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
ERROR: In procedure assoc in expression (assoc "mary" (quote #)):
ERROR: Wrong type argument in position 2 (expecting association list): ((1 . 2) ("key"

(sloppy-assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
#f
```

Secondly, if one of the entries in the specified alist is not a pair:

```
(assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
ERROR: In procedure assoc in expression (assoc 2 (quote #)):
ERROR: Wrong type argument in position 2 (expecting association list): ((1 . 1) 2 (3 .

(sloppy-assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
#f
```

Unless you are explicitly working with badly formed association lists, it is much safer to use the non-`sloppy-` procedures, because they help to highlight coding and data errors that the `sloppy-` versions would silently cover up.

`sloppy-assq` *key alist* [Scheme Procedure]
`scm_sloppy_assq` (*key, alist*) [C Function]

Behaves like `assq` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assv` *key alist* [Scheme Procedure]
`scm_sloppy_assv` (*key, alist*) [C Function]

Behaves like `assv` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assoc` *key alist* [Scheme Procedure]
`scm_sloppy_assoc` (*key, alist*) [C Function]

Behaves like `assoc` but does not do any error checking. Recommended only for use in Guile internals.

5.6.11.6 Alist Example

Here is a longer example of how alists may be used in practice.

```
(define capitals '(("New York" . "Albany")
                   ("Oregon"   . "Salem")
                   ("Florida"  . "Miami")))

;; What's the capital of Oregon?
(assoc "Oregon" capitals)       ⇒ ("Oregon" . "Salem")
(assoc-ref capitals "Oregon")   ⇒ "Salem"

;; We left out South Dakota.
(set! capitals
      (assoc-set! capitals "South Dakota" "Pierre"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Miami"))

;; And we got Florida wrong.
(set! capitals
      (assoc-set! capitals "Florida" "Tallahassee"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Tallahassee"))

;; After Oregon secedes, we can remove it.
(set! capitals
      (assoc-remove! capitals "Oregon"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Florida" . "Tallahassee"))
```

5.6.12 Hash Tables

Hash tables are dictionaries which offer similar functionality as association lists: They provide a mapping from keys to values. The difference is that association lists need time linear in the size of elements when searching for entries, whereas hash tables can normally search in constant time. The drawback is that hash tables require a little bit more memory, and that you can not use the normal list procedures (see Section 5.6.2 [Lists], page 168) for working with them.

Guile provides two types of hashtables. One is an abstract data type that can only be manipulated with the functions in this section. The other type is concrete: it uses a normal vector with alists as elements. The advantage of the abstract hash tables is that they will be automatically resized when they become too full or too empty.

5.6.12.1 Hash Table Examples

For demonstration purposes, this section gives a few usage examples of some hash table procedures, together with some explanation what they do.

First we start by creating a new hash table with 31 slots, and populate it with two key/value pairs.

```
(define h (make-hash-table 31))

;; This is an opaque object
h
⇒
#<hash-table 0/31>

;; We can also use a vector of alists.
(define h (make-vector 7 '()))

h
⇒
#(() () () () () () ())

;; Inserting into a hash table can be done with hashq-set!
(hashq-set! h 'foo "bar")
⇒
"bar"

(hashq-set! h 'braz "zonk")
⇒
"zonk"

;; Or with hash-create-handle!
(hashq-create-handle! h 'frob #f)
⇒
(frob . #f)

;; The vector now contains three elements in the alists and the frob
;; entry is at index (hashq 'frob).
h
⇒
#(() () () () ((frob . #f) (braz . "zonk")) () ((foo . "bar")))

(hashq 'frob)
⇒
4
```

You can get the value for a given key with the procedure `hashq-ref`, but the problem with this procedure is that you cannot reliably determine whether a key does exists in the table. The reason is that the procedure returns `#f` if the key is not in the table, but it will return the same value if the key is in the table and just happens to have the value `#f`, as you can see in the following examples.

```
(hashq-ref h 'foo)
⇒
"bar"

(hashq-ref h 'frob)
⇒
#f

(hashq-ref h 'not-there)
⇒
#f
```

Better is to use the procedure `hashq-get-handle`, which makes a distinction between the two cases. Just like `assq`, this procedure returns a key/value-pair on success, and `#f` if the key is not found.

```
(hashq-get-handle h 'foo)
⇒
(foo . "bar")

(hashq-get-handle h 'not-there)
⇒
#f
```

There is no procedure for calculating the number of key/value-pairs in a hash table, but `hash-fold` can be used for doing exactly that.

```
(hash-fold (lambda (key value seed) (+ 1 seed)) 0 h)
⇒
3
```

5.6.12.2 Hash Table Reference

Like the association list functions, the hash table functions come in several varieties, according to the equality test used for the keys. Plain `hash-` functions use `equal?`, `hashq-` functions use `eq?`, `hashv-` functions use `eqv?`, and the `hashx-` functions use an application supplied test.

A single `make-hash-table` creates a hash table suitable for use with any set of functions, but it's imperative that just one set is then used consistently, or results will be unpredictable.

Hash tables are implemented as a vector indexed by a hash value formed from the key, with an association list of key/value pairs for each bucket in case distinct keys hash together. Direct access to the pairs in those lists is provided by the `-handle-` functions. The abstract kind of hash tables hide the vector in an opaque object that represents the hash table, while for the concrete kind the vector *is* the hashtable.

When the number of table entries in an abstract hash table goes above a threshold, the vector is made larger and the entries are rehashed, to prevent the bucket lists from becoming too long and slowing down accesses. When the number of entries goes below a threshold, the vector is shrunk to save space.

A abstract hash table is created with `make-hash-table`. To create a vector that is suitable as a hash table, use `(make-vector size '())`, for example.

For the `hashx-` "extended" routines, an application supplies a *hash* function producing an integer index like `hashq` etc below, and an *assoc* alist search function like `assq` etc (see Section 5.6.11.3 [Retrieving Alist Entries], page 212). Here's an example of such functions implementing case-insensitive hashing of string keys,

```
(use-modules (srfi srfi-1)
             (srfi srfi-13))

(define (my-hash str size)
  (remainder (string-hash-ci str) size))
(define (my-assoc str alist)
  (find (lambda (pair) (string-ci=? str (car pair))) alist))

(define my-table (make-hash-table))
(hashx-set! my-hash my-assoc my-table "foo" 123)

(hashx-ref my-hash my-assoc my-table "FOO")
⇒ 123
```

In a `hashx-` *hash* function the aim is to spread keys across the vector, so bucket lists don't become long. But the actual values are arbitrary as long as they're in the range 0 to $size - 1$. Helpful functions for forming a hash value, in addition to `hashq` etc below, include `symbol-hash` (see Section 5.5.7.2 [Symbol Keys], page 155), `string-hash` and `string-hash-ci` (see Section 5.5.5.7 [String Comparison], page 136), and `char-set-hash` (see Section 5.5.4.1 [Character Set Predicates/Comparison], page 123).

`make-hash-table` [*size*] [Scheme Procedure]

Create a new abstract hash table object, with an optional minimum vector *size*.

When *size* is given, the table vector will still grow and shrink automatically, as described above, but with *size* as a minimum. If an application knows roughly how many entries the table will hold then it can use *size* to avoid rehashing when initial entries are added.

`hash-table?` *obj* [Scheme Procedure]

`scm_hash_table_p` (*obj*) [C Function]

Return `#t` if *obj* is a abstract hash table object.

`hash-clear!` *table* [Scheme Procedure]
`scm_hash_clear_x` (*table*) [C Function]

Remove all items from *table* (without triggering a resize).

`hash-ref` *table key* [*dflt*] [Scheme Procedure]
`hashq-ref` *table key* [*dflt*] [Scheme Procedure]
`hashv-ref` *table key* [*dflt*] [Scheme Procedure]
`hashx-ref` *hash assoc table key* [*dflt*] [Scheme Procedure]
`scm_hash_ref` (*table, key, dflt*) [C Function]
`scm_hashq_ref` (*table, key, dflt*) [C Function]
`scm_hashv_ref` (*table, key, dflt*) [C Function]
`scm_hashx_ref` (*hash, assoc, table, key, dflt*) [C Function]

Lookup *key* in the given hash *table*, and return the associated value. If *key* is not found, return *dflt*, or `#f` if *dflt* is not given.

`hash-set!` *table key val* [Scheme Procedure]
`hashq-set!` *table key val* [Scheme Procedure]
`hashv-set!` *table key val* [Scheme Procedure]
`hashx-set!` *hash assoc table key val* [Scheme Procedure]
`scm_hash_set_x` (*table, key, val*) [C Function]
`scm_hashq_set_x` (*table, key, val*) [C Function]
`scm_hashv_set_x` (*table, key, val*) [C Function]
`scm_hashx_set_x` (*hash, assoc, table, key, val*) [C Function]

Associate *val* with *key* in the given hash *table*. If *key* is already present then it's associated value is changed. If it's not present then a new entry is created.

`hash-remove!` *table key* [Scheme Procedure]
`hashq-remove!` *table key* [Scheme Procedure]
`hashv-remove!` *table key* [Scheme Procedure]
`hashx-remove!` *hash assoc table key* [Scheme Procedure]
`scm_hash_remove_x` (*table, key*) [C Function]
`scm_hashq_remove_x` (*table, key*) [C Function]
`scm_hashv_remove_x` (*table, key*) [C Function]
`scm_hashx_remove_x` (*hash, assoc, table, key*) [C Function]

Remove any association for *key* in the given hash *table*. If *key* is not in *table* then nothing is done.

`hash` *key size* [Scheme Procedure]
`hashq` *key size* [Scheme Procedure]
`hashv` *key size* [Scheme Procedure]
`scm_hash` (*key, size*) [C Function]
`scm_hashq` (*key, size*) [C Function]
`scm_hashv` (*key, size*) [C Function]

Return a hash value for *key*. This is a number in the range 0 to $size - 1$, which is suitable for use in a hash table of the given *size*.

Note that `hashq` and `hashv` may use internal addresses of objects, so if an object is garbage collected and re-created it can have a different hash value, even when the two are notionally `eq?`. For instance with symbols,

```
(hashq 'something 123)   ⇒ 19
(gc)
(hashq 'something 123)   ⇒ 62
```

In normal use this is not a problem, since an object entered into a hash table won't be garbage collected until removed. It's only if hashing calculations are somehow separated from normal references that its lifetime needs to be considered.

`hash-get-handle` *table key* [Scheme Procedure]
`hashq-get-handle` *table key* [Scheme Procedure]
`hashv-get-handle` *table key* [Scheme Procedure]
`hashx-get-handle` *hash assoc table key* [Scheme Procedure]
`scm_hash_get_handle` (*table, key*) [C Function]
`scm_hashq_get_handle` (*table, key*) [C Function]
`scm_hashv_get_handle` (*table, key*) [C Function]
`scm_hashx_get_handle` (*hash, assoc, table, key*) [C Function]

Return the `(key . value)` pair for *key* in the given hash *table*, or `#f` if *key* is not in *table*.

`hash-create-handle!` *table key init* [Scheme Procedure]
`hashq-create-handle!` *table key init* [Scheme Procedure]
`hashv-create-handle!` *table key init* [Scheme Procedure]
`hashx-create-handle!` *hash assoc table key init* [Scheme Procedure]
`scm_hash_create_handle_x` (*table, key, init*) [C Function]
`scm_hashq_create_handle_x` (*table, key, init*) [C Function]
`scm_hashv_create_handle_x` (*table, key, init*) [C Function]
`scm_hashx_create_handle_x` (*hash, assoc, table, key, init*) [C Function]

Return the `(key . value)` pair for *key* in the given hash *table*. If *key* is not in *table* then create an entry for it with *init* as the value, and return that pair.

`hash-map->list` *proc table* [Scheme Procedure]
`hash-for-each` *proc table* [Scheme Procedure]
`scm_hash_map_to_list` (*proc, table*) [C Function]
`scm_hash_for_each` (*proc, table*) [C Function]

Apply *proc* to the entries in the given hash *table*. Each call is `(proc key value)`. `hash-map->list` returns a list of the results from these calls, `hash-for-each` discards the results and returns an unspecified value.

Calls are made over the table entries in an unspecified order, and for `hash-map->list` the order of the values in the returned list is unspecified. Results will be unpredictable if *table* is modified while iterating.

For example the following returns a new alist comprising all the entries from `mytable`, in no particular order.

```
(hash-map->list cons mytable)
```

`hash-for-each-handle` *proc table* [Scheme Procedure] `scm_hash_for_each_handle` (*proc, table*) [C Function]

Apply *proc* to the entries in the given hash *table*. Each call is `(proc handle)`, where *handle* is a `(key . value)` pair. Return an unspecified value.

`hash-for-each-handle` differs from `hash-for-each` only in the argument list of *proc*.

`hash-fold` *proc init table* [Scheme Procedure]
`scm_hash_fold` (*proc, init, table*) [C Function]

Accumulate a result by applying *proc* to the elements of the given hash *table*. Each call is `(proc key value prior-result)`, where *key* and *value* are from the *table* and *prior-result* is the return from the previous *proc* call. For the first call, *prior-result* is the given *init* value.

Calls are made over the table entries in an unspecified order. Results will be unpredictable if *table* is modified while `hash-fold` is running.

For example, the following returns a count of how many keys in `mytable` are strings.

```
(hash-fold (lambda (key value prior)
             (if (string? key) (1+ prior) prior))
           0 mytable)
```

5.7 Smobs

This chapter contains reference information related to defining and working with smobs. See Section 4.4 [Defining New Types (Smobs)], page 68 for a tutorial-like introduction to smobs.

`scm_t_bits scm_make_smob_type` (*const char *name, size_t size*) [Function]

This function adds a new smob type, named *name*, with instance size *size*, to the system. The return value is a tag that is used in creating instances of the type.

If *size* is 0, the default *free* function will do nothing.

If *size* is not 0, the default *free* function will deallocate the memory block pointed to by `SCM_SMOB_DATA` with `scm_gc_free`. The *WHAT* parameter in the call to `scm_gc_free` will be *NAME*.

Default values are provided for the *mark*, *free*, *print*, and *equalp* functions, as described in Section 4.4 [Defining New Types (Smobs)], page 68. If you want to customize any of these functions, the call to `scm_make_smob_type` should be immediately followed by calls to one or several of `scm_set_smob_mark`, `scm_set_smob_free`, `scm_set_smob_print`, and/or `scm_set_smob_equalp`.

`void scm_set_smob_mark` (*scm_t_bits tc, SCM (*mark) (SCM obj)*) [C Function]

This function sets the smob marking procedure for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *mark* procedure must cause `scm_gc_mark` to be called for every `SCM` value that is directly referenced by the smob instance *obj*. One of these `SCM` values can be returned from the procedure and Guile will call `scm_gc_mark` for it. This can be used to avoid deep recursions for smob instances that form a list.

It must not call any libguile function or macro except `scm_gc_mark`, `SCM_SMOB_FLAGS`, `SCM_SMOB_DATA`, `SCM_SMOB_DATA_2`, and `SCM_SMOB_DATA_3`.

`void scm_set_smob_free` (*scm_t_bits tc, size_t (*free) (SCM obj)*) [C Function]

This function sets the smob freeing procedure for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *free* procedure must deallocate all resources that are directly associated with the smob instance *OBJ*. It must assume that all `SCM` values that it references have already been freed and are thus invalid.

It must also not call any libguile function or macro except `scm_gc_free`, `SCM_SMOB_FLAGS`, `SCM_SMOB_DATA`, `SCM_SMOB_DATA_2`, and `SCM_SMOB_DATA_3`.

The *free* procedure must return 0.

`void scm_set_smob_print` (*scm_t_bits tc, int (*print) (SCM obj, SCM port, scm_print_state* pstate)*) [C Function]

This function sets the smob printing procedure for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *print* procedure should output a textual representation of the smob instance *obj* to *port*, using information in *pstate*.

The textual representation should be of the form `#<name ...>`. This ensures that `read` will not interpret it as some other Scheme value.

It is often best to ignore *pstate* and just print to *port* with `scm_display`, `scm_write`, `scm_simple_format`, and `scm_puts`.

`void scm_set_smob_equalp` (*scm_t_bits tc, SCM* (**equalp*) (*SCM obj1, SCM obj1*)) [C Function]

This function sets the smob equality-testing predicate for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *equalp* procedure should return `SCM_BOOL_T` when *obj1* is `equal?` to *obj2*. Else it should return *SCM_BOOL_F*. Both *obj1* and *obj2* are instances of the smob type *tc*.

`void scm_assert_smob_type` (*scm_t_bits tag, SCM val*) [C Function]

When *val* is a smob of the type indicated by *tag*, do nothing. Else, signal an error.

`int SCM_SMOB_PREDICATE` (*scm_t_bits tag, SCM exp*) [C Macro]

Return true iff *exp* is a smob instance of the type indicated by *tag*. The expression *exp* can be evaluated more than once, so it shouldn't contain any side effects.

`void SCM_NEWSMOB` (*SCM value, scm_t_bits tag, void *data*) [C Macro]
`void SCM_NEWSMOB2` (*SCM value, scm_t_bits tag, void *data, void *data2*) [C Macro]
`void SCM_NEWSMOB3` (*SCM value, scm_t_bits tag, void *data, void *data2, void *data3*) [C Macro]

Make *value* contain a smob instance of the type with tag *tag* and smob data *data*, *data2*, and *data3*, as appropriate.

The *tag* is what has been returned by `scm_make_smob_type`. The initial values *data*, *data2*, and *data3* are of type `scm_t_bits`; when you want to use them for `SCM` values, these values need to be converted to a `scm_t_bits` first by using `SCM_UNPACK`.

The flags of the smob instance start out as zero.

Since it is often the case (e.g., in smob constructors) that you will create a smob instance and return it, there is also a slightly specialized macro for this situation:

`SCM_RETURN_NEWSMOB` (*scm_t_bits tag, void *data*) [C Macro]
`SCM_RETURN_NEWSMOB2` (*scm_t_bits tag, void *data1, void *data2*) [C Macro]
`SCM_RETURN_NEWSMOB3` (*scm_t_bits tag, void *data1, void *data2, void *data3*) [C Macro]

This macro expands to a block of code that creates a smob instance of the type with tag *tag* and smob data *data*, *data2*, and *data3*, as with `SCM_NEWSMOB`, etc., and causes the surrounding function to return that `SCM` value. It should be the last piece of code in a block.

`scm_t_bits SCM_SMOB_FLAGS` (*SCM obj*) [C Macro]

Return the 16 extra bits of the smob *obj*. No meaning is predefined for these bits, you can use them freely.

`scm_t_bits SCM_SET_SMOB_FLAGS` (*SCM obj, scm_t_bits flags*) [C Macro]

Set the 16 extra bits of the smob *obj* to *flags*. No meaning is predefined for these bits, you can use them freely.

`scm_t_bits SCM_SMOB_DATA` (*SCM obj*) [C Macro]
`scm_t_bits SCM_SMOB_DATA_2` (*SCM obj*) [C Macro]
`scm_t_bits SCM_SMOB_DATA_3` (*SCM obj*) [C Macro]

Return the first (second, third) immediate word of the smob *obj* as a `scm_t_bits` value. When the word contains a `SCM` value, use `SCM_SMOB_OBJECT` (etc.) instead.

`void SCM_SET_SMOB_DATA` (*SCM obj, scm_t_bits val*) [C Macro]
`void SCM_SET_SMOB_DATA_2` (*SCM obj, scm_t_bits val*) [C Macro]
`void SCM_SET_SMOB_DATA_3` (*SCM obj, scm_t_bits val*) [C Macro]

Set the first (second, third) immediate word of the smob *obj* to *val*. When the word should be set to a `SCM` value, use `SCM_SMOB_SET_OBJECT` (etc.) instead.

`SCM SCM_SMOB_OBJECT` (*SCM obj*) [C Macro]
`SCM SCM_SMOB_OBJECT_2` (*SCM obj*) [C Macro]
`SCM SCM_SMOB_OBJECT_3` (*SCM obj*) [C Macro]

Return the first (second, third) immediate word of the smob *obj* as a `SCM` value. When the word contains a `scm_t_bits` value, use `SCM_SMOB_DATA` (etc.) instead.

`void SCM_SET_SMOB_OBJECT` (*SCM obj, SCM val*) [C Macro]
`void SCM_SET_SMOB_OBJECT_2` (*SCM obj, SCM val*) [C Macro]
`void SCM_SET_SMOB_OBJECT_3` (*SCM obj, SCM val*) [C Macro]

Set the first (second, third) immediate word of the smob *obj* to *val*. When the word should be set to a `scm_t_bits` value, use `SCM_SMOB_SET_DATA` (etc.) instead.

`SCM * SCM_SMOB_OBJECT_LOC` (*SCM obj*) [C Macro]
`SCM * SCM_SMOB_OBJECT_2_LOC` (*SCM obj*) [C Macro]
`SCM * SCM_SMOB_OBJECT_3_LOC` (*SCM obj*) [C Macro]

Return a pointer to the first (second, third) immediate word of the smob *obj*. Note that this is a pointer to `SCM`. If you need to work with `scm_t_bits` values, use `SCM_PACK` and `SCM_UNPACK`, as appropriate.

`SCM scm_markcdr` (*SCM x*) [Function]

Mark the references in the smob *x*, assuming that *x*'s first data word contains an ordinary Scheme object, and *x* refers to no other objects. This function simply returns *x*'s first data word.

5.8 Procedures and Macros

5.8.1 Lambda: Basic Procedure Creation

A `lambda` expression evaluates to a procedure. The environment which is in effect when a `lambda` expression is evaluated is enclosed in the newly created procedure, this is referred to as a *closure* (see Section 3.1.4 [About Closure], page 24).

When a procedure created by `lambda` is called with some actual arguments, the environment enclosed in the procedure is extended by binding the variables named in the formal argument list to new locations and storing the actual arguments into these locations. Then the body of the `lambda` expression is evaluation sequentially. The result of the last expression in the procedure body is then the result of the procedure invocation.

The following examples will show how procedures can be created using `lambda`, and what you can do with these procedures.

```
(lambda (x) (+ x x))         ⇒ a procedure
((lambda (x) (+ x x)) 4)     ⇒ 8
```

The fact that the environment in effect when creating a procedure is enclosed in the procedure is shown with this example:

```
(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                     ⇒ 10
```

`lambda` *formals body* [syntax]

formals should be a formal argument list as described in the following table.

`(variable1 ...)`
: The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored into the newly created location for the formal variables.

`variable`
: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments will converted into a list and stored into the newly created location for the formal variable.

`(variable1 ... variablen . variablen+1)`
: If a space-delimited period precedes the last variable, then the procedure takes *n* or more variables where *n* is the number of formal arguments before the period. There must be at least one argument before the period. The first *n* actual arguments will be stored into the newly allocated locations for the first *n* formal arguments and the sequence of the remaining actual arguments is converted into a list and the stored into the location for the last formal argument. If there are exactly *n* actual arguments, the empty list is stored into the location of the last formal argument.

The list in *variable* or *variablen+1* is always newly created and the procedure can modify it if desired. This is the case even when the procedure is invoked via `apply`, the required part of the list argument there will be copied (see Section 5.13.3 [Procedures for On the Fly Evaluation], page 291).

body is a sequence of Scheme expressions which are evaluated in order when the procedure is invoked.

5.8.2 Primitive Procedures

Procedures written in C can be registered for use from Scheme, provided they take only arguments of type `SCM` and return `SCM` values. `scm_c_define_gsubr` is likely to be the most useful mechanism, combining the process of registration (`scm_c_make_gsubr`) and definition (`scm_define`).

`SCM scm_c_make_gsubr` (*const char *name, int req, int opt, int rst, fcn*) [Function]

Register a C procedure *FCN* as a "subr" — a primitive subroutine that can be called from Scheme. It will be associated with the given *name* but no environment binding will be created. The arguments *req*, *opt* and *rst* specify the number of required, optional and "rest" arguments respectively. The total number of these arguments should match the actual number of arguments to *fcn*. The number of rest arguments should be 0 or 1. `scm_c_make_gsubr` returns a value of type `SCM` which is a "handle" for the procedure.

`SCM scm_c_define_gsubr` (*const char *name, int req, int opt, int rst, fcn*) [Function]

Register a C procedure *FCN*, as for `scm_c_make_gsubr` above, and additionally create a top-level Scheme binding for the procedure in the "current environment" using `scm_define`. `scm_c_define_gsubr` returns a handle for the procedure in the same way as `scm_c_make_gsubr`, which is usually not further required.

`scm_c_make_gsubr` and `scm_c_define_gsubr` automatically use `scm_c_make_subr` and also `scm_makcclo` if necessary. It is advisable to use the gsubr variants since they provide a slightly higher-level abstraction of the Guile implementation.

5.8.3 Optional Arguments

Scheme procedures, as defined in R5RS, can either handle a fixed number of actual arguments, or a fixed number of actual arguments followed by arbitrarily many additional arguments. Writing procedures of variable arity can be useful, but unfortunately, the syntactic means for handling argument lists of varying length is a bit inconvenient. It is possible to give names to the fixed number of argument, but the remaining (optional) arguments can be only referenced as a list of values (see Section 5.8.1 [Lambda], page 225).

Guile comes with the module `(ice-9 optargs)`, which makes using optional arguments much more convenient. In addition, this module provides syntax for handling keywords in argument lists (see Section 5.5.8 [Keywords], page 162).

Before using any of the procedures or macros defined in this section, you have to load the module `(ice-9 optargs)` with the statement:

```
(use-modules (ice-9 optargs))
```

5.8.3.1 let-optional Reference

The syntax `let-optional` and `let-optional*` are for destructuring rest argument lists and giving names to the various list elements. `let-optional` binds all variables simultaneously, while `let-optional*` binds them sequentially, consistent with `let` and `let*` (see Section 5.10.2 [Local Bindings], page 248).

`let-optional` *rest-arg* (*binding* ...) *expr* ... [library syntax]
`let-optional*` *rest-arg* (*binding* ...) *expr* ... [library syntax]

These two macros give you an optional argument interface that is very *Schemey* and introduces no fancy syntax. They are compatible with the scsh macros of the same name, but are slightly extended. Each of *binding* may be of one of the forms *var* or `(var default-value)`. *rest-arg* should be the rest-argument of the procedures these are used from. The items in *rest-arg* are sequentially bound to the variable names are given. When *rest-arg* runs out, the remaining vars are bound either to the default values or `#f` if no default value was specified. *rest-arg* remains bound to whatever may have been left of *rest-arg*.

After binding the variables, the expressions *expr* ... are evaluated in order.

5.8.3.2 let-keywords Reference

`let-keywords` and `let-keywords*` are used for extracting values from argument lists which use keywords instead of argument position for binding local variables to argument values.

`let-keywords` binds all variables simultaneously, while `let-keywords*` binds them sequentially, consistent with `let` and `let*` (see Section 5.10.2 [Local Bindings], page 248).

`let-keywords` *rest-arg allow-other-keys?* (*binding* ...) *expr* ... [library syntax]
`let-keywords*` *rest-arg allow-other-keys?* (*binding* ...) *expr* ... [library syntax]

These macros pick out keyword arguments from *rest-arg*, but do not modify it. This is consistent at least with Common Lisp, which duplicates keyword arguments in the rest argument. More explanation of what keyword arguments in a lambda list look like can be found below in the documentation for `lambda*` (see Section 5.8.3.3 [lambda* Reference], page 227). *binding*s can have the same form as for `let-optional`. If *allow-other-keys?* is false, an error will be thrown if anything that looks like a keyword argument but does not match a known keyword parameter will result in an error.

After binding the variables, the expressions *expr* ... are evaluated in order.

5.8.3.3 lambda* Reference

When using optional and keyword argument lists, `lambda` for creating a procedure then `let-optional` or `let-keywords` is a bit lengthy. `lambda*` combines the features of those macros into a single convenient syntax.

`lambda*` ([*var*...] [library syntax]
[*#:optional vardef*...]
[*#:key vardef*... [*#:allow-other-keys*]]
[*#:rest var* | . *var*])
body

Create a procedure which takes optional and/or keyword arguments specified with `#:optional` and `#:key`. For example,

```
(lambda* (a b #:optional c d . e) '())
```

is a procedure with fixed arguments *a* and *b*, optional arguments *c* and *d*, and rest argument *e*. If the optional arguments are omitted in a call, the variables for them are bound to `#f`.

`lambda*` can also take keyword arguments. For example, a procedure defined like this:

```
(lambda* (#:key xyzzy larch) '())
```

can be called with any of the argument lists `(#:xyzzy 11)`, `(#:larch 13)`, `(#:larch 42 #:xyzzy 19)`, `()`. Whichever arguments are given as keywords are bound to values (and those not given are `#f`).

Optional and keyword arguments can also have default values to take when not present in a call, by giving a two-element list of variable name and expression. For example in

```
(lambda* (foo #:optional (bar 42) #:key (baz 73))
     (list foo bar baz))
```

foo is a fixed argument, *bar* is an optional argument with default value 42, and baz is a keyword argument with default value 73. Default value expressions are not evaluated unless they are needed, and until the procedure is called.

Normally it's an error if a call has keywords other than those specified by `#:key`, but adding `#:allow-other-keys` to the definition (after the keyword argument declarations) will ignore unknown keywords.

If a call has a keyword given twice, the last value is used. For example,

```
((lambda* (#:key (heads 0) (tails 0))
   (display (list heads tails)))
 #:heads 37 #:tails 42 #:heads 99)
⊣ (99 42)
```

`#:rest` is a synonym for the dotted syntax rest argument. The argument lists `(a . b)` and `(a #:rest b)` are equivalent in all respects. This is provided for more similarity to DSSSL, MIT-Scheme and Kawa among others, as well as for refugees from other Lisp dialects.

When `#:key` is used together with a rest argument, the keyword parameters in a call all remain in the rest list. This is the same as Common Lisp. For example,

```
((lambda* (#:key (x 0) #:allow-other-keys #:rest r)
   (display r))
 #:x 123 #:y 456)
⊣ (#:x 123 #:y 456)
```

`#:optional` and `#:key` establish their bindings successively, from left to right, as per `let-optional*` and `let-keywords*`. This means default expressions can refer back to prior parameters, for example

```
(lambda* (start #:optional (end (+ 10 start)))
  (do ((i start (1+ i)))
      ((> i end))
    (display i)))
```

5.8.3.4 define* Reference

Just like `define` has a shorthand notation for defining procedures (see Section 3.1.2.4 [Lambda Alternatives], page 18), `define*` is provided as an abbreviation of the combination of `define` and `lambda*`.

`define*-public` is the `lambda*` version of `define-public`; `defmacro*` and `defmacro*-public` exist for defining macros with the improved argument list handling possibilities. The `-public` versions not only define the procedures/macros, but also export them from the current module.

`define*` *formals body* [library syntax]
`define*-public` *formals body* [library syntax]

`define*` and `define*-public` support optional arguments with a similar syntax to `lambda*`. They also support arbitrary-depth currying, just like Guile's define. Some examples:

```
(define* (x y #:optional a (z 3) #:key w . u)
   (display (list y z u)))
```

defines a procedure `x` with a fixed argument *y*, an optional argument *a*, another optional argument *z* with default value 3, a keyword argument *w*, and a rest argument *u*.

```
(define-public* ((foo #:optional bar) #:optional baz) '())
```

This illustrates currying. A procedure `foo` is defined, which, when called with an optional argument *bar*, returns a procedure that takes an optional argument *baz*.

Of course, `define*[-public]` also supports `#:rest` and `#:allow-other-keys` in the same way as `lambda*`.

`defmacro*` *name formals body* [library syntax]
`defmacro*-public` *name formals body* [library syntax]

These are just like `defmacro` and `defmacro-public` except that they take `lambda*`-style extended parameter lists, where `#:optional`, `#:key`, `#:allow-other-keys` and `#:rest` are allowed with the usual semantics. Here is an example of a macro with an optional argument:

```
(defmacro* transmorgify (a #:optional b)
    (a 1))
```

5.8.4 Procedure Properties and Meta-information

Procedures always have attached the environment in which they were created and information about how to apply them to actual arguments. In addition to that, properties and meta-information can be stored with procedures. The procedures in this section can be used to test whether a given procedure satisfies a condition; and to access and set a procedure's property.

The first group of procedures are predicates to test whether a Scheme object is a procedure, or a special procedure, respectively. `procedure?` is the most general predicates, it returns `#t` for any kind of procedure. `closure?` does not return `#t` for primitive procedures, and `thunk?` only returns `#t` for procedures which do not accept any arguments.

`procedure?` *obj* [Scheme Procedure]
`scm_procedure_p` (*obj*) [C Function]

Return `#t` if *obj* is a procedure.

`closure?` *obj* [Scheme Procedure]
`scm_closure_p` (*obj*) [C Function]

Return `#t` if *obj* is a closure.

`thunk?` *obj* [Scheme Procedure]
`scm_thunk_p` (*obj*) [C Function]

Return `#t` if *obj* is a thunk.

Procedure properties are general properties to be attached to procedures. These can be the name of a procedure or other relevant information, such as debug hints.

`procedure-name` *proc* [Scheme Procedure]
`scm_procedure_name` (*proc*) [C Function]

Return the name of the procedure *proc*

`procedure-source` *proc* [Scheme Procedure]
`scm_procedure_source` (*proc*) [C Function]

Return the source of the procedure *proc*.

`procedure-environment` *proc* [Scheme Procedure]
`scm_procedure_environment` (*proc*) [C Function]

Return the environment of the procedure *proc*.

`procedure-properties` *proc* [Scheme Procedure]
`scm_procedure_properties` (*proc*) [C Function]

Return *obj*'s property list.

`procedure-property` *obj key* [Scheme Procedure]
`scm_procedure_property` (*obj, key*) [C Function]

Return the property of *obj* with name *key*.

`set-procedure-properties!` *proc alist* [Scheme Procedure]
`scm_set_procedure_properties_x` (*proc, alist*) [C Function]

Set *obj*'s property list to *alist*.

`set-procedure-property!` *obj key value* [Scheme Procedure]
`scm_set_procedure_property_x` (*obj, key, value*) [C Function]

In *obj*'s property list, set the property named *key* to *value*.

Documentation for a procedure can be accessed with the procedure `procedure-documentation`.

`procedure-documentation` *proc* [Scheme Procedure]
`scm_procedure_documentation` (*proc*) [C Function]

Return the documentation string associated with `proc`. By convention, if a procedure contains more than one expression and the first expression is a string constant, that string is assumed to contain documentation for that procedure.

5.8.5 Procedures with Setters

A *procedure with setter* is a special kind of procedure which normally behaves like any accessor procedure, that is a procedure which accesses a data structure. The difference is that this kind of procedure has a so-called *setter* attached, which is a procedure for storing something into a data structure.

Procedures with setters are treated specially when the procedure appears in the special form `set!` (REFFIXME). How it works is best shown by example.

Suppose we have a procedure called `foo-ref`, which accepts two arguments, a value of type `foo` and an integer. The procedure returns the value stored at the given index in the `foo` object. Let `f` be a variable containing such a `foo` data structure.[1]

```
(foo-ref f 0)       ⇒ bar
(foo-ref f 1)       ⇒ braz
```

Also suppose that a corresponding setter procedure called `foo-set!` does exist.

```
(foo-set! f 0 'bla)
(foo-ref f 0)       ⇒ bla
```

Now we could create a new procedure called `foo`, which is a procedure with setter, by calling `make-procedure-with-setter` with the accessor and setter procedures `foo-ref` and `foo-set!`. Let us call this new procedure `foo`.

```
(define foo (make-procedure-with-setter foo-ref foo-set!))
```

`foo` can from now an be used to either read from the data structure stored in `f`, or to write into the structure.

```
(set! (foo f 0) 'dum)
(foo f 0)          ⇒ dum
```

`make-procedure-with-setter` *procedure setter* [Scheme Procedure]
`scm_make_procedure_with_setter` (*procedure, setter*) [C Function]

Create a new procedure which behaves like *procedure*, but with the associated setter *setter*.

`procedure-with-setter?` *obj* [Scheme Procedure]
`scm_procedure_with_setter_p` (*obj*) [C Function]

Return `#t` if *obj* is a procedure with an associated setter procedure.

`procedure` *proc* [Scheme Procedure]
`scm_procedure` (*proc*) [C Function]

Return the procedure of *proc*, which must be either a procedure with setter, or an operator struct.

`setter` *proc* [Scheme Procedure]

Return the setter of *proc*, which must be either a procedure with setter or an operator struct.

5.8.6 Lisp Style Macro Definitions

Macros are objects which cause the expression that they appear in to be transformed in some way *before* being evaluated. In expressions that are intended for macro transformation, the identifier that names the relevant macro must appear as the first element, like this:

[1] Working definitions would be:

```
(define foo-ref vector-ref)
(define foo-set! vector-set!)
(define f (make-vector 2 #f))
```

```
(macro-name macro-args ...)
```

In Lisp-like languages, the traditional way to define macros is very similar to procedure definitions. The key differences are that the macro definition body should return a list that describes the transformed expression, and that the definition is marked as a macro definition (rather than a procedure definition) by the use of a different definition keyword: in Lisp, `defmacro` rather than `defun`, and in Scheme, `define-macro` rather than `define`.

Guile supports this style of macro definition using both `defmacro` and `define-macro`. The only difference between them is how the macro name and arguments are grouped together in the definition:

```
(defmacro name (args ...) body ...)
```

is the same as

```
(define-macro (name args ...) body ...)
```

The difference is analogous to the corresponding difference between Lisp's `defun` and Scheme's `define`.

`false-if-exception`, from the '`boot-9.scm`' file in the Guile distribution, is a good example of macro definition using `defmacro`:

```
(defmacro false-if-exception (expr)
  `(catch #t
          (lambda () ,expr)
          (lambda args #f)))
```

The effect of this definition is that expressions beginning with the identifier `false-if-exception` are automatically transformed into a `catch` expression following the macro definition specification. For example:

```
(false-if-exception (open-input-file "may-not-exist"))
≡
(catch #t
       (lambda () (open-input-file "may-not-exist"))
       (lambda args #f))
```

5.8.7 The R5RS `syntax-rules` System

R5RS defines an alternative system for macro and syntax transformations using the keywords `define-syntax`, `let-syntax`, `letrec-syntax` and `syntax-rules`.

The main difference between the R5RS system and the traditional macros of the previous section is how the transformation is specified. In R5RS, rather than permitting a macro definition to return an arbitrary expression, the transformation is specified in a pattern language that

- does not require complicated quoting and extraction of components of the source expression using `caddr` etc.
- is designed such that the bindings associated with identifiers in the transformed expression are well defined, and such that it is impossible for the transformed expression to construct new identifiers.

The last point is commonly referred to as being *hygienic*: the R5RS `syntax-case` system provides *hygienic macros*.

For example, the R5RS pattern language for the `false-if-exception` example of the previous section looks like this:

```
(syntax-rules ()
  ((_ expr)
   (catch #t
          (lambda () expr)
          (lambda args #f))))
```

In Guile, the `syntax-rules` system is provided by the `(ice-9 syncase)` module. To make these facilities available in your code, include the expression `(use-syntax (ice-9 syncase))` (see Section 5.16.3.2 [Using Guile Modules], page 305) before the first usage of `define-syntax` etc. If you are writing a Scheme module, you can alternatively include the form `#:use-syntax (ice-9 syncase)` in your `define-module` declaration (see Section 5.16.3.3 [Creating Guile Modules], page 307).

5.8.7.1 The `syntax-rules` Pattern Language

5.8.7.2 Top Level Syntax Definitions

define-syntax: The gist is

(define-syntax <keyword> <transformer-spec>)

makes the <keyword> into a macro so that

(<keyword> ...)

expands at _compile_ or _read_ time (i.e. before any evaluation begins) into some expression that is given by the <transformer-spec>.

5.8.7.3 Local Syntax Definitions

5.8.8 Support for the `syntax-case` System

5.8.9 Internal Representation of Macros and Syntax

Internally, Guile uses three different flavors of macros. The three flavors are called *acro* (or *syntax*), *macro* and *mmacro*.

Given the expression

```
(foo ...)
```

with `foo` being some flavor of macro, one of the following things will happen when the expression is evaluated.

- When `foo` has been defined to be an *acro*, the procedure used in the acro definition of `foo` is passed the whole expression and the current lexical environment, and whatever that procedure returns is the value of evaluating the expression. You can think of this a procedure that receives its argument as an unevaluated expression.
- When `foo` has been defined to be a *macro*, the procedure used in the macro definition of `foo` is passed the whole expression and the current lexical environment, and whatever that procedure returns is evaluated again. That is, the procedure should return a valid Scheme expression.

- When `foo` has been defined to be a *mmacro*, the procedure used in the mmacro definition of 'foo' is passed the whole expression and the current lexical environment, and whatever that procedure returns replaces the original expression. Evaluation then starts over from the new expression that has just been returned.

The key difference between a *macro* and a *mmacro* is that the expression returned by a *mmacro* procedure is remembered (or *memoized*) so that the expansion does not need to be done again next time the containing code is evaluated.

The primitives `procedure->syntax`, `procedure->macro` and `procedure->memoizing-macro` are used to construct acros, macros and mmacros respectively. However, if you do not have a very special reason to use one of these primitives, you should avoid them: they are very specific to Guile's current implementation and therefore likely to change. Use `defmacro`, `define-macro` (see Section 5.8.6 [Macros], page 231) or `define-syntax` (see Section 5.8.7 [Syntax Rules], page 232) instead. (In low level terms, `defmacro`, `define-macro` and `define-syntax` are all implemented as mmacros.)

`procedure->syntax` *code* [Scheme Procedure]
`scm_makacro` (*code*) [C Function]

Return a macro which, when a symbol defined to this value appears as the first symbol in an expression, returns the result of applying *code* to the expression and the environment.

`procedure->macro` *code* [Scheme Procedure]
`scm_makmacro` (*code*) [C Function]

Return a macro which, when a symbol defined to this value appears as the first symbol in an expression, evaluates the result of applying *code* to the expression and the environment. For example:

```
(define trace
  (procedure->macro
    (lambda (x env)
      `(set! ,(cadr x) (tracef ,(cadr x) ',(cadr x))))))

(trace foo)
≡
(set! foo (tracef foo 'foo)).
```

`procedure->memoizing-macro` *code* [Scheme Procedure]
`scm_makmmacro` (*code*) [C Function]

Return a macro which, when a symbol defined to this value appears as the first symbol in an expression, evaluates the result of applying *code* to the expression and the environment. `procedure->memoizing-macro` is the same as `procedure->macro`, except that the expression returned by *code* replaces the original macro expression in the memoized form of the containing code.

In the following primitives, *acro* flavor macros are referred to as *syntax transformers*.

`macro?` *obj* [Scheme Procedure]
`scm_macro_p` (*obj*) [C Function]

Return `#t` if *obj* is a regular macro, a memoizing macro or a syntax transformer.

`macro-type` *m* [Scheme Procedure]
`scm_macro_type` (*m*) [C Function]

Return one of the symbols `syntax`, `macro` or `macro!`, depending on whether *m* is a syntax transformer, a regular macro, or a memoizing macro, respectively. If *m* is not a macro, `#f` is returned.

`macro-name` *m* [Scheme Procedure]
`scm_macro_name` (*m*) [C Function]

Return the name of the macro *m*.

`macro-transformer` *m* [Scheme Procedure]
`scm_macro_transformer` (*m*) [C Function]

Return the transformer of the macro *m*.

`cons-source` *xorig x y* [Scheme Procedure]
`scm_cons_source` (*xorig*, *x*, *y*) [C Function]

Create and return a new pair whose car and cdr are *x* and *y*. Any source properties associated with *xorig* are also associated with the new pair.

5.9 General Utility Functions

This chapter contains information about procedures which are not cleanly tied to a specific data type. Because of their wide range of applications, they are collected in a *utility* chapter.

5.9.1 Equality

There are three kinds of core equality predicates in Scheme, described below. The same kinds of comparisons arise in other functions, like `memq` and friends (see Section 5.6.2.7 [List Searching], page 173).

For all three tests, objects of different types are never equal. So for instance a list and a vector are not `equal?`, even if their contents are the same. Exact and inexact numbers are considered different types too, and are hence not equal even if their values are the same.

`eq?` tests just for the same object (essentially a pointer comparison). This is fast, and can be used when searching for a particular object, or when working with symbols or keywords (which are always unique objects).

`eqv?` extends `eq?` to look at the value of numbers and characters. It can for instance be used somewhat like `=` (see Section 5.5.2.8 [Comparison], page 111) but without an error if one operand isn't a number.

`equal?` goes further, it looks (recursively) into the contents of lists, vectors, etc. This is good for instance on lists that have been read or calculated in various places and are the same, just not made up of the same pairs. Such lists look the same (when printed), and `equal?` will consider them the same.

`eq?` *x y* [Scheme Procedure]
`scm_eq_p` (*x*, *y*) [C Function]

Return `#t` if *x* and *y* are the same object, except for numbers and characters. For example,

```
(define x (vector 1 2 3))
(define y (vector 1 2 3))

(eq? x x)  ⇒ #t
(eq? x y)  ⇒ #f
```

Numbers and characters are not equal to any other object, but the problem is they're not necessarily `eq?` to themselves either. This is even so when the number comes directly from a variable,

```
(let ((n (+ 2 3)))
  (eq? n n))       ⇒ *unspecified*
```

Generally `eqv?` below should be used when comparing numbers or characters. `=` (see Section 5.5.2.8 [Comparison], page 111) or `char=?` (see Section 5.5.3 [Characters], page 121) can be used too.

It's worth noting that end-of-list `()`, `#t`, `#f`, a symbol of a given name, and a keyword of a given name, are unique objects. There's just one of each, so for instance no matter how `()` arises in a program, it's the same object and can be compared with `eq?`,

```
(define x (cdr '(123)))
(define y (cdr '(456)))
(eq? x y) ⇒ #t

(define x (string->symbol "foo"))
(eq? x 'foo) ⇒ #t
```

`int scm_is_eq` (*SCM x, SCM y*) [C Function]

Return `1` when *x* and *y* are equal in the sense of `eq?`, otherwise return `0`.

The `==` operator should not be used on `SCM` values, an `SCM` is a C type which cannot necessarily be compared using `==` (see Section 5.2 [The SCM Type], page 94).

`eqv?` *x y* [Scheme Procedure]
`scm_eqv_p` (*x, y*) [C Function]

Return `#t` if *x* and *y* are the same object, or for characters and numbers the same value.

On objects except characters and numbers, `eqv?` is the same as `eq?` above, it's true if *x* and *y* are the same object.

If *x* and *y* are numbers or characters, `eqv?` compares their type and value. An exact number is not `eqv?` to an inexact number (even if their value is the same).

```
(eqv? 3 (+ 1 2)) ⇒ #t
(eqv? 1 1.0)     ⇒ #f
```

`equal?` *x y* [Scheme Procedure]
`scm_equal_p` (*x, y*) [C Function]

Return `#t` if *x* and *y* are the same type, and their contents or value are equal.

For a pair, string, vector, array or structure, `equal?` compares the contents, and does so using using the same `equal?` recursively, so a deep structure can be traversed.

```
(equal? (list 1 2 3) (list 1 2 3))   ⇒ #t
(equal? (list 1 2 3) (vector 1 2 3)) ⇒ #f
```

For other objects, `equal?` compares as per `eqv?` above, which means characters and numbers are compared by type and value (and like `eqv?`, exact and inexact numbers are not `equal?`, even if their value is the same).

```
(equal? 3 (+ 1 2)) ⇒ #t
(equal? 1 1.0)     ⇒ #f
```

Hash tables are currently only compared as per `eq?`, so two different tables are not `equal?`, even if their contents are the same.

`equal?` does not support circular data structures, it may go into an infinite loop if asked to compare two circular lists or similar.

New application-defined object types (see Section 4.4 [Defining New Types (Smobs)], page 68) have an `equalp` handler which is called by `equal?`. This lets an application traverse the contents or control what is considered `equal?` for two objects of such a type. If there's no such handler, the default is to just compare as per `eq?`.

5.9.2 Object Properties

It's often useful to associate a piece of additional information with a Scheme object even though that object does not have a dedicated slot available in which the additional information could be stored. Object properties allow you to do just that.

Guile's representation of an object property is a procedure-with-setter (see Section 5.8.5 [Procedures with Setters], page 230) that can be used with the generalized form of `set!` (REFFIXME) to set and retrieve that property for any Scheme object. So, setting a property looks like this:

```
(set! (my-property obj1) value-for-obj1)
(set! (my-property obj2) value-for-obj2)
```

And retrieving values of the same property looks like this:

```
(my-property obj1)
⇒
value-for-obj1

(my-property obj2)
⇒
value-for-obj2
```

To create an object property in the first place, use the `make-object-property` procedure:

```
(define my-property (make-object-property))
```

`make-object-property` [Scheme Procedure]

Create and return an object property. An object property is a procedure-with-setter that can be called in two ways. `(set! (`*`property obj`*`) `*`val`*`)` sets *obj*'s *property* to *val*. `(`*`property obj`*`)` returns the current setting of *obj*'s *property*.

A single object property created by `make-object-property` can associate distinct property values with all Scheme values that are distinguishable by `eq?` (including, for example, integers).

Internally, object properties are implemented using a weak key hash table. This means that, as long as a Scheme value with property values is protected from garbage collection, its property values are also protected. When the Scheme value is collected, its entry in the property table is removed and so the (ex-) property values are no longer protected by the table.

5.9.2.1 Low Level Property Implementation.

`primitive-make-property` *not-found-proc* [Scheme Procedure]
`scm_primitive_make_property` (*not_found_proc*) [C Function]

Create a *property token* that can be used with `primitive-property-ref` and `primitive-property-set!`. See `primitive-property-ref` for the significance of *not-found-proc*.

`primitive-property-ref` *prop obj* [Scheme Procedure]
`scm_primitive_property_ref` (*prop, obj*) [C Function]

Return the property *prop* of *obj*.

When no value has yet been associated with *prop* and *obj*, the *not-found-proc* from *prop* is used. A call `(not-found-proc prop obj)` is made and the result set as the property value. If *not-found-proc* is `#f` then `#f` is the property value.

`primitive-property-set!` *prop obj val* [Scheme Procedure]
`scm_primitive_property_set_x` (*prop, obj, val*) [C Function]

Set the property *prop* of *obj* to *val*.

`primitive-property-del!` *prop obj* [Scheme Procedure]
`scm_primitive_property_del_x` (*prop, obj*) [C Function]

Remove any value associated with *prop* and *obj*.

5.9.2.2 An Older Approach to Properties

Traditionally, Lisp systems provide a different object property interface to that provided by `make-object-property`, in which the object property that is being set or retrieved is indicated by a symbol.

Guile includes this older kind of interface as well, but it may well be removed in a future release, as it is less powerful than `make-object-property` and so increases the size of the Guile library for no benefit. (And it is trivial to write a compatibility layer in Scheme.)

`object-properties` *obj* [Scheme Procedure]
`scm_object_properties` (*obj*) [C Function]

Return *obj*'s property list.

`set-object-properties!` *obj alist* [Scheme Procedure]
`scm_set_object_properties_x` (*obj, alist*) [C Function]

Set *obj*'s property list to *alist*.

`object-property` *obj key* [Scheme Procedure]
`scm_object_property` (*obj, key*) [C Function]

Return the property of *obj* with name *key*.

`set-object-property!` *obj key value* [Scheme Procedure]
`scm_set_object_property_x` (*obj, key, value*) [C Function]

In *obj*'s property list, set the property named *key* to *value*.

5.9.3 Sorting

Sorting is very important in computer programs. Therefore, Guile comes with several sorting procedures built-in. As always, procedures with names ending in `!` are side-effecting, that means that they may modify their parameters in order to produce their results.

The first group of procedures can be used to merge two lists (which must be already sorted on their own) and produce sorted lists containing all elements of the input lists.

`merge` *alist blist less* [Scheme Procedure]
`scm_merge` (*alist, blist, less*) [C Function]

Merge two already sorted lists into one. Given two lists *alist* and *blist*, such that `(sorted? alist less?)` and `(sorted? blist less?)`, return a new list in which the elements of *alist* and *blist* have been stably interleaved so that `(sorted? (merge alist blist less?) less?)`. Note: this does _not_ accept vectors.

`merge!` *alist blist less* [Scheme Procedure]
`scm_merge_x` (*alist, blist, less*) [C Function]

Takes two lists *alist* and *blist* such that `(sorted? alist less?)` and `(sorted? blist less?)` and returns a new list in which the elements of *alist* and *blist* have been stably interleaved so that `(sorted? (merge alist blist less?) less?)`. This is the destructive variant of `merge` Note: this does _not_ accept vectors.

The following procedures can operate on sequences which are either vectors or list. According to the given arguments, they return sorted vectors or lists, respectively. The first of the following procedures determines whether a sequence is already sorted, the other sort a given sequence. The variants with names starting with `stable-` are special in that they maintain a special property of the input sequences: If two or more elements are the same according to the comparison predicate, they are left in the same order as they appeared in the input.

`sorted?` *items less* [Scheme Procedure]
`scm_sorted_p` (*items, less*) [C Function]

Return `#t` iff *items* is a list or a vector such that for all 1 <= i <= m, the predicate *less* returns true when applied to all elements i - 1 and i

`sort` *items less* [Scheme Procedure]
`scm_sort` (*items, less*) [C Function]

Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. This is not a stable sort.

`sort!` *items less* [Scheme Procedure]
`scm_sort_x` (*items, less*) [C Function]

Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. The sorting is destructive, that means that the input sequence is modified to produce the sorted result. This is not a stable sort.

`stable-sort` *items less* [Scheme Procedure]
`scm_stable_sort` (*items, less*) [C Function]

Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. This is a stable sort.

`stable-sort!` *items less* [Scheme Procedure]
`scm_stable_sort_x` (*items, less*) [C Function]

Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. The sorting is destructive, that means that the input sequence is modified to produce the sorted result. This is a stable sort.

The procedures in the last group only accept lists or vectors as input, as their names indicate.

`sort-list` *items less* [Scheme Procedure]
`scm_sort_list` (*items, less*) [C Function]

Sort the list *items*, using *less* for comparing the list elements. This is a stable sort.

`sort-list!` *items less* [Scheme Procedure]
`scm_sort_list_x` (*items, less*) [C Function]

Sort the list *items*, using *less* for comparing the list elements. The sorting is destructive, that means that the input list is modified to produce the sorted result. This is a stable sort.

`restricted-vector-sort!` *vec less startpos endpos* [Scheme Procedure]
`scm_restricted_vector_sort_x` (*vec, less, startpos, endpos*) [C Function]

Sort the vector *vec*, using *less* for comparing the vector elements. *startpos* (inclusively) and *endpos* (exclusively) delimit the range of the vector which gets sorted. The return value is not specified.

5.9.4 Copying Deep Structures

The procedures for copying lists (see Section 5.6.2 [Lists], page 168) only produce a flat copy of the input list, and currently Guile does not even contain procedures for copying vectors. `copy-tree` can be used for these application, as it does not only copy the spine of a list, but also copies any pairs in the cars of the input lists.

`copy-tree` *obj* [Scheme Procedure]
`scm_copy_tree` (*obj*) [C Function]

Recursively copy the data tree that is bound to *obj*, and return a the new data structure. `copy-tree` recurses down the contents of both pairs and vectors (since both cons cells and vector cells may point to arbitrary objects), and stops recursing when it hits any other object.

5.9.5 General String Conversion

When debugging Scheme programs, but also for providing a human-friendly interface, a procedure for converting any Scheme object into string format is very useful. Conversion from/to strings can of course be done with specialized procedures when the data type of the object to convert is known, but with this procedure, it is often more comfortable.

`object->string` converts an object by using a print procedure for writing to a string port, and then returning the resulting string. Converting an object back from the string is only possible if the object type has a read syntax and the read syntax is preserved by the printing procedure.

`object->string` *obj* [*printer*] [Scheme Procedure]
`scm_object_to_string` (*obj, printer*) [C Function]

Return a Scheme string obtained by printing *obj*. Printing function can be specified by the optional second argument *printer* (default: `write`).

5.9.6 Hooks

A hook is a list of procedures to be called at well defined points in time. Typically, an application provides a hook *h* and promises its users that it will call all of the procedures in *h* at a defined point in the application's processing. By adding its own procedure to *h*, an application user can tap into or even influence the progress of the application.

Guile itself provides several such hooks for debugging and customization purposes: these are listed in a subsection below.

When an application first creates a hook, it needs to know how many arguments will be passed to the hook's procedures when the hook is run. The chosen number of arguments (which may be none) is declared when the hook is created, and all the procedures that are added to that hook must be capable of accepting that number of arguments.

A hook is created using `make-hook`. A procedure can be added to or removed from a hook using `add-hook!` or `remove-hook!`, and all of a hook's procedures can be removed together using `reset-hook!`. When an application wants to run a hook, it does so using `run-hook`.

5.9.6.1 Hook Usage by Example

Hook usage is shown by some examples in this section. First, we will define a hook of arity 2 — that is, the procedures stored in the hook will have to accept two arguments.

```
(define hook (make-hook 2))
hook
⇒ #<hook 2 40286c90>
```

Now we are ready to add some procedures to the newly created hook with `add-hook!`. In the following example, two procedures are added, which print different messages and do different things with their arguments.

```
(add-hook! hook (lambda (x y)
                    (display "Foo: ")
                    (display (+ x y))
                    (newline)))
(add-hook! hook (lambda (x y)
                    (display "Bar: ")
                    (display (* x y))
                    (newline)))
```

Once the procedures have been added, we can invoke the hook using `run-hook`.

```
(run-hook hook 3 4)
⊣ Bar: 12
⊣ Foo: 7
```

Note that the procedures are called in the reverse of the order with which they were added. This is because the default behaviour of `add-hook!` is to add its procedure to the *front* of the hook's procedure list. You can force `add-hook!` to add its procedure to the *end* of the list instead by providing a third `#t` argument on the second call to `add-hook!`.

```
(add-hook! hook (lambda (x y)
                    (display "Foo: ")
                    (display (+ x y))
                    (newline)))
(add-hook! hook (lambda (x y)
                    (display "Bar: ")
                    (display (* x y))
                    (newline))
                    #t)                ; <- Change here!

(run-hook hook 3 4)
```

```
⊣ Foo: 7
⊣ Bar: 12
```

5.9.6.2 Hook Reference

When you create a hook with `make-hook`, you must specify the arity of the procedures which can be added to the hook. If the arity is not given explicitly as an argument to `make-hook`, it defaults to zero. All procedures of a given hook must have the same arity, and when the procedures are invoked using `run-hook`, the number of arguments passed must match the arity specified at hook creation time.

The order in which procedures are added to a hook matters. If the third parameter to `add-hook!` is omitted or is equal to `#f`, the procedure is added in front of the procedures which might already be on that hook, otherwise the procedure is added at the end. The procedures are always called from the front to the end of the list when they are invoked via `run-hook`.

The ordering of the list of procedures returned by `hook->list` matches the order in which those procedures would be called if the hook was run using `run-hook`.

Note that the C functions in the following entries are for handling *Scheme-level* hooks in C. There are also *C-level* hooks which have their own interface (see Section 5.9.6.4 [C Hooks], page 244).

`make-hook` [*n_args*] [Scheme Procedure]
`scm_make_hook` (*n_args*) [C Function]
: Create a hook for storing procedure of arity *n_args*. *n_args* defaults to zero. The returned value is a hook object to be used with the other hook procedures.

`hook?` *x* [Scheme Procedure]
`scm_hook_p` (*x*) [C Function]
: Return `#t` if *x* is a hook, `#f` otherwise.

`hook-empty?` *hook* [Scheme Procedure]
`scm_hook_empty_p` (*hook*) [C Function]
: Return `#t` if *hook* is an empty hook, `#f` otherwise.

`add-hook!` *hook proc* [*append_p*] [Scheme Procedure]
`scm_add_hook_x` (*hook, proc, append_p*) [C Function]
: Add the procedure *proc* to the hook *hook*. The procedure is added to the end if *append_p* is true, otherwise it is added to the front. The return value of this procedure is not specified.

`remove-hook!` *hook proc* [Scheme Procedure]
`scm_remove_hook_x` (*hook, proc*) [C Function]
: Remove the procedure *proc* from the hook *hook*. The return value of this procedure is not specified.

`reset-hook!` *hook* [Scheme Procedure]
`scm_reset_hook_x` (*hook*) [C Function]
: Remove all procedures from the hook *hook*. The return value of this procedure is not specified.

`hook->list` *hook* [Scheme Procedure]
`scm_hook_to_list` (*hook*) [C Function]
Convert the procedure list of *hook* to a list.

`run-hook` *hook . args* [Scheme Procedure]
`scm_run_hook` (*hook, args*) [C Function]
Apply all procedures from the hook *hook* to the arguments *args*. The order of the procedure application is first to last. The return value of this procedure is not specified.

If, in C code, you are certain that you have a hook object and well formed argument list for that hook, you can also use `scm_c_run_hook`, which is identical to `scm_run_hook` but does no type checking.

`void scm_c_run_hook` (*SCM hook, SCM args*) [C Function]
The same as `scm_run_hook` but without any type checking to confirm that *hook* is actually a hook object and that *args* is a well-formed list matching the arity of the hook.

For C code, `SCM_HOOKP` is a faster alternative to `scm_hook_p`:

`int SCM_HOOKP` (*x*) [C Macro]
Return 1 if *x* is a Scheme-level hook, 0 otherwise.

5.9.6.3 Handling Scheme-level hooks from C code

Here is an example of how to handle Scheme-level hooks from C code using the above functions.

```
if (scm_is_true (scm_hook_p (obj)))
  /* handle Scheme-level hook using C functions */
  scm_reset_hook_x (obj);
else
  /* do something else (obj is not a hook) */
```

5.9.6.4 Hooks For C Code.

The hooks already described are intended to be populated by Scheme-level procedures. In addition to this, the Guile library provides an independent set of interfaces for the creation and manipulation of hooks that are designed to be populated by functions implemented in C.

The original motivation here was to provide a kind of hook that could safely be invoked at various points during garbage collection. Scheme-level hooks are unsuitable for this purpose as running them could itself require memory allocation, which would then invoke garbage collection recursively . . . However, it is also the case that these hooks are easier to work with than the Scheme-level ones if you only want to register C functions with them. So if that is mainly what your code needs to do, you may prefer to use this interface.

To create a C hook, you should allocate storage for a structure of type `scm_t_c_hook` and then initialize it using `scm_c_hook_init`.

`scm_t_c_hook` [C Type]
Data type for a C hook. The internals of this type should be treated as opaque.

`scm_t_c_hook_type` [C Enum]

Enumeration of possible hook types, which are:

`SCM_C_HOOK_NORMAL`
: Type of hook for which all the registered functions will always be called.

`SCM_C_HOOK_OR`
: Type of hook for which the sequence of registered functions will be called only until one of them returns C true (a non-NULL pointer).

`SCM_C_HOOK_AND`
: Type of hook for which the sequence of registered functions will be called only until one of them returns C false (a NULL pointer).

`void scm_c_hook_init` (*scm_t_c_hook *hook, void *hook_data, scm_t_c_hook_type type*) [C Function]

Initialize the C hook at memory pointed to by *hook*. *type* should be one of the values of the `scm_t_c_hook_type` enumeration, and controls how the hook functions will be called. *hook_data* is a closure parameter that will be passed to all registered hook functions when they are called.

To add or remove a C function from a C hook, use `scm_c_hook_add` or `scm_c_hook_remove`. A hook function must expect three `void *` parameters which are, respectively:

hook_data The hook closure data that was specified at the time the hook was initialized by `scm_c_hook_init`.

func_data The function closure data that was specified at the time that that function was registered with the hook by `scm_c_hook_add`.

data The call closure data specified by the `scm_c_hook_run` call that runs the hook.

`scm_t_c_hook_function` [C Type]

Function type for a C hook function: takes three `void *` parameters and returns a `void *` result.

`void scm_c_hook_add` (*scm_t_c_hook *hook, scm_t_c_hook_function func, void *func_data, int appendp*) [C Function]

Add function *func*, with function closure data *func_data*, to the C hook *hook*. The new function is appended to the hook's list of functions if *appendp* is non-zero, otherwise prepended.

`void scm_c_hook_remove` (*scm_t_c_hook *hook, scm_t_c_hook_function func, void *func_data*) [C Function]

Remove function *func*, with function closure data *func_data*, from the C hook *hook*. `scm_c_hook_remove` checks both *func* and *func_data* so as to allow for the same *func* being registered multiple times with different closure data.

Finally, to invoke a C hook, call the `scm_c_hook_run` function specifying the hook and the call closure data for this run:

`void * scm_c_hook_run` (*scm_t_c_hook *hook, void *data*) [C Function]

Run the C hook *hook* will call closure data *data*. Subject to the variations for hook types `SCM_C_HOOK_OR` and `SCM_C_HOOK_AND`, `scm_c_hook_run` calls *hook*'s registered functions in turn, passing them the hook's closure data, each function's closure data, and the call closure data.

`scm_c_hook_run`'s return value is the return value of the last function to be called.

5.9.6.5 Hooks for Garbage Collection

Whenever Guile performs a garbage collection, it calls the following hooks in the order shown.

`scm_before_gc_c_hook` [C Hook]

C hook called at the very start of a garbage collection, after setting `scm_gc_running_p` to 1, but before entering the GC critical section.

If garbage collection is blocked because `scm_block_gc` is non-zero, GC exits early soon after calling this hook, and no further hooks will be called.

`scm_before_mark_c_hook` [C Hook]

C hook called before beginning the mark phase of garbage collection, after the GC thread has entered a critical section.

`scm_before_sweep_c_hook` [C Hook]

C hook called before beginning the sweep phase of garbage collection. This is the same as at the end of the mark phase, since nothing else happens between marking and sweeping.

`scm_after_sweep_c_hook` [C Hook]

C hook called after the end of the sweep phase of garbage collection, but while the GC thread is still inside its critical section.

`scm_after_gc_c_hook` [C Hook]

C hook called at the very end of a garbage collection, after the GC thread has left its critical section.

`after-gc-hook` [Scheme Hook]

Scheme hook with arity 0. This hook is run asynchronously (see Section 5.17.2 [Asyncs], page 323) soon after the GC has completed and any other events that were deferred during garbage collection have been processed. (Also accessible from C with the name `scm_after_gc_hook`.)

All the C hooks listed here have type `SCM_C_HOOK_NORMAL`, are initialized with hook closure data NULL, are are invoked by `scm_c_hook_run` with call closure data NULL.

The Scheme hook `after-gc-hook` is particularly useful in conjunction with guardians (see Section 5.14.4 [Guardians], page 301). Typically, if you are using a guardian, you want to call the guardian after garbage collection to see if any of the objects added to the guardian have been collected. By adding a thunk that performs this call to `after-gc-hook`, you can ensure that your guardian is tested after every garbage collection cycle.

5.9.6.6 Hooks into the Guile REPL

5.10 Definitions and Variable Bindings

Scheme supports the definition of variables in different contexts. Variables can be defined at the top level, so that they are visible in the entire program, and variables can be defined locally to procedures and expressions. This is important for modularity and data abstraction.

5.10.1 Top Level Variable Definitions

On the top level of a program (i.e. when not inside the body of a procedure definition or a `let`, `let*` or `letrec` expression), a definition of the form

```
(define a value)
```

defines a variable called `a` and sets it to the value *value*.

If the variable already exists, because it has already been created by a previous `define` expression with the same name, its value is simply changed to the new *value*. In this case, then, the above form is completely equivalent to

```
(set! a value)
```

This equivalence means that `define` can be used interchangeably with `set!` to change the value of variables at the top level of the REPL or a Scheme source file. It is useful during interactive development when reloading a Scheme file that you have modified, because it allows the `define` expressions in that file to work as expected both the first time that the file is loaded and on subsequent occasions.

Note, though, that `define` and `set!` are not always equivalent. For example, a `set!` is not allowed if the named variable does not already exist, and the two expressions can behave differently in the case where there are imported variables visible from another module.

`define` *name value* [Scheme Syntax]

Create a top level variable named *name* with value *value*. If the named variable already exists, just change its value. The return value of a `define` expression is unspecified.

The C API equivalents of `define` are `scm_define` and `scm_c_define`, which differ from each other in whether the variable name is specified as a `SCM` symbol or as a null-terminated C string.

`scm_define` (*sym, value*) [C Function]
`scm_c_define` (*const char *name, value*) [C Function]

C equivalents of `define`, with variable name specified either by *sym*, a symbol, or by *name*, a null-terminated C string. Both variants return the new or preexisting variable object.

`define` (when it occurs at top level), `scm_define` and `scm_c_define` all create or set the value of a variable in the top level environment of the current module. If there was not already a variable with the specified name belonging to the current module, but a similarly named variable from another module was visible through having been imported, the newly created variable in the current module will shadow the imported variable, such that the imported variable is no longer visible.

Attention: Scheme definitions inside local binding constructs (see Section 5.10.2 [Local Bindings], page 248) act differently (see Section 5.10.3 [Internal Definitions], page 249).

5.10.2 Local Variable Bindings

As opposed to definitions at the top level, which are visible in the whole program (or current module, when Guile modules are used), it is also possible to define variables which are only visible in a well-defined part of the program. Normally, this part of a program will be a procedure or a subexpression of a procedure.

With the constructs for local binding (`let`, `let*` and `letrec`), the Scheme language has a block structure like most other programming languages since the days of Algol 60. Readers familiar to languages like C or Java should already be used to this concept, but the family of `let` expressions has a few properties which are well worth knowing.

The first local binding construct is `let`. The other constructs `let*` and `letrec` are specialized versions for usage where using plain `let` is a bit inconvenient.

`let` *bindings body* [syntax]

bindings has the form

```
((variable1 init1) ...)
```

that is zero or more two-element lists of a variable and an arbitrary expression each. All *variable* names must be distinct.

A `let` expression is evaluated as follows.

- All *init* expressions are evaluated.
- New storage is allocated for the *variables*.
- The values of the *init* expressions are stored into the variables.
- The expressions in *body* are evaluated in order, and the value of the last expression is returned as the value of the `let` expression.
- The storage for the *variables* is freed.

The *init* expressions are not allowed to refer to any of the *variables*.

`let*` *bindings body* [syntax]

Similar to `let`, but the variable bindings are performed sequentially, that means that all *init* expression are allowed to use the variables defined on their left in the binding list.

A `let*` expression can always be expressed with nested `let` expressions.

```
(let* ((a 1) (b a))
   b)
≡
(let ((a 1))
  (let ((b a))
    b))
```

`letrec` *bindings body* [syntax]

Similar to `let`, but it is possible to refer to the *variable* from lambda expression created in any of the *inits*. That is, procedures created in the *init* expression can recursively refer to the defined variables.

```
(letrec ((even?
          (lambda (n)
                   (if (zero? n)
                       #t
                       (odd? (- n 1)))))
              (odd?
               (lambda (n)
                   (if (zero? n)
                       #f
                       (even? (- n 1))))))
       (even? 88))
     ⇒
     #t
```

There is also an alternative form of the `let` form, which is used for expressing iteration. Because of the use as a looping construct, this form (the *named let*) is documented in the section about iteration (see Section 5.11.4 [while do], page 252)

5.10.3 Internal definitions

A `define` form which appears inside the body of a `lambda`, `let`, `let*`, `letrec` or equivalent expression is called an *internal definition*. An internal definition differs from a top level definition (see Section 5.10.1 [Top Level], page 247), because the definition is only visible inside the complete body of the enclosing form. Let us examine the following example.

```
(let ((frumble "froz"))
   (define banana (lambda () (apple 'peach)))
   (define apple (lambda (x) x))
   (banana))
⇒
peach
```

Here the enclosing form is a `let`, so the `define`s in the `let`-body are internal definitions. Because the scope of the internal definitions is the **complete** body of the `let`-expression, the `lambda`-expression which gets bound to the variable `banana` may refer to the variable `apple`, even though it's definition appears lexically *after* the definition of `banana`. This is because a sequence of internal definition acts as if it were a `letrec` expression.

```
(let ()
  (define a 1)
  (define b 2)
  (+ a b))
```

is equivalent to

```
(let ()
  (letrec ((a 1) (b 2))
    (+ a b)))
```

Another noteworthy difference to top level definitions is that within one group of internal definitions all variable names must be distinct. That means where on the top level a second define for a given variable acts like a `set!`, an exception is thrown for internal definitions with duplicate bindings.

5.10.4 Querying variable bindings

Guile provides a procedure for checking whether a symbol is bound in the top level environment.

`defined?` *sym* [*env*] [Scheme Procedure]
`scm_defined_p` (*sym*, *env*) [C Function]

Return `#t` if *sym* is defined in the lexical environment *env*. When *env* is not specified, look in the top-level environment as defined by the current module.

5.11 Controlling the Flow of Program Execution

See Section 4.3.3 [Control Flow], page 63 for a discussion of how the more general control flow of Scheme affects C code.

5.11.1 Evaluating a Sequence of Expressions

The `begin` syntax is used for grouping several expressions together so that they are treated as if they were one expression. This is particularly important when syntactic expressions are used which only allow one expression, but the programmer wants to use more than one expression in that place. As an example, consider the conditional expression below:

```
(if (> x 0)
    (begin (display "greater") (newline)))
```

If the two calls to `display` and `newline` were not embedded in a `begin`-statement, the call to `newline` would get misinterpreted as the else-branch of the `if`-expression.

`begin` *expr1 expr2 . . .* [syntax]

The expression(s) are evaluated in left-to-right order and the value of the last expression is returned as the value of the `begin`-expression. This expression type is used when the expressions before the last one are evaluated for their side effects.

Guile also allows the expression `(begin)`, a `begin` with no sub-expressions. Such an expression returns the 'unspecified' value.

5.11.2 Simple Conditional Evaluation

Guile provides three syntactic constructs for conditional evaluation. `if` is the normal if-then-else expression (with an optional else branch), `cond` is a conditional expression with multiple branches and `case` branches if an expression has one of a set of constant values.

`if` *test consequent* [*alternate*] [syntax]

All arguments may be arbitrary expressions. First, *test* is evaluated. If it returns a true value, the expression *consequent* is evaluated and *alternate* is ignored. If *test* evaluates to `#f`, *alternate* is evaluated instead. The value of the evaluated branch (*consequent* or *alternate*) is returned as the value of the `if` expression.

When *alternate* is omitted and the *test* evaluates to `#f`, the value of the expression is not specified.

`cond` *clause1 clause2 . . .* [syntax]

Each `cond`-clause must look like this:

```
(test expression ...)
```

where *test* and *expression* are arbitrary expression, or like this

```
(test => expression)
```

where *expression* must evaluate to a procedure.

The *test*s of the clauses are evaluated in order and as soon as one of them evaluates to a true values, the corresponding *expression*s are evaluated in order and the last value is returned as the value of the `cond`-expression. For the `=>` clause type, *expression* is evaluated and the resulting procedure is applied to the value of *test*. The result of this procedure application is then the result of the `cond`-expression.

One additional `cond`-clause is available as an extension to standard Scheme:

```
(test guard => expression)
```

where *guard* and *expression* must evaluate to procedures. For this clause type, *test* may return multiple values, and `cond` ignores its boolean state; instead, `cond` evaluates *guard* and applies the resulting procedure to the value(s) of *test*, as if *guard* were the *consumer* argument of `call-with-values`. Iff the result of that procedure call is a true value, it evaluates *expression* and applies the resulting procedure to the value(s) of *test*, in the same manner as the *guard* was called.

The *test* of the last *clause* may be the symbol `else`. Then, if none of the preceding *tests* is true, the *expressions* following the `else` are evaluated to produce the result of the `cond`-expression.

`case` *key clause1 clause2 . . .* [syntax]

key may be any expression, the *clauses* must have the form

```
((datum1 ...) expr1 expr2 ...)
```

and the last *clause* may have the form

```
(else expr1 expr2 ...)
```

All *datums* must be distinct. First, *key* is evaluated. The the result of this evaluation is compared against all *datums* using `eqv?`. When this comparison succeeds, the expression(s) following the *datum* are evaluated from left to right, returning the value of the last expression as the result of the `case` expression.

If the *key* matches no *datum* and there is an `else`-clause, the expressions following the `else` are evaluated. If there is no such clause, the result of the expression is unspecified.

5.11.3 Conditional Evaluation of a Sequence of Expressions

`and` and `or` evaluate all their arguments in order, similar to `begin`, but evaluation stops as soon as one of the expressions evaluates to false or true, respectively.

`and` *expr . . .* [syntax]

Evaluate the *exprs* from left to right and stop evaluation as soon as one expression evaluates to `#f`; the remaining expressions are not evaluated. The value of the last evaluated expression is returned. If no expression evaluates to `#f`, the value of the last expression is returned.

If used without expressions, `#t` is returned.

`or` *expr . . .* [syntax]

Evaluate the *exprs* from left to right and stop evaluation as soon as one expression evaluates to a true value (that is, a value different from `#f`); the remaining expressions are not evaluated. The value of the last evaluated expression is returned. If all expressions evaluate to `#f`, `#f` is returned.

If used without expressions, `#f` is returned.

5.11.4 Iteration mechanisms

Scheme has only few iteration mechanisms, mainly because iteration in Scheme programs is normally expressed using recursion. Nevertheless, R5RS defines a construct for programming loops, calling `do`. In addition, Guile has an explicit looping syntax called `while`.

`do` ((*variable init* [*step*]) . . .) (*test* [*expr* . . .]) *body* . . . [syntax]

Bind *variable*s and evaluate *body* until *test* is true. The return value is the last *expr* after *test*, if given. A simple example will illustrate the basic form,

```
(do ((i 1 (1+ i)))
    ((> i 4))
  (display i))
⊣ 1234
```

Or with two variables and a final return value,

```
(do ((i 1 (1+ i))
     (p 3 (* 3 p)))
    ((> i 4)
     p)
  (format #t "3**~s is ~s\n" i p))
⊣
3**1 is 3
3**2 is 9
3**3 is 27
3**4 is 81
⇒
789
```

The *variable* bindings are established like a `let`, in that the expressions are all evaluated and then all bindings made. When iterating, the optional *step* expressions are evaluated with the previous bindings in scope, then new bindings all made.

The *test* expression is a termination condition. Looping stops when the *test* is true. It's evaluated before running the *body* each time, so if it's true the first time then *body* is not run at all.

The optional *expr*s after the *test* are evaluated at the end of looping, with the final *variable* bindings available. The last *expr* gives the return value, or if there are no *expr*s the return value is unspecified.

Each iteration establishes bindings to fresh locations for the *variable*s, like a new `let` for each iteration. This is done for *variable*s without *step* expressions too. The following illustrates this, showing how a new `i` is captured by the `lambda` in each iteration (see Section 3.1.4 [The Concept of Closure], page 24).

```
(define lst '())
(do ((i 1 (1+ i)))
    ((> i 4))
  (set! lst (cons (lambda () i) lst)))
(map (lambda (proc) (proc)) lst)
⇒
(4 3 2 1)
```

`while` *cond body* . . . [syntax]

Run a loop executing the *body* forms while *cond* is true. *cond* is tested at the start of each iteration, so if it's `#f` the first time then *body* is not executed at all. The return value is unspecified.

Within `while`, two extra bindings are provided, they can be used from both *cond* and *body*.

`break` [Scheme Procedure]
Break out of the `while` form.

`continue` [Scheme Procedure]
Abandon the current iteration, go back to the start and test *cond* again, etc.

Each `while` form gets its own `break` and `continue` procedures, operating on that `while`. This means when loops are nested the outer `break` can be used to escape all the way out. For example,

```
(while (test1)
  (let ((outer-break break))
    (while (test2)
      (if (something)
        (outer-break #f))
      ...)))
```

Note that each `break` and `continue` procedure can only be used within the dynamic extent of its `while`. Outside the `while` their behaviour is unspecified.

Another very common way of expressing iteration in Scheme programs is the use of the so-called *named let*.

Named let is a variant of `let` which creates a procedure and calls it in one step. Because of the newly created procedure, named let is more powerful than `do`–it can be used for iteration, but also for arbitrary recursion.

`let` *variable bindings body* [syntax]
For the definition of *bindings* see the documentation about `let` (see Section 5.10.2 [Local Bindings], page 248).

Named `let` works as follows:

- A new procedure which accepts as many arguments as are in *bindings* is created and bound locally (using `let`) to *variable*. The new procedure's formal argument names are the name of the *variables*.
- The *body* expressions are inserted into the newly created procedure.
- The procedure is called with the *init* expressions as the formal arguments.

The next example implements a loop which iterates (by recursion) 1000 times.

```
(let lp ((x 1000))
  (if (positive? x)
      (lp (- x 1))
      x))
⇒
0
```

5.11.5 Continuations

A "continuation" is the code that will execute when a given function or expression returns. For example, consider

```
(define (foo)
  (display "hello\n")
  (display (bar)) (newline)
  (exit))
```

The continuation from the call to `bar` comprises a `display` of the value returned, a `newline` and an `exit`. This can be expressed as a function of one argument.

```
(lambda (r)
  (display r) (newline)
  (exit))
```

In Scheme, continuations are represented as special procedures just like this. The special property is that when a continuation is called it abandons the current program location and jumps directly to that represented by the continuation.

A continuation is like a dynamic label, capturing at run-time a point in program execution, including all the nested calls that have lead to it (or rather the code that will execute when those calls return).

Continuations are created with the following functions.

`call-with-current-continuation` *proc* [Scheme Procedure]
`call/cc` *proc* [Scheme Procedure]

Capture the current continuation and call `(`*proc cont*`)` with it. The return value is the value returned by *proc*, or when `(`*cont value*`)` is later invoked, the return is the *value* passed.

Normally *cont* should be called with one argument, but when the location resumed is expecting multiple values (see Section 5.11.6 [Multiple Values], page 257) then they should be passed as multiple arguments, for instance `(`*cont x y z*`)`.

cont may only be used from the same side of a continuation barrier as it was created (see Section 5.17.3 [Continuation Barriers], page 325), and in a multi-threaded program only from the thread in which it was created.

The call to *proc* is not part of the continuation captured, it runs only when the continuation is created. Often a program will want to store *cont* somewhere for later use; this can be done in *proc*.

The `call` in the name `call-with-current-continuation` refers to the way a call to *proc* gives the newly created continuation. It's not related to the way a call is used later to invoke that continuation.

`call/cc` is an alias for `call-with-current-continuation`. This is in common use since the latter is rather long.

`SCM scm_make_continuation` (*int *first*) [C Function]

Capture the current continuation as described above. The return value is the new continuation, and **first* is set to 1.

When the continuation is invoked, `scm_make_continuation` will return again, this time returning the value (or set of multiple values) passed in that invocation, and with **first* set to 0.

Here is a simple example,

```
(define kont #f)
(format #t "the return is ~a\n"
        (call/cc (lambda (k)
                   (set! kont k)
                   1)))
⇒ the return is 1

(kont 2)
⇒ the return is 2
```

`call/cc` captures a continuation in which the value returned is going to be displayed by `format`. The `lambda` stores this in `kont` and gives an initial return `1` which is displayed. The later invocation of `kont` resumes the captured point, but this time returning `2`, which is displayed.

When Guile is run interactively, a call to `format` like this has an implicit return back to the read-eval-print loop. `call/cc` captures that like any other return, which is why interactively `kont` will come back to read more input.

C programmers may note that `call/cc` is like `setjmp` in the way it records at runtime a point in program execution. A call to a continuation is like a `longjmp` in that it abandons the present location and goes to the recorded one. Like `longjmp`, the value passed to the continuation is the value returned by `call/cc` on resuming there. However `longjmp` can only go up the program stack, but the continuation mechanism can go anywhere.

When a continuation is invoked, `call/cc` and subsequent code effectively "returns" a second time. It can be confusing to imagine a function returning more times than it was called. It may help instead to think of it being stealthily re-entered and then program flow going on as normal.

`dynamic-wind` (see Section 5.11.9 [Dynamic Wind], page 266) can be used to ensure setup and cleanup code is run when a program locus is resumed or abandoned through the continuation mechanism.

Continuations are a powerful mechanism, and can be used to implement almost any sort of control structure, such as loops, coroutines, or exception handlers.

However the implementation of continuations in Guile is not as efficient as one might hope, because Guile is designed to cooperate with programs written in other languages, such as C, which do not know about continuations. Basically continuations are captured by a block copy of the stack, and resumed by copying back.

For this reason, generally continuations should be used only when there is no other simple way to achieve the desired result, or when the elegance of the continuation mechanism outweighs the need for performance.

Escapes upwards from loops or nested functions are generally best handled with exceptions (see Section 5.11.7 [Exceptions], page 258). Coroutines can be efficiently implemented with cooperating threads (a thread holds a full program stack but doesn't copy it around the way continuations do).

5.11.6 Returning and Accepting Multiple Values

Scheme allows a procedure to return more than one value to its caller. This is quite different to other languages which only allow single-value returns. Returning multiple values is different from returning a list (or pair or vector) of values to the caller, because conceptually not *one* compound object is returned, but several distinct values.

The primitive procedures for handling multiple values are `values` and `call-with-values`. `values` is used for returning multiple values from a procedure. This is done by placing a call to `values` with zero or more arguments in tail position in a procedure body. `call-with-values` combines a procedure returning multiple values with a procedure which accepts these values as parameters.

`values` *arg1 . . . argN* [Scheme Procedure]
`scm_values` (*args*) [C Function]

Delivers all of its arguments to its continuation. Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value. The effect of passing no value or more than one value to continuations that were not created by `call-with-values` is unspecified.

For `scm_values`, *args* is a list of arguments and the return is a multiple-values object which the caller can return. In the current implementation that object shares structure with *args*, so *args* should not be modified subsequently.

`call-with-values` *producer consumer* [Scheme Procedure]

Calls its *producer* argument with no values and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
⇒ 5


(call-with-values * -)
⇒ -1
```

In addition to the fundamental procedures described above, Guile has a module which exports a syntax called `receive`, which is much more convenient. This is in the `(ice-9 receive)` and is the same as specified by SRFI-8 (see Section 6.4.7 [SRFI-8], page 439).

```
(use-modules (ice-9 receive))
```

`receive` *formals expr body . . .* [library syntax]

Evaluate the expression *expr*, and bind the result values (zero or more) to the formal arguments in *formals*. *formals* is a list of symbols, like the argument list in a `lambda` (see Section 5.8.1 [Lambda], page 225). After binding the variables, the expressions in *body* ... are evaluated in order, the return value is the result from the last expression.

For example getting results from `partition` in SRFI-1 (see Section 6.4.3 [SRFI-1], page 424),

```
(receive (odds evens)
    (partition odd? '(7 4 2 8 3))
  (display odds)
  (display " and ")
  (display evens))
⊣ (7 3) and (4 2 8)
```

5.11.7 Exceptions

A common requirement in applications is to want to jump *non-locally* from the depths of a computation back to, say, the application's main processing loop. Usually, the place that is the target of the jump is somewhere in the calling stack of procedures that called the procedure that wants to jump back. For example, typical logic for a key press driven application might look something like this:

```
main-loop:
  read the next key press and call dispatch-key

dispatch-key:
  lookup the key in a keymap and call an appropriate procedure,
  say find-file

find-file:
  interactively read the required file name, then call
  find-specified-file

find-specified-file:
  check whether file exists; if not, jump back to main-loop
  ...
```

The jump back to `main-loop` could be achieved by returning through the stack one procedure at a time, using the return value of each procedure to indicate the error condition, but Guile (like most modern programming languages) provides an additional mechanism called *exception handling* that can be used to implement such jumps much more conveniently.

5.11.7.1 Exception Terminology

There are several variations on the terminology for dealing with non-local jumps. It is useful to be aware of them, and to realize that they all refer to the same basic mechanism.

- Actually making a non-local jump may be called *raising an exception*, *raising a signal*, *throwing an exception* or *doing a long jump*. When the jump indicates an error condition, people may talk about *signalling*, *raising* or *throwing an error*.
- Handling the jump at its target may be referred to as *catching* or *handling* the *exception*, *signal* or, where an error condition is involved, *error*.

Where *signal* and *signalling* are used, special care is needed to avoid the risk of confusion with POSIX signals.

This manual prefers to speak of throwing and catching exceptions, since this terminology matches the corresponding Guile primitives.

5.11.7.2 Catching Exceptions

`catch` is used to set up a target for a possible non-local jump. The arguments of a `catch` expression are a *key*, which restricts the set of exceptions to which this `catch` applies, a thunk that specifies the code to execute and one or two *handler* procedures that say what to do if an exception is thrown while executing the code. If the execution thunk executes *normally*, which means without throwing any exceptions, the handler procedures are not called at all.

When an exception is thrown using the `throw` function, the first argument of the `throw` is a symbol that indicates the type of the exception. For example, Guile throws an exception using the symbol `numerical-overflow` to indicate numerical overflow errors such as division by zero:

```
(/ 1 0)
⇒
ABORT: (numerical-overflow)
```

The *key* argument in a `catch` expression corresponds to this symbol. *key* may be a specific symbol, such as `numerical-overflow`, in which case the `catch` applies specifically to exceptions of that type; or it may be `#t`, which means that the `catch` applies to all exceptions, irrespective of their type.

The second argument of a `catch` expression should be a thunk (i.e. a procedure that accepts no arguments) that specifies the normal case code. The `catch` is active for the execution of this thunk, including any code called directly or indirectly by the thunk's body. Evaluation of the `catch` expression activates the catch and then calls this thunk.

The third argument of a `catch` expression is a handler procedure. If an exception is thrown, this procedure is called with exactly the arguments specified by the `throw`. Therefore, the handler procedure must be designed to accept a number of arguments that corresponds to the number of arguments in all `throw` expressions that can be caught by this `catch`.

The fourth, optional argument of a `catch` expression is another handler procedure, called the *pre-unwind* handler. It differs from the third argument in that if an exception is thrown, it is called, *before* the third argument handler, in exactly the dynamic context of the `throw` expression that threw the exception. This means that it is useful for capturing or displaying the stack at the point of the `throw`, or for examining other aspects of the dynamic context, such as fluid values, before the context is unwound back to that of the prevailing `catch`.

`catch` *key thunk handler* [*pre-unwind-handler*] [Scheme Procedure]
`scm_catch_with_pre_unwind_handler` (*key, thunk, handler, pre_unwind_handler*) [C Function]
`scm_catch` (*key, thunk, handler*) [C Function]

Invoke *thunk* in the dynamic context of *handler* for exceptions matching *key*. If thunk throws to the symbol *key*, then *handler* is invoked this way:

```
(handler key args ...)
```

key is a symbol or `#t`.

thunk takes no arguments. If *thunk* returns normally, that is the return value of `catch`.

Handler is invoked outside the scope of its own `catch`. If *handler* again throws to the same key, a new handler from further up the call chain is invoked.

If the key is `#t`, then a throw to *any* symbol will match this call to `catch`.

If a *pre-unwind-handler* is given and *thunk* throws an exception that matches *key*, Guile calls the *pre-unwind-handler* before unwinding the dynamic state and invoking the main *handler*. *pre-unwind-handler* should be a procedure with the same signature as *handler*, that is `(lambda (key . args))`. It is typically used to save the stack at the point where the exception occurred, but can also query other parts of the dynamic state at that point, such as fluid values.

A *pre-unwind-handler* can exit either normally or non-locally. If it exits normally, Guile unwinds the stack and dynamic context and then calls the normal (third argument) handler. If it exits non-locally, that exit determines the continuation.

If a handler procedure needs to match a variety of `throw` expressions with varying numbers of arguments, you should write it like this:

```
(lambda (key . args)
  ...)
```

The *key* argument is guaranteed always to be present, because a `throw` without a *key* is not valid. The number and interpretation of the *args* varies from one type of exception to another, but should be specified by the documentation for each exception type.

Note that, once the normal (post-unwind) handler procedure is invoked, the catch that led to the handler procedure being called is no longer active. Therefore, if the handler procedure itself throws an exception, that exception can only be caught by another active catch higher up the call stack, if there is one.

`SCM scm_c_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, scm_t_catch_handler pre_unwind_handler, void *pre_unwind_handler_data*) [C Function]

`SCM scm_internal_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]

The above `scm_catch_with_pre_unwind_handler` and `scm_catch` take Scheme procedures as body and handler arguments. `scm_c_catch` and `scm_internal_catch` are equivalents taking C functions.

body is called as ***`body (body_data)`*** with a catch on exceptions of the given *tag* type. If an exception is caught, *pre_unwind_handler* and *handler* are called as ***`handler (handler_data, key, args)`***. *key* and *args* are the `SCM` key and argument list from the `throw`.

body and *handler* should have the following prototypes. `scm_t_catch_body` and `scm_t_catch_handler` are pointer typedefs for these.

```
SCM body (void *data);
SCM handler (void *data, SCM key, SCM args);
```

The *body_data* and *handler_data* parameters are passed to the respective calls so an application can communicate extra information to those functions.

If the data consists of an `SCM` object, care should be taken that it isn't garbage collected while still required. If the `SCM` is a local C variable, one way to protect it is to pass a pointer to that variable as the data parameter, since the C compiler will then know the value must be held on the stack. Another way is to use `scm_remember_upto_here_1` (see Section 4.4.6 [Remembering During Operations], page 75).

5.11.7.3 Throw Handlers

It's sometimes useful to be able to intercept an exception that is being thrown, but without changing where in the dynamic context that exception will eventually be caught. This could be to clean up some related state or to pass information about the exception to a debugger, for example. The `with-throw-handler` procedure provides a way to do this.

`with-throw-handler` *key thunk handler* [Scheme Procedure]
`scm_with_throw_handler` (*key*, *thunk*, *handler*) [C Function]

Add *handler* to the dynamic context as a throw handler for key *key*, then invoke *thunk*.

`SCM scm_c_with_throw_handler` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, int lazy_catch_p*) [C Function]

The above `scm_with_throw_handler` takes Scheme procedures as body (thunk) and handler arguments. `scm_c_with_throw_handler` is an equivalent taking C functions. See `scm_c_catch` (see Section 5.11.7.2 [Catch], page 259) for a description of the parameters, the behaviour however of course follows `with-throw-handler`.

If *thunk* throws an exception, Guile handles that exception by invoking the innermost `catch` or throw handler whose key matches that of the exception. When the innermost thing is a throw handler, Guile calls the specified handler procedure using `(apply handler key args)`. The handler procedure may either return normally or exit non-locally. If it returns normally, Guile passes the exception on to the next innermost `catch` or throw handler. If it exits non-locally, that exit determines the continuation.

The behaviour of a throw handler is very similar to that of a `catch` expression's optional pre-unwind handler. In particular, a throw handler's handler procedure is invoked in the exact dynamic context of the `throw` expression, just as a pre-unwind handler is. `with-throw-handler` may be seen as a half-`catch`: it does everything that a `catch` would do until the point where `catch` would start unwinding the stack and dynamic context, but then it rethrows to the next innermost `catch` or throw handler instead.

5.11.7.4 Catch Without Unwinding

Before version 1.8, Guile's closest equivalent to `with-throw-handler` was `lazy-catch`. From version 1.8 onwards we recommend using `with-throw-handler` because its behaviour is more useful than that of `lazy-catch`, but `lazy-catch` is still supported as well.

A *lazy catch* is used in the same way as a normal `catch`, with *key*, *thunk* and *handler* arguments specifying the exception type, normal case code and handler procedure, but differs in one important respect: the handler procedure is executed without unwinding the call stack from the context of the `throw` expression that caused the handler to be invoked.

`lazy-catch` *key thunk handler* [Scheme Procedure]
`scm_lazy_catch` (*key, thunk, handler*) [C Function]

This behaves exactly like `catch`, except that it does not unwind the stack before invoking *handler*. If the *handler* procedure returns normally, Guile rethrows the same exception again to the next innermost catch, lazy-catch or throw handler. If the *handler* exits non-locally, that exit determines the continuation.

`SCM scm_internal_lazy_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]

The above `scm_lazy_catch` takes Scheme procedures as body and handler arguments. `scm_internal_lazy_catch` is an equivalent taking C functions. See `scm_internal_catch` (see Section 5.11.7.2 [Catch], page 259) for a description of the parameters, the behaviour however of course follows `lazy-catch`.

Typically *handler* is used to display a backtrace of the stack at the point where the corresponding `throw` occurred, or to save off this information for possible display later.

Not unwinding the stack means that throwing an exception that is caught by a `lazy-catch` is *almost* equivalent to calling the `lazy-catch`'s handler inline instead of each `throw`, and then omitting the surrounding `lazy-catch`. In other words,

```
(lazy-catch 'key
  (lambda () ... (throw 'key args ...) ...)
  handler)
```

is *almost* equivalent to

```
((lambda () ... (handler 'key args ...) ...))
```

But why only *almost*? The difference is that with `lazy-catch` (as with normal `catch`), the dynamic context is unwound back to just outside the `lazy-catch` expression before invoking the handler. (For an introduction to what is meant by dynamic context, See Section 5.11.9 [Dynamic Wind], page 266.)

Then, when the handler *itself* throws an exception, that exception must be caught by some kind of `catch` (including perhaps another `lazy-catch`) higher up the call stack.

The dynamic context also includes `with-fluids` blocks (see Section 5.17.8 [Fluids and Dynamic States], page 330), so the effect of unwinding the dynamic context can also be seen in fluid variable values. This is illustrated by the following code, in which the normal case thunk uses `with-fluids` to temporarily change the value of a fluid:

```
(define f (make-fluid))
(fluid-set! f "top level value")

(define (handler . args)
  (cons (fluid-ref f) args))

(lazy-catch 'foo
             (lambda ()
               (with-fluids ((f "local value"))
                 (throw 'foo)))
             handler)
⇒
("top level value" foo)

((lambda ()
   (with-fluids ((f "local value"))
     (handler 'foo))))
⇒
("local value" foo)
```

In the `lazy-catch` version, the unwinding of dynamic context restores `f` to its value outside the `with-fluids` block before the handler is invoked, so the handler's `(fluid-ref f)` returns the external value.

`lazy-catch` is useful because it permits the implementation of debuggers and other reflective programming tools that need to access the state of the call stack at the exact point where an exception or an error is thrown. For an example of this, see REFFIXME:stack-catch.

It should be obvious from the above that `lazy-catch` is very similar to `with-throw-handler`. In fact Guile implements `lazy-catch` in exactly the same way as `with-throw-handler`, except with a flag set to say “where there are slight differences between what `with-throw-handler` and `lazy-catch` would do, do what `lazy-catch` has always done”. There are two such differences:

1. `with-throw-handler` handlers execute in the full dynamic context of the originating `throw` call. `lazy-catch` handlers execute in the dynamic context of the `lazy-catch` expression, excepting only that the stack has not yet been unwound from the point of the `throw` call.
2. If a `with-throw-handler` handler throws to a key that does not match the `with-throw-handler` expression's *key*, the new throw may be handled by a `catch` or throw handler that is _closer_ to the throw than the first `with-throw-handler`. If a `lazy-catch` handler throws, it will always be handled by a `catch` or throw handler that is higher up the dynamic context than the first `lazy-catch`.

Here is an example to illustrate the second difference:

```
(catch 'a
  (lambda ()
    (with-throw-handler 'b
      (lambda ()
        (catch 'a
          (lambda ()
            (throw 'b))
          inner-handler))
      (lambda (key . args)
        (throw 'a))))
  outer-handler)
```

This code will call `inner-handler` and then continue with the continuation of the inner `catch`. If the `with-throw-handler` was changed to `lazy-catch`, however, the code would call `outer-handler` and then continue with the continuation of the outer `catch`.

Modulo these two differences, any statements in the previous and following subsections about throw handlers apply to lazy catches as well.

5.11.7.5 Throwing Exceptions

The `throw` primitive is used to throw an exception. One argument, the *key*, is mandatory, and must be a symbol; it indicates the type of exception that is being thrown. Following the *key*, `throw` accepts any number of additional arguments, whose meaning depends on the exception type. The documentation for each possible type of exception should specify the additional arguments that are expected for that kind of exception.

`throw` *key . args* [Scheme Procedure]
`scm_throw` (*key, args*) [C Function]

Invoke the catch form matching *key*, passing *args* to the *handler*.

key is a symbol. It will match catches of the same symbol or of `#t`.

If there is no handler at all, Guile prints an error and then exits.

When an exception is thrown, it will be caught by the innermost `catch` or throw handler that applies to the type of the thrown exception; in other words, whose *key* is either `#t` or the same symbol as that used in the `throw` expression. Once Guile has identified the appropriate `catch` or throw handler, it handles the exception by applying the relevant handler procedure(s) to the arguments of the `throw`.

If there is no appropriate `catch` or throw handler for a thrown exception, Guile prints an error to the current error port indicating an uncaught exception, and then exits. In practice, it is quite difficult to observe this behaviour, because Guile when used interactively installs a top level `catch` handler that will catch all exceptions and print an appropriate error message *without* exiting. For example, this is what happens if you try to throw an unhandled exception in the standard Guile REPL; note that Guile's command loop continues after the error message:

```
guile> (throw 'badex)
<unnamed port>:3:1: In procedure gsubr-apply ...
<unnamed port>:3:1: unhandled-exception: badex
ABORT: (misc-error)
guile>
```

The default uncaught exception behaviour can be observed by evaluating a `throw` expression from the shell command line:

```
$ guile -c "(begin (throw 'badex) (display \"here\\n\"))"
guile: uncaught throw to badex: ()
$
```

That Guile exits immediately following the uncaught exception is shown by the absence of any output from the `display` expression, because Guile never gets to the point of evaluating that expression.

5.11.7.6 How Guile Implements Exceptions

It is traditional in Scheme to implement exception systems using `call-with-current-continuation`. Continuations (see Section 5.11.5 [Continuations], page 255) are such a powerful concept that any other control mechanism — including `catch` and `throw` — can be implemented in terms of them.

Guile does not implement `catch` and `throw` like this, though. Why not? Because Guile is specifically designed to be easy to integrate with applications written in C. In a mixed Scheme/C environment, the concept of *continuation* must logically include "what happens next" in the C parts of the application as well as the Scheme parts, and it turns out that the only reasonable way of implementing continuations like this is to save and restore the complete C stack.

So Guile's implementation of `call-with-current-continuation` is a stack copying one. This allows it to interact well with ordinary C code, but means that creating and calling a continuation is slowed down by the time that it takes to copy the C stack.

The more targeted mechanism provided by `catch` and `throw` does not need to save and restore the C stack because the `throw` always jumps to a location higher up the stack of the code that executes the `throw`. Therefore Guile implements the `catch` and `throw` primitives independently of `call-with-current-continuation`, in a way that takes advantage of this *upwards only* nature of exceptions.

5.11.8 Procedures for Signaling Errors

Guile provides a set of convenience procedures for signaling error conditions that are implemented on top of the exception primitives just described.

`error` *msg args . . .* [Scheme Procedure]

Raise an error with key `misc-error` and a message constructed by displaying *msg* and writing *args*.

`scm-error` *key subr message args data* [Scheme Procedure]
`scm_error_scm` (*key, subr, message, args, data*) [C Function]

Raise an error with key *key*. *subr* can be a string naming the procedure associated with the error, or `#f`. *message* is the error message string, possibly containing `~S` and `~A` escapes. When an error is reported, these are replaced by formatting the corresponding members of *args*: `~A` (was `%s` in older versions of Guile) formats using `display` and `~S` (was `%S`) formats using `write`. *data* is a list or `#f` depending on *key*: if *key* is `system-error` then it should be a list containing the Unix `errno` value; If *key* is `signal` then it should be a list containing the Unix signal number; If *key* is `out-of-range` or `wrong-type-arg`, it is a list containing the bad value; otherwise it will usually be `#f`.

`strerror` *err* [Scheme Procedure]
`scm_strerror` (*err*) [C Function]

Return the Unix error message corresponding to *err*, an integer `errno` value.

When `setlocale` has been called (see Section 6.2.13 [Locales], page 417), the message is in the language and charset of `LC_MESSAGES`. (This is done by the C library.)

`false-if-exception` *expr* [syntax]

Returns the result of evaluating its argument; however if an exception occurs then `#f` is returned instead.

5.11.9 Dynamic Wind

For Scheme code, the fundamental procedure to react to non-local entry and exits of dynamic contexts is `dynamic-wind`. C code could use `scm_internal_dynamic_wind`, but since C does not allow the convenient construction of anonymous procedures that close over lexical variables, this will be, well, inconvenient.

Therefore, Guile offers the functions `scm_dynwind_begin` and `scm_dynwind_end` to delimit a dynamic extent. Within this dynamic extent, which is calles a *dynwind context*, you can perform various *dynwind actions* that control what happens when the dynwind context is entered or left. For example, you can register a cleanup routine with `scm_dynwind_unwind_handler` that is executed when the context is left. There are several other more specialized dynwind actions as well, for example to temporarily block the execution of asyncs or to temporarily change the current output port. They are described elsewhere in this manual.

Here is an example that shows how to prevent memory leaks.

```
/* Suppose there is a function called FOO in some library that you
   would like to make available to Scheme code (or to C code that
   follows the Scheme conventions).

   FOO takes two C strings and returns a new string.  When an error has
   occurred in FOO, it returns NULL.
*/

char *foo (char *s1, char *s2);

/* SCM_FOO interfaces the C function FOO to the Scheme way of life.
   It takes care to free up all temporary strings in the case of
   non-local exits.
 */

SCM
scm_foo (SCM s1, SCM s2)
{
  char *c_s1, *c_s2, *c_res;

  scm_dynwind_begin (0);

  c_s1 = scm_to_locale_string (s1);

  /* Call 'free (c_s1)' when the dynwind context is left.
  */
  scm_dynwind_unwind_handler (free, c_s1, SCM_F_WIND_EXPLICITLY);
  c_s2 = scm_to_locale_string (s2);

  /* Same as above, but more concisely.
  */
  scm_dynwind_free (c_s2);

  c_res = foo (c_s1, c_s2);
  if (c_res == NULL)
    scm_memory_error ("foo");

  scm_dynwind_end ();

  return scm_take_locale_string (res);
}
```

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
`scm_dynamic_wind` (*in_guard, thunk, out_guard*) [C Function]

All three arguments must be 0-argument procedures. *in_guard* is called, then *thunk*, then *out_guard*.

If, any time during the execution of *thunk*, the dynamic extent of the `dynamic-wind` expression is escaped non-locally, *out_guard* is called. If the dynamic extent of the dynamic-wind is re-entered, *in_guard* is called. Thus *in_guard* and *out_guard* may be called any number of times.

```
(define x 'normal-binding)
⇒ x
(define a-cont
  (call-with-current-continuation
   (lambda (escape)
     (let ((old-x x))
       (dynamic-wind
           ;; in-guard:
           ;;
           (lambda () (set! x 'special-binding))

           ;; thunk
           ;;
           (lambda () (display x) (newline)
                      (call-with-current-continuation escape)
                      (display x) (newline)
                     x)

           ;; out-guard:
           ;;
           (lambda () (set! x old-x)))))))
;; Prints:
special-binding
```

```
        ;; Evaluates to:
        ⇒ a-cont
        x
        ⇒ normal-binding
        (a-cont #f)
        ;; Prints:
        special-binding
        ;; Evaluates to:
        ⇒ a-cont  ;; the value of the (define a-cont...)
        x
        ⇒ normal-binding
        a-cont
        ⇒ special-binding
```

`scm_t_dynwind_flags` [C Type]

This is an enumeration of several flags that modify the behavior of `scm_dynwind_begin`. The flags are listed in the following table.

`SCM_F_DYNWIND_REWINDABLE`

The dynamic context is *rewindable*. This means that it can be reentered non-locally (via the invokation of a continuation). The default is that a dynwind context can not be reentered non-locally.

`void scm_dynwind_begin` (*scm_t_dynwind_flags flags*) [C Function]

The function `scm_dynwind_begin` starts a new dynamic context and makes it the 'current' one.

The *flags* argument determines the default behavior of the context. Normally, use 0. This will result in a context that can not be reentered with a captured continuation. When you are prepared to handle reentries, include `SCM_F_DYNWIND_REWINDABLE` in *flags*.

Being prepared for reentry means that the effects of unwind handlers can be undone on reentry. In the example above, we want to prevent a memory leak on non-local exit and thus register an unwind handler that frees the memory. But once the memory is freed, we can not get it back on reentry. Thus reentry can not be allowed.

The consequence is that continuations become less useful when non-reenterable contexts are captured, but you don't need to worry about that too much.

The context is ended either implicitly when a non-local exit happens, or explicitly with `scm_dynwind_end`. You must make sure that a dynwind context is indeed ended properly. If you fail to call `scm_dynwind_end` for each `scm_dynwind_begin`, the behavior is undefined.

`void scm_dynwind_end ()` [C Function]

End the current dynamic context explicitly and make the previous one current.

`scm_t_wind_flags` [C Type]

This is an enumeration of several flags that modify the behavior of `scm_dynwind_unwind_handler` and `scm_dynwind_rewind_handler`. The flags are listed in the following table.

`SCM_F_WIND_EXPLICITLY`
: The registered action is also carried out when the dynwind context is entered or left locally.

`void scm_dynwind_unwind_handler` (*void (*func)(void *), void *data, scm_t_wind_flags flags*) [C Function]
`void scm_dynwind_unwind_handler_with_scm` (*void (*func)(SCM), SCM data, scm_t_wind_flags flags*) [C Function]

Arranges for *func* to be called with *data* as its arguments when the current context ends implicitly. If *flags* contains `SCM_F_WIND_EXPLICITLY`, *func* is also called when the context ends explicitly with `scm_dynwind_end`.

The function `scm_dynwind_unwind_handler_with_scm` takes care that *data* is protected from garbage collection.

`void scm_dynwind_rewind_handler` (*void (*func)(void *), void *data, scm_t_wind_flags flags*) [C Function]
`void scm_dynwind_rewind_handler_with_scm` (*void (*func)(SCM), SCM data, scm_t_wind_flags flags*) [C Function]

Arrange for *func* to be called with *data* as its argument when the current context is restarted by rewinding the stack. When *flags* contains `SCM_F_WIND_EXPLICITLY`, *func* is called immediately as well.

The function `scm_dynwind_rewind_handler_with_scm` takes care that *data* is protected from garbage collection.

`void scm_dynwind_free` (*void *mem*) [C Function]

Arrange for *mem* to be freed automatically whenever the current context is exited, whether normally or non-locally. `scm_dynwind_free (mem)` is an equivalent shorthand for `scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`.

5.11.10 How to Handle Errors

Error handling is based on `catch` and `throw`. Errors are always thrown with a *key* and four arguments:

- *key*: a symbol which indicates the type of error. The symbols used by libguile are listed below.
- *subr*: the name of the procedure from which the error is thrown, or `#f`.
- *message*: a string (possibly language and system dependent) describing the error. The tokens `~A` and `~S` can be embedded within the message: they will be replaced with members of the *args* list when the message is printed. `~A` indicates an argument printed using `display`, while `~S` indicates an argument printed using `write`. *message* can also be `#f`, to allow it to be derived from the *key* by the error handler (may be useful if the *key* is to be thrown from both C and Scheme).
- *args*: a list of arguments to be used to expand `~A` and `~S` tokens in *message*. Can also be `#f` if no arguments are required.
- *rest*: a list of any additional objects required. e.g., when the key is `'system-error`, this contains the C errno value. Can also be `#f` if no additional objects are required.

In addition to `catch` and `throw`, the following Scheme facilities are available:

`display-error` *stack port subr message args rest* [Scheme Procedure]
`scm_display_error` (*stack, port, subr, message, args, rest*) [C Function]

Display an error message to the output port *port*. *stack* is the saved stack for the error, *subr* is the name of the procedure in which the error occurred and *message* is the actual error message, which may contain formatting instructions. These will format the arguments in the list *args* accordingly. *rest* is currently ignored.

The following are the error keys defined by libguile and the situations in which they are used:

- `error-signal`: thrown after receiving an unhandled fatal signal such as SIGSEGV, SIGBUS, SIGFPE etc. The *rest* argument in the throw contains the coded signal number (at present this is not the same as the usual Unix signal number).
- `system-error`: thrown after the operating system indicates an error condition. The *rest* argument in the throw contains the errno value.
- `numerical-overflow`: numerical overflow.
- `out-of-range`: the arguments to a procedure do not fall within the accepted domain.
- `wrong-type-arg`: an argument to a procedure has the wrong type.
- `wrong-number-of-args`: a procedure was called with the wrong number of arguments.
- `memory-allocation-error`: memory allocation error.
- `stack-overflow`: stack overflow error.
- `regular-expression-syntax`: errors generated by the regular expression library.
- `misc-error`: other errors.

5.11.10.1 C Support

In the following C functions, *SUBR* and *MESSAGE* parameters can be `NULL` to give the effect of `#f` described above.

`SCM scm_error` (*SCM* `key`, *char* *`subr`, *char* *`message`, *SCM* `args`, *SCM* `rest`) [C Function]

Throw an error, as per `scm-error` above.

`void scm_syserror` (*char* *`subr`) [C Function]
`void scm_syserror_msg` (*char* *`subr`, *char* *`message`, *SCM* `args`) [C Function]

Throw an error with key `system-error` and supply `errno` in the *rest* argument. For `scm_syserror` the message is generated using `strerror`.

Care should be taken that any code in between the failing operation and the call to these routines doesn't change `errno`.

`void scm_num_overflow` (*char* *`subr`) [C Function]
`void scm_out_of_range` (*char* *`subr`, *SCM* `bad_value`) [C Function]
`void scm_wrong_num_args` (*SCM* `proc`) [C Function]
`void scm_wrong_type_arg` (*char* *`subr`, *int* `argnum`, *SCM* `bad_value`) [C Function]
`void scm_memory_error` (*char* *`subr`) [C Function]

Throw an error with the various keys described above.

For `scm_wrong_num_args`, *proc* should be a Scheme symbol which is the name of the procedure incorrectly invoked.

5.12 Input and Output

5.12.1 Ports

Sequential input/output in Scheme is represented by operations on a *port*. This chapter explains the operations that Guile provides for working with ports.

Ports are created by opening, for instance `open-file` for a file (see Section 5.12.9.1 [File Ports], page 280). Characters can be read from an input port and written to an output port, or both on an input/output port. A port can be closed (see Section 5.12.4 [Closing], page 275) when no longer required, after which any attempt to read or write is an error.

The formal definition of a port is very generic: an input port is simply "an object which can deliver characters on demand," and an output port is "an object which can accept characters." Because this definition is so loose, it is easy to write functions that simulate ports in software. *Soft ports* and *string ports* are two interesting and powerful examples of this technique. (see Section 5.12.9.3 [Soft Ports], page 283, and Section 5.12.9.2 [String Ports], page 282.)

Ports are garbage collected in the usual way (see Section 5.14 [Memory Management], page 296), and will be closed at that time if not already closed. In this case any errors occuring in the close will not be reported. Usually a program will want to explicitly close so as to be sure all its operations have been successful. Of course if a program has abandoned something due to an error or other condition then closing problems are probably not of interest.

It is strongly recommended that file ports be closed explicitly when no longer required. Most systems have limits on how many files can be open, both on a per-process and a system-wide basis. A program that uses many files should take care not to hit those limits. The same applies to similar system resources such as pipes and sockets.

Note that automatic garbage collection is triggered only by memory consumption, not by file or other resource usage, so a program cannot rely on that to keep it away from system limits. An explicit call to `gc` can of course be relied on to pick up unreferenced ports. If program flow makes it hard to be certain when to close then this may be an acceptable way to control resource usage.

All file access uses the "LFS" large file support functions when available, so files bigger than 2 Gbytes ($2^3$1 bytes) can be read and written on a 32-bit system.

`input-port?` *x* [Scheme Procedure]
`scm_input_port_p` (*x*) [C Function]

> Return `#t` if *x* is an input port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`output-port?` *x* [Scheme Procedure]
`scm_output_port_p` (*x*) [C Function]

> Return `#t` if *x* is an output port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`port?` *x* [Scheme Procedure]
`scm_port_p` (*x*) [C Function]

> Return a boolean indicating whether *x* is a port. Equivalent to `(or (input-port?` *x*`) (output-port?` *x*`))`.

5.12.2 Reading

[Generic procedures for reading from ports.]

`eof-object?` *x* [Scheme Procedure]
`scm_eof_object_p` (*x*) [C Function]

Return `#t` if *x* is an end-of-file object; otherwise return `#f`.

`char-ready?` [*port*] [Scheme Procedure]
`scm_char_ready_p` (*port*) [C Function]

Return `#t` if a character is ready on input *port* and return `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on *port* is guaranteed not to hang. If *port* is a file port at end of file then `char-ready?` returns `#t`.

`char-ready?` exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must make sure that characters whose existence has been asserted by `char-ready?` cannot be rubbed out. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

`read-char` [*port*] [Scheme Procedure]
`scm_read_char` (*port*) [C Function]

Return the next character available from *port*, updating *port* to point to the following character. If no more characters are available, the end-of-file object is returned.

`size_t scm_c_read` (*SCM port, void *buffer, size_t size*) [C Function]

Read up to *size* bytes from *port* and store them in *buffer*. The return value is the number of bytes actually read, which can be less than *size* if end-of-file has been reached.

Note that this function does not update `port-line` and `port-column` below.

`peek-char` [*port*] [Scheme Procedure]
`scm_peek_char` (*port*) [C Function]

Return the next character available from *port*, *without* updating *port* to point to the following character. If no more characters are available, the end-of-file object is returned.

The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` on the same port. The only difference is that the very next call to `read-char` or `peek-char` on that *port* will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

`unread-char` *cobj* [*port*] [Scheme Procedure]
`scm_unread_char` (*cobj, port*) [C Function]

Place *char* in *port* so that it will be read by the next read operation. If called multiple times, the unread characters will be read again in last-in first-out order. If *port* is not supplied, the current input port is used.

`unread-string` *str port* [Scheme Procedure]
`scm_unread_string` (*str, port*) [C Function]

Place the string *str* in *port* so that its characters will be read from left-to-right as the next characters from *port* during subsequent read operations. If called multiple times, the unread characters will be read again in last-in first-out order. If *port* is not supplied, the `current-input-port` is used.

`drain-input` *port* [Scheme Procedure]
`scm_drain_input` (*port*) [C Function]

This procedure clears a port's input buffers, similar to the way that force-output clears the output buffer. The contents of the buffers are returned as a single string, e.g.,

```
(define p (open-input-file ...))
(drain-input p) => empty string, nothing buffered yet.
(unread-char (read-char p) p)
(drain-input p) => initial chars from p, up to the buffer size.
```

Draining the buffers may be useful for cleanly finishing buffered I/O so that the file descriptor can be used directly for further input.

`port-column` *port* [Scheme Procedure]
`port-line` *port* [Scheme Procedure]
`scm_port_column` (*port*) [C Function]
`scm_port_line` (*port*) [C Function]

Return the current column number or line number of *port*. If the number is unknown, the result is #f. Otherwise, the result is a 0-origin integer - i.e. the first character of the first line is line 0, column 0. (However, when you display a file position, for example in an error message, we recommend you add 1 to get 1-origin integers. This is because lines and column numbers traditionally start with 1, and that is what non-programmers will find most natural.)

`set-port-column!` *port column* [Scheme Procedure]
`set-port-line!` *port line* [Scheme Procedure]
`scm_set_port_column_x` (*port, column*) [C Function]
`scm_set_port_line_x` (*port, line*) [C Function]

Set the current column or line number of *port*.

5.12.3 Writing

[Generic procedures for writing to ports.]

`get-print-state` *port* [Scheme Procedure]
`scm_get_print_state` (*port*) [C Function]

Return the print state of the port *port*. If *port* has no associated print state, `#f` is returned.

`write` *obj* [*port*] [Scheme Procedure]

Send a representation of *obj* to *port* or to the current output port if not given.

The output is designed to be machine readable, and can be read back with `read` (see Section 5.12.2 [Reading], page 272). Strings are printed in doublequotes, with escapes if necessary, and characters are printed in '`#\`' notation.

`display` *obj* [*port*] [Scheme Procedure]

Send a representation of *obj* to *port* or to the current output port if not given.

The output is designed for human readability, it differs from `write` in that strings are printed without doublequotes and escapes, and characters are printed as per `write-char`, not in '`#\`' form.

`newline` [*port*] [Scheme Procedure]
`scm_newline` (*port*) [C Function]

Send a newline to *port*. If *port* is omitted, send to the current output port.

`port-with-print-state` *port* [*pstate*] [Scheme Procedure]
`scm_port_with_print_state` (*port*, *pstate*) [C Function]

Create a new port which behaves like *port*, but with an included print state *pstate*. *pstate* is optional. If *pstate* isn't supplied and *port* already has a print state, the old print state is reused.

`print-options-interface` [*setting*] [Scheme Procedure]
`scm_print_options` (*setting*) [C Function]

Option interface for the print options. Instead of using this procedure directly, use the procedures `print-enable`, `print-disable`, `print-set!` and `print-options`.

`simple-format` *destination message . args* [Scheme Procedure]
`scm_simple_format` (*destination*, *message*, *args*) [C Function]

Write *message* to *destination*, defaulting to the current output port. *message* can contain `~A` (was `%s`) and `~S` (was `%S`) escapes. When printed, the escapes are replaced with corresponding members of *ARGS*: `~A` formats using `display` and `~S` formats using `write`. If *destination* is `#t`, then use the current output port, if *destination* is `#f`, then return a string containing the formatted text. Does not add a trailing newline.

`write-char` *chr* [*port*] [Scheme Procedure]
`scm_write_char` (*chr*, *port*) [C Function]

Send character *chr* to *port*.

`void scm_c_write` (*SCM port*, *const void *buffer*, *size_t size*) [C Function]

Write *size* bytes at *buffer* to *port*.

Note that this function does not update `port-line` and `port-column` (see Section 5.12.2 [Reading], page 272).

`force-output` [*port*] [Scheme Procedure]
`scm_force_output` (*port*) [C Function]

Flush the specified output port, or the current output port if *port* is omitted. The current output buffer contents are passed to the underlying port implementation (e.g., in the case of fports, the data will be written to the file and the output buffer will be cleared.) It has no effect on an unbuffered port.

The return value is unspecified.

`flush-all-ports` [Scheme Procedure]
`scm_flush_all_ports` () [C Function]

Equivalent to calling `force-output` on all open output ports. The return value is unspecified.

5.12.4 Closing

`close-port` *port* [Scheme Procedure]
`scm_close_port` (*port*) [C Function]

Close the specified port object. Return `#t` if it successfully closes a port or `#f` if it was already closed. An exception may be raised if an error occurs, for example when flushing buffered output. See also Section 6.2.2 [Ports and File Descriptors], page 374, for a procedure which can close file descriptors.

`close-input-port` *port* [Scheme Procedure]
`close-output-port` *port* [Scheme Procedure]
`scm_close_input_port` (*port*) [C Function]
`scm_close_output_port` (*port*) [C Function]

Close the specified input or output *port*. An exception may be raised if an error occurs while closing. If *port* is already closed, nothing is done. The return value is unspecified.

See also Section 6.2.2 [Ports and File Descriptors], page 374, for a procedure which can close file descriptors.

`port-closed?` *port* [Scheme Procedure]
`scm_port_closed_p` (*port*) [C Function]

Return `#t` if *port* is closed or `#f` if it is open.

5.12.5 Random Access

`seek` *fd_port offset whence* [Scheme Procedure]
`scm_seek` (*fd_port*, *offset*, *whence*) [C Function]

Sets the current position of *fd/port* to the integer *offset*, which is interpreted according to the value of *whence*.

One of the following variables should be supplied for *whence*:

`SEEK_SET` [Variable]

Seek from the beginning of the file.

`SEEK_CUR` [Variable]

Seek from the current position.

`SEEK_END` [Variable]

Seek from the end of the file.

If *fd/port* is a file descriptor, the underlying system call is `lseek`. *port* may be a string port.

The value returned is the new position in the file. This means that the current position of a port can be obtained using:

```
(seek port 0 SEEK_CUR)
```

`ftell` *fd_port* [Scheme Procedure]
`scm_ftell` (*fd_port*) [C Function]

Return an integer representing the current position of *fd/port*, measured from the beginning. Equivalent to:

```
(seek port 0 SEEK_CUR)
```

`truncate-file` *file* [*length*] [Scheme Procedure]
`scm_truncate_file` (*file, length*) [C Function]

Truncate *file* to *length* bytes. *file* can be a filename string, a port object, or an integer file descriptor. The return value is unspecified.

For a port or file descriptor *length* can be omitted, in which case the file is truncated at the current position (per `ftell` above).

On most systems a file can be extended by giving a length greater than the current size, but this is not mandatory in the POSIX standard.

5.12.6 Line Oriented and Delimited Text

The delimited-I/O module can be accessed with:

```
(use-modules (ice-9 rdelim))
```

It can be used to read or write lines of text, or read text delimited by a specified set of characters. It's similar to the `(scsh rdelim)` module from guile-scsh, but does not use multiple values or character sets and has an extra procedure `write-line`.

`read-line` [*port*] [*handle-delim*] [Scheme Procedure]

Return a line of text from *port* if specified, otherwise from the value returned by `(current-input-port)`. Under Unix, a line of text is terminated by the first end-of-line character or by end-of-file.

If *handle-delim* is specified, it should be one of the following symbols:

`trim` Discard the terminating delimiter. This is the default, but it will be impossible to tell whether the read terminated with a delimiter or end-of-file.

`concat` Append the terminating delimiter (if any) to the returned string.

`peek` Push the terminating delimiter (if any) back on to the port.

`split` Return a pair containing the string read from the port and the terminating delimiter or end-of-file object.

`read-line!` *buf* [*port*] [Scheme Procedure]

Read a line of text into the supplied string *buf* and return the number of characters added to *buf*. If *buf* is filled, then `#f` is returned. Read from *port* if specified, otherwise from the value returned by `(current-input-port)`.

`read-delimited` *delims* [*port*] [*handle-delim*] [Scheme Procedure]

Read text until one of the characters in the string *delims* is found or end-of-file is reached. Read from *port* if supplied, otherwise from the value returned by `(current-input-port)`. *handle-delim* takes the same values as described for `read-line`.

`read-delimited!` *delims buf* [*port*] [*handle-delim*] [*start*] [*end*] [Scheme Procedure]

Read text into the supplied string *buf* and return the number of characters added to *buf* (subject to *handle-delim*, which takes the same values specified for `read-line`. If *buf* is filled, `#f` is returned for both the number of characters read and the delimiter. Also terminates if one of the characters in the string *delims* is found or end-of-file is reached. Read from *port* if supplied, otherwise from the value returned by `(current-input-port)`.

`write-line` *obj* [*port*] [Scheme Procedure]
`scm_write_line` (*obj, port*) [C Function]

Display *obj* and a newline character to *port*. If *port* is not specified, `(current-output-port)` is used. This function is equivalent to:

```
(display obj [port])
(newline [port])
```

Some of the abovementioned I/O functions rely on the following C primitives. These will mainly be of interest to people hacking Guile internals.

`%read-delimited!` *delims str gobble* [*port* [*start* [*end*]]] [Scheme Procedure]
`scm_read_delimited_x` (*delims, str, gobble, port, start, end*) [C Function]

Read characters from *port* into *str* until one of the characters in the *delims* string is encountered. If *gobble* is true, discard the delimiter character; otherwise, leave it in the input stream for the next read. If *port* is not specified, use the value of `(current-input-port)`. If *start* or *end* are specified, store data only into the substring of *str* bounded by *start* and *end* (which default to the beginning and end of the string, respectively).

Return a pair consisting of the delimiter that terminated the string and the number of characters read. If reading stopped at the end of file, the delimiter returned is the *eof-object*; if the string was filled without encountering a delimiter, this value is `#f`.

`%read-line` [*port*] [Scheme Procedure]
`scm_read_line` (*port*) [C Function]

Read a newline-terminated line from *port*, allocating storage as necessary. The newline terminator (if any) is removed from the string, and a pair consisting of the line and its delimiter is returned. The delimiter may be either a newline or the *eof-object*; if `%read-line` is called at the end of file, it returns the pair `(#<eof> . #<eof>)`.

5.12.7 Block reading and writing

The Block-string-I/O module can be accessed with:

```
(use-modules (ice-9 rw))
```

It currently contains procedures that help to implement the `(scsh rw)` module in guile-scsh.

`read-string!/partial` *str* [*port_or_fdes* [*start* [*end*]]] [Scheme Procedure]
`scm_read_string_x_partial` (*str, port_or_fdes, start, end*) [C Function]

Read characters from a port or file descriptor into a string *str*. A port must have an underlying file descriptor — a so-called fport. This procedure is scsh-compatible and can efficiently read large strings. It will:

- attempt to fill the entire string, unless the *start* and/or *end* arguments are supplied. i.e., *start* defaults to 0 and *end* defaults to `(string-length str)`
- use the current input port if *port_or_fdes* is not supplied.
- return fewer than the requested number of characters in some cases, e.g., on end of file, if interrupted by a signal, or if not all the characters are immediately available.
- wait indefinitely for some input if no characters are currently available, unless the port is in non-blocking mode.
- read characters from the port's input buffers if available, instead from the underlying file descriptor.
- return `#f` if end-of-file is encountered before reading any characters, otherwise return the number of characters read.
- return 0 if the port is in non-blocking mode and no characters are immediately available.
- return 0 if the request is for 0 bytes, with no end-of-file check.

`write-string/partial` *str* [*port_or_fdes* [*start* [*end*]]] [Scheme Procedure]
`scm_write_string_partial` (*str*, *port_or_fdes*, *start*, *end*) [C Function]

Write characters from a string *str* to a port or file descriptor. A port must have an underlying file descriptor — a so-called fport. This procedure is scsh-compatible and can efficiently write large strings. It will:

- attempt to write the entire string, unless the *start* and/or *end* arguments are supplied. i.e., *start* defaults to 0 and *end* defaults to `(string-length str)`
- use the current output port if *port_of_fdes* is not supplied.
- in the case of a buffered port, store the characters in the port's output buffer, if all will fit. If they will not fit then any existing buffered characters will be flushed before attempting to write the new characters directly to the underlying file descriptor. If the port is in non-blocking mode and buffered characters can not be flushed immediately, then an `EAGAIN` system-error exception will be raised (Note: scsh does not support the use of non-blocking buffered ports.)
- write fewer than the requested number of characters in some cases, e.g., if interrupted by a signal or if not all of the output can be accepted immediately.
- wait indefinitely for at least one character from *str* to be accepted by the port, unless the port is in non-blocking mode.
- return the number of characters accepted by the port.
- return 0 if the port is in non-blocking mode and can not accept at least one character from *str* immediately
- return 0 immediately if the request size is 0 bytes.

5.12.8 Default Ports for Input, Output and Errors

`current-input-port` [Scheme Procedure]
`scm_current_input_port` () [C Function]

Return the current input port. This is the default port used by many input procedures.

Initially this is the *standard input* in Unix and C terminology. When the standard input is a tty the port is unbuffered, otherwise it's fully buffered.

Unbuffered input is good if an application runs an interactive subprocess, since any type-ahead input won't go into Guile's buffer and be unavailable to the subprocess.

Note that Guile buffering is completely separate from the tty "line discipline". In the usual cooked mode on a tty Guile only sees a line of input once the user presses ⟨Return⟩.

`current-output-port` [Scheme Procedure]
`scm_current_output_port` () [C Function]

Return the current output port. This is the default port used by many output procedures.

Initially this is the *standard output* in Unix and C terminology. When the standard output is a tty this port is unbuffered, otherwise it's fully buffered.

Unbuffered output to a tty is good for ensuring progress output or a prompt is seen. But an application which always prints whole lines could change to line buffered, or an application with a lot of output could go fully buffered and perhaps make explicit `force-output` calls (see Section 5.12.3 [Writing], page 273) at selected points.

`current-error-port` [Scheme Procedure]
`scm_current_error_port` () [C Function]

Return the port to which errors and warnings should be sent.

Initially this is the *standard error* in Unix and C terminology. When the standard error is a tty this port is unbuffered, otherwise it's fully buffered.

`set-current-input-port` *port* [Scheme Procedure]
`set-current-output-port` *port* [Scheme Procedure]
`set-current-error-port` *port* [Scheme Procedure]
`scm_set_current_input_port` (*port*) [C Function]
`scm_set_current_output_port` (*port*) [C Function]
`scm_set_current_error_port` (*port*) [C Function]

Change the ports returned by `current-input-port`, `current-output-port` and `current-error-port`, respectively, so that they use the supplied *port* for input or output.

`void scm_dynwind_current_input_port` (*SCM port*) [C Function]
`void scm_dynwind_current_output_port` (*SCM port*) [C Function]
`void scm_dynwind_current_error_port` (*SCM port*) [C Function]

These functions must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 5.11.9 [Dynamic Wind], page 266). During the dynwind context, the indicated port is set to *port*.

More precisely, the current port is swapped with a 'backup' value whenever the dynwind context is entered or left. The backup value is initialized with the *port* argument.

5.12.9 Types of Port

[Types of port; how to make them.]

5.12.9.1 File Ports

The following procedures are used to open file ports. See also Section 6.2.2 [Ports and File Descriptors], page 374, for an interface to the Unix `open` system call.

Most systems have limits on how many files can be open, so it's strongly recommended that file ports be closed explicitly when no longer required (see Section 5.12.1 [Ports], page 271).

`open-file` *filename mode* [Scheme Procedure]
`scm_open_file` (*filename*, *mode*) [C Function]

Open the file whose name is *filename*, and return a port representing that file. The attributes of the port are determined by the *mode* string. The way in which this is interpreted is similar to C stdio. The first character must be one of the following:

‘`r`’ Open an existing file for input.

‘`w`’ Open a file for output, creating it if it doesn't already exist or removing its contents if it does.

‘`a`’ Open a file for output, creating it if it doesn't already exist. All writes to the port will go to the end of the file. The "append mode" can be turned off while the port is in use see Section 6.2.2 [Ports and File Descriptors], page 374

The following additional characters can be appended:

‘`+`’ Open the port for both input and output. E.g., `r+`: open an existing file for both input and output.

‘`0`’ Create an "unbuffered" port. In this case input and output operations are passed directly to the underlying port implementation without additional buffering. This is likely to slow down I/O operations. The buffering mode can be changed while a port is in use see Section 6.2.2 [Ports and File Descriptors], page 374

‘`l`’ Add line-buffering to the port. The port output buffer will be automatically flushed whenever a newline character is written.

‘`b`’ Use binary mode. On DOS systems the default text mode converts CR+LF in the file to newline for the program, whereas binary mode reads and writes all bytes unchanged. On Unix-like systems there is no such distinction, text files already contain just newlines and no conversion is ever made. The `b` flag is accepted on all systems, but has no effect on Unix-like systems.

(For reference, Guile leaves text versus binary up to the C library, `b` here just adds `O_BINARY` to the underlying `open` call, when that flag is available.)

If a file cannot be opened with the access requested, `open-file` throws an exception.

In theory we could create read/write ports which were buffered in one direction only. However this isn't included in the current interfaces.

`open-input-file` *filename* [Scheme Procedure]

Open *filename* for input. Equivalent to

```
(open-file filename "r")
```

`open-output-file` *filename* [Scheme Procedure]

Open *filename* for output. Equivalent to

```
(open-file filename "w")
```

`call-with-input-file` *filename proc* [Scheme Procedure]
`call-with-output-file` *filename proc* [Scheme Procedure]

Open *filename* for input or output, and call `(proc port)` with the resulting port. Return the value returned by *proc*. *filename* is opened as per `open-input-file` or `open-output-file` respectively, and an error is signalled if it cannot be opened.

When *proc* returns, the port is closed. If *proc* does not return (eg. if it throws an error), then the port might not be closed automatically, though it will be garbage collected in the usual way if not otherwise referenced.

`with-input-from-file` *filename thunk* [Scheme Procedure]
`with-output-to-file` *filename thunk* [Scheme Procedure]
`with-error-to-file` *filename thunk* [Scheme Procedure]

Open *filename* and call `(thunk)` with the new port setup as respectively the `current-input-port`, `current-output-port`, or `current-error-port`. Return the value returned by *thunk*. *filename* is opened as per `open-input-file` or `open-output-file` respectively, and an error is signalled if it cannot be opened.

When *thunk* returns, the port is closed and the previous setting of the respective current port is restored.

The current port setting is managed with `dynamic-wind`, so the previous value is restored no matter how *thunk* exits (eg. an exception), and if *thunk* is re-entered (via a captured continuation) then it's set again to the *FILENAME* port.

The port is closed when *thunk* returns normally, but not when exited via an exception or new continuation. This ensures it's still ready for use if *thunk* is re-entered by a captured continuation. Of course the port is always garbage collected and closed in the usual way when no longer referenced anywhere.

`port-mode` *port* [Scheme Procedure]
`scm_port_mode` (*port*) [C Function]

Return the port modes associated with the open port *port*. These will not necessarily be identical to the modes used when the port was opened, since modes such as "append" which are used only during port creation are not retained.

`port-filename` *port* [Scheme Procedure]
`scm_port_filename` (*port*) [C Function]

Return the filename associated with *port*. This function returns the strings "standard input", "standard output" and "standard error" when called on the current input, output and error ports respectively.

port must be open, `port-filename` cannot be used once the port is closed.

`set-port-filename!` *port filename* [Scheme Procedure]
`scm_set_port_filename_x` (*port, filename*) [C Function]
: Change the filename associated with *port*, using the current input port if none is specified. Note that this does not change the port's source of data, but only the value that is returned by `port-filename` and reported in diagnostic output.

`file-port?` *obj* [Scheme Procedure]
`scm_file_port_p` (*obj*) [C Function]
: Determine whether *obj* is a port that is related to a file.

5.12.9.2 String Ports

The following allow string ports to be opened by analogy to R4R* file port facilities:

`call-with-output-string` *proc* [Scheme Procedure]
`scm_call_with_output_string` (*proc*) [C Function]
: Calls the one-argument procedure *proc* with a newly created output port. When the function returns, the string composed of the characters written into the port is returned. *proc* should not close the port.

`call-with-input-string` *string proc* [Scheme Procedure]
`scm_call_with_input_string` (*string, proc*) [C Function]
: Calls the one-argument procedure *proc* with a newly created input port from which *string*'s contents may be read. The value yielded by the *proc* is returned.

`with-output-to-string` *thunk* [Scheme Procedure]
: Calls the zero-argument procedure *thunk* with the current output port set temporarily to a new string port. It returns a string composed of the characters written to the current output.

`with-input-from-string` *string thunk* [Scheme Procedure]
: Calls the zero-argument procedure *thunk* with the current input port set temporarily to a string port opened on the specified *string*. The value yielded by *thunk* is returned.

`open-input-string` *str* [Scheme Procedure]
`scm_open_input_string` (*str*) [C Function]
: Take a string and return an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`open-output-string` [Scheme Procedure]
`scm_open_output_string` () [C Function]
: Return an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`get-output-string` *port* [Scheme Procedure]
`scm_get_output_string` (*port*) [C Function]
: Given an output port created by `open-output-string`, return a string consisting of the characters that have been output to the port so far.

: `get-output-string` must be used before closing *port*, once closed the string cannot be obtained.

A string port can be used in many procedures which accept a port but which are not dependent on implementation details of fports. E.g., seeking and truncating will work on a string port, but trying to extract the file descriptor number will fail.

5.12.9.3 Soft Ports

A *soft-port* is a port based on a vector of procedures capable of accepting or delivering characters. It allows emulation of I/O ports.

`make-soft-port` *pv modes* [Scheme Procedure]
`scm_make_soft_port` (*pv*, *modes*) [C Function]

Return a port capable of receiving or delivering characters as specified by the *modes* string (see Section 5.12.9.1 [File Ports], page 280). *pv* must be a vector of length 5 or 6. Its components are as follows:

0. procedure accepting one character for output
1. procedure accepting a string for output
2. thunk for flushing output
3. thunk for getting one character
4. thunk for closing port (not by garbage collection)
5. (if present and not `#f`) thunk for computing the number of characters that can be read from the port without blocking.

For an output-only port only elements 0, 1, 2, and 4 need be procedures. For an input-only port only elements 3 and 4 need be procedures. Thunks 2 and 4 can instead be `#f` if there is no useful operation for them to perform.

If thunk 3 returns `#f` or an `eof-object` (see section "Input" in *The Revised^5 Report on Scheme*) it indicates that the port has reached end-of-file. For example:

```
(define stdout (current-output-port))
(define p (make-soft-port
           (vector
            (lambda (c) (write c stdout))
            (lambda (s) (display s stdout))
            (lambda () (display "." stdout))
            (lambda () (char-upcase (read-char)))
            (lambda () (display "@" stdout)))
           "rw"))

(write p p) ⇒ #<input-output: soft 8081e20>
```

5.12.9.4 Void Ports

This kind of port causes any data to be discarded when written to, and always returns the end-of-file object when read from.

`%make-void-port` *mode* [Scheme Procedure]
`scm_sys_make_void_port` (*mode*) [C Function]

Create and return a new void port. A void port acts like '`/dev/null`'. The *mode* argument specifies the input/output modes for this port: see the documentation for `open-file` in Section 5.12.9.1 [File Ports], page 280.

5.12.10 Using and Extending Ports in C

5.12.10.1 C Port Interface

This section describes how to use Scheme ports from C.

Port basics

There are two main data structures. A port type object (ptob) is of type `scm_ptob_descriptor`. A port instance is of type `scm_port`. Given an `SCM` variable which points to a port, the corresponding C port object can be obtained using the `SCM_PTAB_ENTRY` macro. The ptob can be obtained by using `SCM_PTOBNUM` to give an index into the `scm_ptobs` global array.

Port buffers

An input port always has a read buffer and an output port always has a write buffer. However the size of these buffers is not guaranteed to be more than one byte (e.g., the `shortbuf` field in `scm_port` which is used when no other buffer is allocated). The way in which the buffers are allocated depends on the implementation of the ptob. For example in the case of an fport, buffers may be allocated with malloc when the port is created, but in the case of an strport the underlying string is used as the buffer.

The `rw_random` flag

Special treatment is required for ports which can be seeked at random. Before various operations, such as seeking the port or changing from input to output on a bidirectional port or vice versa, the port implementation must be given a chance to update its state. The write buffer is updated by calling the `flush` ptob procedure and the input buffer is updated by calling the `end_input` ptob procedure. In the case of an fport, `flush` causes buffered output to be written to the file descriptor, while `end_input` causes the descriptor position to be adjusted to account for buffered input which was never read.

The special treatment must be performed if the `rw_random` flag in the port is non-zero.

The `rw_active` variable

The `rw_active` variable in the port is only used if `rw_random` is set. It's defined as an enum with the following values:

`SCM_PORT_READ`
    the read buffer may have unread data.

`SCM_PORT_WRITE`
    the write buffer may have unwritten data.

`SCM_PORT_NEITHER`
    neither the write nor the read buffer has data.

Reading from a port.

To read from a port, it's possible to either call existing libguile procedures such as `scm_getc` and `scm_read_line` or to read data from the read buffer directly. Reading from the buffer involves the following steps:

1. Flush output on the port, if `rw_active` is `SCM_PORT_WRITE`.
2. Fill the read buffer, if it's empty, using `scm_fill_input`.
3. Read the data from the buffer and update the read position in the buffer. Steps 2) and 3) may be repeated as many times as required.
4. Set rw_active to `SCM_PORT_READ` if `rw_random` is set.
5. update the port's line and column counts.

Writing to a port.

To write data to a port, calling `scm_lfwrite` should be sufficient for most purposes. This takes care of the following steps:

1. End input on the port, if `rw_active` is `SCM_PORT_READ`.
2. Pass the data to the ptob implementation using the `write` ptob procedure. The advantage of using the ptob `write` instead of manipulating the write buffer directly is that it allows the data to be written in one operation even if the port is using the single-byte `shortbuf`.
3. Set `rw_active` to `SCM_PORT_WRITE` if `rw_random` is set.

5.12.10.2 Port Implementation

This section describes how to implement a new port type in C.

As described in the previous section, a port type object (ptob) is a structure of type `scm_ptob_descriptor`. A ptob is created by calling `scm_make_port_type`.

`scm_t_bits scm_make_port_type` (*char *name, int* (**fill_input*) (*SCM port*), *void* (**write*) (*SCM port, const void *data, size_t size*)) [Function]

Return a new port type object. The *name*, *fill_input* and *write* parameters are initial values for those port type fields, as described below. The other fields are initialized with default values and can be changed later.

All of the elements of the ptob, apart from `name`, are procedures which collectively implement the port behaviour. Creating a new port type mostly involves writing these procedures.

`name` A pointer to a NUL terminated string: the name of the port type. This is the only element of `scm_ptob_descriptor` which is not a procedure. Set via the first argument to `scm_make_port_type`.

`mark` Called during garbage collection to mark any SCM objects that a port object may contain. It doesn't need to be set unless the port has `SCM` components. Set using

`void scm_set_port_mark` (*scm_t_bits tc, SCM* (**mark*) (*SCM port*)) [Function]

`free` Called when the port is collected during gc. It should free any resources used by the port. Set using `void scm_set_port_free (`*scm_t_bits tc, size_t* `(`**free*`) (`*SCM port*`))` [Function]

`print` Called when `write` is called on the port object, to print a port description. E.g., for an fport it may produce something like: `#<input: /etc/passwd 3>`. Set using

`void scm_set_port_print (`*scm_t_bits tc, int* `(`**print*`) (`*SCM port, SCM dest_port, scm_print_state *pstate*`))` [Function]

The first argument *port* is the object being printed, the second argument *dest_port* is where its description should go.

`equalp` Not used at present. Set using

`void scm_set_port_equalp (`*scm_t_bits tc, SCM* `(`**equalp*`) (`*SCM, SCM*`))` [Function]

`close` Called when the port is closed, unless it was collected during gc. It should free any resources used by the port. Set using

`void scm_set_port_close (`*scm_t_bits tc, int* `(`**close*`) (`*SCM port*`))` [Function]

`write` Accept data which is to be written using the port. The port implementation may choose to buffer the data instead of processing it directly. Set via the third argument to `scm_make_port_type`.

`flush` Complete the processing of buffered output data. Reset the value of `rw_active` to `SCM_PORT_NEITHER`. Set using

`void scm_set_port_flush (`*scm_t_bits tc, void* `(`**flush*`) (`*SCM port*`))` [Function]

`end_input`
Perform any synchronization required when switching from input to output on the port. Reset the value of `rw_active` to `SCM_PORT_NEITHER`. Set using

`void scm_set_port_end_input (`*scm_t_bits tc, void* `(`**end_input*`) (`*SCM port, int offset*`))` [Function]

`fill_input`
Read new data into the read buffer and return the first character. It can be assumed that the read buffer is empty when this procedure is called. Set via the second argument to `scm_make_port_type`.

`input_waiting`
Return a lower bound on the number of bytes that could be read from the port without blocking. It can be assumed that the current state of `rw_active` is `SCM_PORT_NEITHER`. Set using

`void scm_set_port_input_waiting (`*scm_t_bits tc, int* `(`**input_waiting*`) (`*SCM port*`))` [Function]

`seek` Set the current position of the port. The procedure can not make any assumptions about the value of `rw_active` when it's called. It can reset the buffers first if desired by using something like:

```
if (pt->rw_active == SCM_PORT_READ)
  scm_end_input (port);
else if (pt->rw_active == SCM_PORT_WRITE)
  ptob->flush (port);
```

However note that this will have the side effect of discarding any data in the unread-char buffer, in addition to any side effects from the `end_input` and `flush` ptob procedures. This is undesirable when seek is called to measure the current position of the port, i.e., `(seek p 0 SEEK_CUR)`. The libguile fport and string port implementations take care to avoid this problem.

The procedure is set using

`void scm_set_port_seek` (*scm_t_bits tc, off_t* (**seek*) (*SCM port, off_t offset, int whence*)) [Function]

`truncate` Truncate the port data to be specified length. It can be assumed that the current state of `rw_active` is `SCM_PORT_NEITHER`. Set using

`void scm_set_port_truncate` (*scm_t_bits tc, void* (**truncate*) (*SCM port, off_t length*)) [Function]

5.13 Reading and Evaluating Scheme Code

This chapter describes Guile functions that are concerned with reading, loading and evaluating Scheme code at run time.

5.13.1 Scheme Syntax: Standard and Guile Extensions

5.13.1.1 Expression Syntax

An expression to be evaluated takes one of the following forms.

symbol
A symbol is evaluated by dereferencing. A binding of that symbol is sought and the value there used. For example,

```
(define x 123)
x ⇒ 123
```

(*proc args* ...)
A parenthesised expression is a function call. *proc* and each argument are evaluated, then the function (which *proc* evaluated to) is called with those arguments.

The order in which *proc* and the arguments are evaluated is unspecified, so be careful when using expressions with side effects.

```
(max 1 2 3) ⇒ 3

(define (get-some-proc)  min)
((get-some-proc) 1 2 3) ⇒ 1
```

The same sort of parenthesised form is used for a macro invocation, but in that case the arguments are not evaluated. See the descriptions of macros for more on this (see Section 5.8.6 [Macros], page 231, and see Section 5.8.7 [Syntax Rules], page 232).

constant
Number, string, character and boolean constants evaluate "to themselves", so can appear as literals.

```
123     ⇒ 123
99.9    ⇒ 99.9
"hello" ⇒ "hello"
#\z     ⇒ #\z
#t      ⇒ #t
```

Note that an application must not attempt to modify literal strings, since they may be in read-only memory.

(quote *data*)
'*data*
Quoting is used to obtain a literal symbol (instead of a variable reference), a literal list (instead of a function call), or a literal vector. ' is simply a shorthand for a quote form. For example,

```
'x                   ⇒ x
'(1 2 3)             ⇒ (1 2 3)
'#(1 (2 3) 4)        ⇒ #(1 (2 3) 4)
(quote x)            ⇒ x
```

```
(quote (1 2 3))       ⇒ (1 2 3)
(quote #(1 (2 3) 4)) ⇒ #(1 (2 3) 4)
```

Note that an application must not attempt to modify literal lists or vectors obtained from a `quote` form, since they may be in read-only memory.

`(quasiquote` *`data`*`)`
`‘`*`data`* Backquote quasi-quotation is like `quote`, but selected sub-expressions are evaluated. This is a convenient way to construct a list or vector structure most of which is constant, but at certain points should have expressions substituted.

The same effect can always be had with suitable `list`, `cons` or `vector` calls, but quasi-quoting is often easier.

`(unquote` *`expr`*`)`
`,`*`expr`* Within the quasiquote *data*, `unquote` or `,` indicates an expression to be evaluated and inserted. The comma syntax `,` is simply a shorthand for an `unquote` form. For example,

```
‘(1 2 ,(* 9 9) 3 4)       ⇒ (1 2 81 3 4)
‘(1 (unquote (+ 1 1)) 3) ⇒ (1 2 3)
‘#(1 ,(/ 12 2))           ⇒ #(1 6)
```

`(unquote-splicing` *`expr`*`)`
`,@`*`expr`* Within the quasiquote *data*, `unquote-splicing` or `,@` indicates an expression to be evaluated and the elements of the returned list inserted. *expr* must evaluate to a list. The “comma-at” syntax `,@` is simply a shorthand for an `unquote-splicing` form.

```
(define x '(2 3))
‘(1 ,@x 4)                          ⇒ (1 2 3 4)
‘(1 (unquote-splicing (map 1+ x))) ⇒ (1 3 4)
‘#(9 ,@x 9)                         ⇒ #(9 2 3 9)
```

Notice `,@` differs from plain `,` in the way one level of nesting is stripped. For `,@` the elements of a returned list are inserted, whereas with `,` it would be the list itself inserted.

5.13.1.2 Comments

Comments in Scheme source files are written by starting them with a semicolon character (`;`). The comment then reaches up to the end of the line. Comments can begin at any column, and the may be inserted on the same line as Scheme code.

```
; Comment
;; Comment too
(define x 1)        ; Comment after expression
(let ((y 1))
  ;; Display something.
  (display y)
;;; Comment at left margin.
  (display (+ y 1)))
```

It is common to use a single semicolon for comments following expressions on a line, to use two semicolons for comments which are indented like code, and three semicolons for comments which start at column 0, even if they are inside an indented code block. This convention is used when indenting code in Emacs' Scheme mode.

5.13.1.3 Block Comments

In addition to the standard line comments defined by R5RS, Guile has another comment type for multiline comments, called *block comments*. This type of comment begins with the character sequence `#!` and ends with the characters `!#`, which must appear on a line of their own. These comments are compatible with the block comments in the Scheme Shell '`scsh`' (see Section 6.14 [The Scheme shell (scsh)], page 483). The characters `#!` were chosen because they are the magic characters used in shell scripts for indicating that the name of the program for executing the script follows on the same line.

Thus a Guile script often starts like this.

```
#! /usr/local/bin/guile -s
!#
```

More details on Guile scripting can be found in the scripting section (see Section 3.3 [Guile Scripting], page 33).

5.13.1.4 Case Sensitivity

Scheme as defined in R5RS is not case sensitive when reading symbols. Guile, on the contrary is case sensitive by default, so the identifiers

```
guile-whuzzy
Guile-Whuzzy
```

are the same in R5RS Scheme, but are different in Guile.

It is possible to turn off case sensitivity in Guile by setting the reader option `case-insensitive`. More on reader options can be found at (see Section 5.18.3.3 [Reader options], page 339).

```
(read-enable 'case-insensitive)
```

Note that this is seldom a problem, because Scheme programmers tend not to use uppercase letters in their identifiers anyway.

5.13.1.5 Keyword Syntax

5.13.1.6 Reader Extensions

`read-hash-extend` *chr proc* [Scheme Procedure]
`scm_read_hash_extend` (*chr*, *proc*) [C Function]

Install the procedure *proc* for reading expressions starting with the character sequence `#` and *chr*. *proc* will be called with two arguments: the character *chr* and the port to read further data from. The object returned will be the return value of `read`.

5.13.2 Reading Scheme Code

`read` [*port*] [Scheme Procedure]
`scm_read` (*port*) [C Function]

Read an s-expression from the input port *port*, or from the current input port if *port* is not specified. Any whitespace before the next token is discarded.

The behaviour of Guile's Scheme reader can be modified by manipulating its read options. For more information about options, See Section 5.18.3.2 [User level options interfaces], page 339. If you want to know which reader options are available, See Section 5.18.3.3 [Reader options], page 339.

`read-options` [*setting*] [Scheme Procedure]

Display the current settings of the read options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise, *setting* should be one of the following symbols:

`help`	Display the complete option settings.
`full`	Like `help`, but also print programmer options.

`read-enable` *option-name* [Scheme Procedure]
`read-disable` *option-name* [Scheme Procedure]
`read-set!` *option-name value* [Scheme Procedure]

Modify the read options. `read-enable` should be used with boolean options and switches them on, `read-disable` switches them off. `read-set!` can be used to set an option to a specific value.

`read-options-interface` [*setting*] [Scheme Procedure]
`scm_read_options` (*setting*) [C Function]

Option interface for the read options. Instead of using this procedure directly, use the procedures `read-enable`, `read-disable`, `read-set!` and `read-options`.

5.13.3 Procedures for On the Fly Evaluation

See Section 5.16.2 [Environments], page 303.

`eval` *exp module_or_state* [Scheme Procedure]
`scm_eval` (*exp, module_or_state*) [C Function]

Evaluate *exp*, a list representing a Scheme expression, in the top-level environment specified by *module*. While *exp* is evaluated (using `primitive-eval`), *module* is made the current module. The current module is reset to its previous value when *eval* returns. XXX - dynamic states. Example: (eval '(+ 1 2) (interaction-environment))

`interaction-environment` [Scheme Procedure]
`scm_interaction_environment` () [C Function]

Return a specifier for the environment that contains implementation–defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

`eval-string` *string* [*module*] [Scheme Procedure]
`scm_eval_string` (*string*) [C Function]
`scm_eval_string_in_module` (*string, module*) [C Function]

Evaluate *string* as the text representation of a Scheme form or forms, and return whatever value they produce. Evaluation takes place in the given module, or in the current module when no module is given. While the code is evaluated, the given module is made the current one. The current module is restored when this procedure returns.

`SCM scm_c_eval_string` (*const char *string*) [C Function]

`scm_eval_string`, but taking a C string instead of an `SCM`.

`apply` *proc arg1 . . . argN arglst* [Scheme Procedure]
`scm_apply_0` (*proc, arglst*) [C Function]
`scm_apply_1` (*proc, arg1, arglst*) [C Function]
`scm_apply_2` (*proc, arg1, arg2, arglst*) [C Function]
`scm_apply_3` (*proc, arg1, arg2, arg3, arglst*) [C Function]
`scm_apply` (*proc, arg, rest*) [C Function]

Call *proc* with arguments *arg1 . . . argN* plus the elements of the *arglst* list.

`scm_apply` takes parameters corresponding to a Scheme level `(lambda (proc arg . rest) ...)`. So *arg* and all but the last element of the *rest* list make up *arg1 . . . argN* and the last element of *rest* is the *arglst* list. Or if *rest* is the empty list `SCM_EOL` then there's no *arg1 . . . argN* and *arg* is the *arglst*.

arglst is not modified, but the *rest* list passed to `scm_apply` is modified.

`scm_call_0` (*proc*) [C Function]
`scm_call_1` (*proc, arg1*) [C Function]
`scm_call_2` (*proc, arg1, arg2*) [C Function]
`scm_call_3` (*proc, arg1, arg2, arg3*) [C Function]
`scm_call_4` (*proc, arg1, arg2, arg3, arg4*) [C Function]

Call *proc* with the given arguments.

`apply:nconc2last` *lst* [Scheme Procedure]
`scm_nconc2last` (*lst*) [C Function]

lst should be a list (*arg1 . . . argN arglst*), with *arglst* being a list. This function returns a list comprising *arg1* to *argN* plus the elements of *arglst*. *lst* is modified to form the return. *arglst* is not modified, though the return does share structure with it.

This operation collects up the arguments from a list which is `apply` style parameters.

`primitive-eval` *exp* [Scheme Procedure]
`scm_primitive_eval` (*exp*) [C Function]

Evaluate *exp* in the top-level environment specified by the current module.

5.13.4 Loading Scheme Code from File

`load` *filename* [*reader*] [Scheme Procedure]

Load *filename* and evaluate its contents in the top-level environment. The load paths are not searched.

reader if provided should be either `#f`, or a procedure with the signature `(lambda (port) ...)` which reads the next expression from *port*. If *reader* is `#f` or absent, Guile's built-in `read` procedure is used (see Section 5.13.2 [Scheme Read], page 290).

The *reader* argument takes effect by setting the value of the `current-reader` fluid (see below) before loading the file, and restoring its previous value when loading is complete. The Scheme code inside *filename* can itself change the current reader procedure on the fly by setting `current-reader` fluid.

If the variable `%load-hook` is defined, it should be bound to a procedure that will be called before any code is loaded. See documentation for `%load-hook` later in this section.

`load-from-path` *filename* [Scheme Procedure]

Similar to `load`, but searches for *filename* in the load paths.

`primitive-load` *filename* [Scheme Procedure]
`scm_primitive_load` (*filename*) [C Function]

Load the file named *filename* and evaluate its contents in the top-level environment. The load paths are not searched; *filename* must either be a full pathname or be a pathname relative to the current directory. If the variable `%load-hook` is defined, it should be bound to a procedure that will be called before any code is loaded. See the documentation for `%load-hook` later in this section.

`SCM scm_c_primitive_load` (*const char *filename*) [C Function]

`scm_primitive_load`, but taking a C string instead of an `SCM`.

`primitive-load-path` *filename* [Scheme Procedure]
`scm_primitive_load_path` (*filename*) [C Function]

Search `%load-path` for the file named *filename* and load it into the top-level environment. If *filename* is a relative pathname and is not found in the list of search paths, an error is signalled.

`%search-load-path` *filename* [Scheme Procedure]
`scm_sys_search_load_path` (*filename*) [C Function]

Search `%load-path` for the file named *filename*, which must be readable by the current user. If *filename* is found in the list of paths to search or is an absolute pathname, return its full pathname. Otherwise, return `#f`. Filenames may have any of the optional extensions in the `%load-extensions` list; `%search-load-path` will try each extension automatically.

`current-reader` [Variable]

`current-reader` holds the read procedure that is currently being used by the above loading procedures to read expressions (from the file that they are loading). `current-reader` is a fluid, so it has an independent value in each dynamic root and should be read and set using `fluid-ref` and `fluid-set!` (see Section 5.17.8 [Fluids and Dynamic States], page 330).

`%load-hook` [Variable]

A procedure to be called `(%load-hook` *`filename`*`)` whenever a file is loaded, or `#f` for no such call. `%load-hook` is used by all of the above loading functions (`load`, `load-path`, `primitive-load` and `primitive-load-path`).

For example an application can set this to show what's loaded,

```
(set! %load-hook (lambda (filename)
                   (format #t "Loading ~a ...\n" filename)))
(load-from-path "foo.scm")
⊣ Loading /usr/local/share/guile/site/foo.scm ...
```

`current-load-port` [Scheme Procedure]
`scm_current_load_port ()` [C Function]

Return the current-load-port. The load port is used internally by `primitive-load`.

`%load-extensions` [Variable]

A list of default file extensions for files containing Scheme code. `%search-load-path` tries each of these extensions when looking for a file to load. By default, `%load-extensions` is bound to the list `("" ".scm")`.

5.13.5 Delayed Evaluation

Promises are a convenient way to defer a calculation until its result is actually needed, and to run such a calculation only once.

`delay` *expr* [syntax]

Return a promise object which holds the given *expr* expression, ready to be evaluated by a later `force`.

`promise?` *obj* [Scheme Procedure]
`scm_promise_p` (*obj*) [C Function]

Return true if *obj* is a promise.

`force` *p* [Scheme Procedure]
`scm_force` (*p*) [C Function]

Return the value obtained from evaluating the *expr* in the given promise *p*. If *p* has previously been forced then its *expr* is not evaluated again, instead the value obtained at that time is simply returned.

During a `force`, an *expr* can call `force` again on its own promise, resulting in a recursive evaluation of that *expr*. The first evaluation to return gives the value for the promise. Higher evaluations run to completion in the normal way, but their results are ignored, `force` always returns the first value.

5.13.6 Local Evaluation

[the-environment]

`local-eval` *exp* [*env*] [Scheme Procedure]
`scm_local_eval` (*exp*, *env*) [C Function]

Evaluate *exp* in its environment. If *env* is supplied, it is the environment in which to evaluate *exp*. Otherwise, *exp* must be a memoized code object (in which case, its environment is implicit).

5.13.7 Evaluator Behaviour

The behaviour of Guile's evaluator can be modified by manipulating the evaluator options. For more information about options, See Section 5.18.3.2 [User level options interfaces], page 339. If you want to know which evaluator options are available, See Section 5.18.3.5 [Evaluator options], page 340.

`eval-options` [*setting*] [Scheme Procedure]

Display the current settings of the evaluator options. If *setting* is omitted, only a short form of the current evaluator options is printed. Otherwise, *setting* should be one of the following symbols:

`help`	Display the complete option settings.
`full`	Like `help`, but also print programmer options.

`eval-enable` *option-name* [Scheme Procedure]
`eval-disable` *option-name* [Scheme Procedure]
`eval-set!` *option-name value* [Scheme Procedure]

Modify the evaluator options. `eval-enable` should be used with boolean options and switches them on, `eval-disable` switches them off. `eval-set!` can be used to set an option to a specific value.

`eval-options-interface` [*setting*] [Scheme Procedure]
`scm_eval_options_interface` (*setting*) [C Function]

Option interface for the evaluation options. Instead of using this procedure directly, use the procedures `eval-enable`, `eval-disable`, `eval-set!` and `eval-options`.

`traps` [*setting*] [Scheme Procedure]

Display the current settings of the evaluator traps options. If *setting* is omitted, only a short form of the current evaluator traps options is printed. Otherwise, *setting* should be one of the following symbols:

`help`	Display the complete option settings.
`full`	Like `help`, but also print programmer options.

`trap-enable` *option-name* [Scheme Procedure]
`trap-disable` *option-name* [Scheme Procedure]
`trap-set!` *option-name value* [Scheme Procedure]

Modify the evaluator options. `trap-enable` should be used with boolean options and switches them on, `trap-disable` switches them off. `trap-set!` can be used to set an option to a specific value.

`evaluator-traps-interface` [*setting*] [Scheme Procedure]
`scm_evaluator_traps` (*setting*) [C Function]

Option interface for the evaluator trap options.

5.14 Memory Management and Garbage Collection

Guile uses a *garbage collector* to manage most of its objects. While the garbage collector is designed to be mostly invisible, you sometimes need to interact with it explicitely.

See Section 4.3.2 [Garbage Collection], page 62 for a general discussion of how garbage collection relates to using Guile from C.

5.14.1 Function related to Garbage Collection

`gc` [Scheme Procedure]
`scm_gc ()` [C Function]

Scans all of SCM objects and reclaims for further use those that are no longer accessible. You normally don't need to call this function explicitly. It is called automatically when appropriate.

`SCM scm_gc_protect_object (`*SCM* `obj)` [C Function]

Protects *obj* from being freed by the garbage collector, when it otherwise might be. When you are done with the object, call `scm_gc_unprotect_object` on the object. Calls to `scm_gc_protect/scm_gc_unprotect_object` can be nested, and the object remains protected until it has been unprotected as many times as it was protected. It is an error to unprotect an object more times than it has been protected. Returns the SCM object it was passed.

`SCM scm_gc_unprotect_object (`*SCM* `obj)` [C Function]

Unprotects an object from the garbage collector which was protected by `scm_gc_unprotect_object`. Returns the SCM object it was passed.

`SCM scm_permanent_object (`*SCM* `obj)` [C Function]

Similar to `scm_gc_protect_object` in that it causes the collector to always mark the object, except that it should not be nested (only call `scm_permanent_object` on an object once), and it has no corresponding unpermanent function. Once an object is declared permanent, it will never be freed. Returns the SCM object it was passed.

`void scm_remember_upto_here_1 (`*SCM obj*`)` [C Macro]
`void scm_remember_upto_here_2 (`*SCM obj1, SCM obj2*`)` [C Macro]

Create a reference to the given object or objects, so they're certain to be present on the stack or in a register and hence will not be freed by the garbage collector before this point.

Note that these functions can only be applied to ordinary C local variables (ie. "automatics"). Objects held in global or static variables or some malloced block or the like cannot be protected with this mechanism.

`gc-stats` [Scheme Procedure]
`scm_gc_stats ()` [C Function]

Return an association list of statistics about Guile's current use of storage.

`gc-live-object-stats` [Scheme Procedure]
`scm_gc_live_object_stats ()` [C Function]

Return an alist of statistics of the current live objects.

`void scm_gc_mark (`*SCM x*`)` [Function]

Mark the object *x*, and recurse on any objects *x* refers to. If *x*'s mark bit is already set, return immediately. This function must only be called during the mark-phase of garbage collection, typically from a smob *mark* function.

5.14.2 Memory Blocks

In C programs, dynamic management of memory blocks is normally done with the functions malloc, realloc, and free. Guile has additional functions for dynamic memory allocation that are integrated into the garbage collector and the error reporting system.

Memory blocks that are associated with Scheme objects (for example a smob) should be allocated and freed with `scm_gc_malloc` and `scm_gc_free`. The function `scm_gc_malloc` will either return a valid pointer or signal an error. It will also assume that the new memory can be freed by a garbage collection. The garbage collector uses this information to decide when to try to actually collect some garbage. Memory blocks allocated with `scm_gc_malloc` must be freed with `scm_gc_free`.

For memory that is not associated with a Scheme object, you can use `scm_malloc` instead of `malloc`. Like `scm_gc_malloc`, it will either return a valid pointer or signal an error. However, it will not assume that the new memory block can be freed by a garbage collection. The memory can be freed with `free`.

There is also `scm_gc_realloc` and `scm_realloc`, to be used in place of `realloc` when appropriate, and `scm_gc_calloc` and `scm_calloc`, to be used in place of `calloc` when appropriate.

The function `scm_dynwind_free` can be useful when memory should be freed when a dynwind context, See Section 5.11.9 [Dynamic Wind], page 266.

For really specialized needs, take at look at `scm_gc_register_collectable_memory` and `scm_gc_unregister_collectable_memory`.

`void * scm_malloc (`*size_t* `size)` [C Function]
`void * scm_calloc (`*size_t* `size)` [C Function]

Allocate *size* bytes of memory and return a pointer to it. When *size* is 0, return `NULL`. When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

The memory is allocated by the libc `malloc` function and can be freed with `free`. There is no `scm_free` function to go with `scm_malloc` to make it easier to pass memory back and forth between different modules.

The function `scm_calloc` is similar to `scm_malloc`, but initializes the block of memory to zero as well.

`void * scm_realloc (`*void *mem*, *size_t* `new_size)` [C Function]

Change the size of the memory block at *mem* to *new_size* and return its new location. When *new_size* is 0, this is the same as calling `free` on *mem* and `NULL` is returned. When *mem* is `NULL`, this function behaves like `scm_malloc` and allocates a new block of size *new_size*.

When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

`void scm_gc_register_collectable_memory` (*void *mem, size_t size, const char *what*) [C Function]

Informs the GC that the memory at *mem* of size *size* can potentially be freed during a GC. That is, announce that *mem* is part of a GC controlled object and when the GC happens to free that object, *size* bytes will be freed along with it. The GC will **not** free the memory itself, it will just know that so-and-so much bytes of memory are associated with GC controlled objects and the memory system figures this into its decisions when to run a GC.

mem does not need to come from `scm_malloc`. You can only call this function once for every memory block.

The *what* argument is used for statistical purposes. It should describe the type of object that the memory will be used for so that users can identify just what strange objects are eating up their memory.

`void scm_gc_unregister_collectable_memory` (*void *mem, size_t size*) [C Function]

Informs the GC that the memory at *mem* of size *size* is no longer associated with a GC controlled object. You must take care to match up every call to `scm_gc_register_collectable_memory` with a call to `scm_gc_unregister_collectable_memory`. If you don't do this, the GC might have a wrong impression of what is going on and run much less efficiently than it could.

`void * scm_gc_malloc` (*size_t size, const char *what*) [C Function]
`void * scm_gc_realloc` (*void *mem, size_t old_size, size_t new_size, const char *what);* [C Function]
`void * scm_gc_calloc` (*size_t size, const char *what*) [C Function]

Like `scm_malloc`, `scm_realloc` or `scm_calloc`, but also call `scm_gc_register_collectable_memory`. Note that you need to pass the old size of a reallocated memory block as well. See below for a motivation.

`void scm_gc_free` (*void *mem, size_t size, const char *what*) [C Function]

Like `free`, but also call `scm_gc_unregister_collectable_memory`.

Note that you need to explicitely pass the *size* parameter. This is done since it should normally be easy to provide this parameter (for memory that is associated with GC controlled objects) and this frees us from tracking this value in the GC itself, which will keep the memory management overhead very low.

`void scm_frame_free` (*void *mem*) [C Function]

Equivalent to `scm_frame_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`. That is, the memory block at *mem* will be freed when the current frame is left.

`malloc-stats` [Scheme Procedure]

Return an alist ((*what* . *n*) ...) describing number of malloced objects. *what* is the second argument to `scm_gc_malloc`, *n* is the number of objects of that type currently allocated.

5.14.2.1 Upgrading from scm_must_malloc et al.

Version 1.6 of Guile and earlier did not have the functions from the previous section. In their place, it had the functions `scm_must_malloc`, `scm_must_realloc` and `scm_must_free`. This section explains why we want you to stop using them, and how to do this.

The functions `scm_must_malloc` and `scm_must_realloc` behaved like `scm_gc_malloc` and `scm_gc_realloc` do now, respectively. They would inform the GC about the newly allocated memory via the internal equivalent of `scm_gc_register_collectable_memory`. However, `scm_must_free` did not unregister the memory it was about to free. The usual way to unregister memory was to return its size from a smob free function.

This disconnectedness of the actual freeing of memory and reporting this to the GC proved to be bad in practice. It was easy to make mistakes and report the wrong size because allocating and freeing was not done with symmetric code, and because it is cumbersome to compute the total size of nested data structures that were freed with multiple calls to `scm_must_free`. Additionally, there was no equivalent to `scm_malloc`, and it was tempting to just use `scm_must_malloc` and never to tell the GC that the memory has been freed.

The effect was that the internal statistics kept by the GC drifted out of sync with reality and could even overflow in long running programs. When this happened, the result was a dramatic increase in (senseless) GC activity which would effectively stop the program dead.

The functions `scm_done_malloc` and `scm_done_free` were introduced to help restore balance to the force, but existing bugs did not magically disappear, of course.

Therefore we decided to force everybody to review their code by deprecating the existing functions and introducing new ones in their place that are hopefully easier to use correctly.

For every use of `scm_must_malloc` you need to decide whether to use `scm_malloc` or `scm_gc_malloc` in its place. When the memory block is not part of a smob or some other Scheme object whose lifetime is ultimately managed by the garbage collector, use `scm_malloc` and `free`. When it is part of a smob, use `scm_gc_malloc` and change the smob free function to use `scm_gc_free` instead of `scm_must_free` or `free` and make it return zero.

The important thing is to always pair `scm_malloc` with `free`; and to always pair `scm_gc_malloc` with `scm_gc_free`.

The same reasoning applies to `scm_must_realloc` and `scm_realloc` versus `scm_gc_realloc`.

5.14.3 Weak References

[FIXME: This chapter is based on Mikael Djurfeldt's answer to a question by Michael Livshin. Any mistakes are not theirs, of course.]

Weak references let you attach bookkeeping information to data so that the additional information automatically disappears when the original data is no longer in use and gets garbage collected. In a weak key hash, the hash entry for that key disappears as soon as the key is no longer referenced from anywhere else. For weak value hashes, the same happens as soon as the value is no longer in use. Entries in a doubly weak hash disappear when either the key or the value are not used anywhere else anymore.

Object properties offer the same kind of functionality as weak key hashes in many situations. (see Section 5.9.2 [Object Properties], page 238)

Here's an example (a little bit strained perhaps, but one of the examples is actually used in Guile):

Assume that you're implementing a debugging system where you want to associate information about filename and position of source code expressions with the expressions themselves.

Hashtables can be used for that, but if you use ordinary hash tables it will be impossible for the scheme interpreter to "forget" old source when, for example, a file is reloaded.

To implement the mapping from source code expressions to positional information it is necessary to use weak-key tables since we don't want the expressions to be remembered just because they are in our table.

To implement a mapping from source file line numbers to source code expressions you would use a weak-value table.

To implement a mapping from source code expressions to the procedures they constitute a doubly-weak table has to be used.

5.14.3.1 Weak hash tables

`make-weak-key-hash-table` *size* [Scheme Procedure]
`make-weak-value-hash-table` *size* [Scheme Procedure]
`make-doubly-weak-hash-table` *size* [Scheme Procedure]
`scm_make_weak_key_hash_table` (*size*) [C Function]
`scm_make_weak_value_hash_table` (*size*) [C Function]
`scm_make_doubly_weak_hash_table` (*size*) [C Function]

> Return a weak hash table with *size* buckets. As with any hash table, choosing a good size for the table requires some caution.
>
> You can modify weak hash tables in exactly the same way you would modify regular hash tables. (see Section 5.6.12 [Hash Tables], page 215)

`weak-key-hash-table?` *obj* [Scheme Procedure]
`weak-value-hash-table?` *obj* [Scheme Procedure]
`doubly-weak-hash-table?` *obj* [Scheme Procedure]
`scm_weak_key_hash_table_p` (*obj*) [C Function]
`scm_weak_value_hash_table_p` (*obj*) [C Function]
`scm_doubly_weak_hash_table_p` (*obj*) [C Function]

> Return `#t` if *obj* is the specified weak hash table. Note that a doubly weak hash table is neither a weak key nor a weak value hash table.

5.14.3.2 Weak vectors

Weak vectors are mainly useful in Guile's implementation of weak hash tables.

`make-weak-vector` *size* [*fill*] [Scheme Procedure]
`scm_make_weak_vector` (*size, fill*) [C Function]

> Return a weak vector with *size* elements. If the optional argument *fill* is given, all entries in the vector will be set to *fill*. The default value for *fill* is the empty list.

`weak-vector` *. l* [Scheme Procedure]
`list->weak-vector` *l* [Scheme Procedure]

`scm_weak_vector` (*l*) [C Function]

Construct a weak vector from a list: `weak-vector` uses the list of its arguments while `list->weak-vector` uses its only argument *l* (a list) to construct a weak vector the same way `list->vector` would.

`weak-vector?` *obj* [Scheme Procedure]
`scm_weak_vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a weak vector. Note that all weak hashes are also weak vectors.

5.14.4 Guardians

Guardians provide a way to be notified about objects that would otherwise be collected as garbage. Guarding them prevents the objects from being collected and cleanup actions can be performed on them, for example.

See R. Kent Dybvig, Carl Bruggeman, and David Eby (1993) "Guardians in a Generation-Based Garbage Collector". ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1993.

`make-guardian` [Scheme Procedure]
`scm_make_guardian` () [C Function]

Create a new guardian. A guardian protects a set of objects from garbage collection, allowing a program to apply cleanup or other actions.

`make-guardian` returns a procedure representing the guardian. Calling the guardian procedure with an argument adds the argument to the guardian's set of protected objects. Calling the guardian procedure without an argument returns one of the protected objects which are ready for garbage collection, or `#f` if no such object is available. Objects which are returned in this way are removed from the guardian.

You can put a single object into a guardian more than once and you can put a single object into more than one guardian. The object will then be returned multiple times by the guardian procedures.

An object is eligible to be returned from a guardian when it is no longer referenced from outside any guardian.

There is no guarantee about the order in which objects are returned from a guardian. If you want to impose an order on finalization actions, for example, you can do that by keeping objects alive in some global data structure until they are no longer needed for finalizing other objects.

Being an element in a weak vector, a key in a hash table with weak keys, or a value in a hash table with weak values does not prevent an object from being returned by a guardian. But as long as an object can be returned from a guardian it will not be removed from such a weak vector or hash table. In other words, a weak link does not prevent an object from being considered collectable, but being inside a guardian prevents a weak link from being broken.

A key in a weak key hash table can be thought of as having a strong reference to its associated value as long as the key is accessible. Consequently, when the key is only accessible from within a guardian, the reference from the key to the value is also considered to be coming from within a guardian. Thus, if there is no other reference to the value, it is eligible to be returned from a guardian.

5.15 Objects

`entity?` *obj* [Scheme Procedure]
`scm_entity_p` (*obj*) [C Function]

Return `#t` if *obj* is an entity.

`operator?` *obj* [Scheme Procedure]
`scm_operator_p` (*obj*) [C Function]

Return `#t` if *obj* is an operator.

`set-object-procedure!` *obj proc* [Scheme Procedure]
`scm_set_object_procedure_x` (*obj*, *proc*) [C Function]

Set the object procedure of *obj* to *proc*. *obj* must be either an entity or an operator.

`make-class-object` *metaclass layout* [Scheme Procedure]
`scm_make_class_object` (*metaclass*, *layout*) [C Function]

Create a new class object of class *metaclass*, with the slot layout specified by *layout*.

`make-subclass-object` *class layout* [Scheme Procedure]
`scm_make_subclass_object` (*class*, *layout*) [C Function]

Create a subclass object of *class*, with the slot layout specified by *layout*.

5.16 Modules

When programs become large, naming conflicts can occur when a function or global variable defined in one file has the same name as a function or global variable in another file. Even just a *similarity* between function names can cause hard-to-find bugs, since a programmer might type the wrong function name.

The approach used to tackle this problem is called *information encapsulation*, which consists of packaging functional units into a given name space that is clearly separated from other name spaces.

The language features that allow this are usually called *the module system* because programs are broken up into modules that are compiled separately (or loaded separately in an interpreter).

Older languages, like C, have limited support for name space manipulation and protection. In C a variable or function is public by default, and can be made local to a module with the `static` keyword. But you cannot reference public variables and functions from another module with different names.

More advanced module systems have become a common feature in recently designed languages: ML, Python, Perl, and Modula 3 all allow the *renaming* of objects from a foreign module, so they will not clutter the global name space.

In addition, Guile offers variables as first-class objects. They can be used for interacting with the module system.

5.16.1 provide and require

Aubrey Jaffer, mostly to support his portable Scheme library SLIB, implemented a provide/require mechanism for many Scheme implementations. Library files in SLIB *provide* a feature, and when user programs *require* that feature, the library file is loaded in.

For example, the file '`random.scm`' in the SLIB package contains the line

```
(provide 'random)
```

so to use its procedures, a user would type

```
(require 'random)
```

and they would magically become available, *but still have the same names!* So this method is nice, but not as good as a full-featured module system.

When SLIB is used with Guile, provide and require can be used to access its facilities.

5.16.2 Environments

Scheme, as defined in R5RS, does *not* have a full module system. However it does define the concept of a top-level *environment*. Such an environment maps identifiers (symbols) to Scheme objects such as procedures and lists: Section 3.1.4 [About Closure], page 24. In other words, it implements a set of *bindings*.

Environments in R5RS can be passed as the second argument to `eval` (see Section 5.13.3 [Fly Evaluation], page 291). Three procedures are defined to return environments: `scheme-report-environment`, `null-environment` and `interaction-environment` (see Section 5.13.3 [Fly Evaluation], page 291).

In addition, in Guile any module can be used as an R5RS environment, i.e., passed as the second argument to `eval`.

Note: the following two procedures are available only when the `(ice-9 r5rs)` module is loaded:

```
(use-modules (ice-9 r5rs))
```

`scheme-report-environment` *version* [Scheme Procedure]
`null-environment` *version* [Scheme Procedure]

version must be the exact integer '5', corresponding to revision 5 of the Scheme report (the Revised^5 Report on Scheme). `scheme-report-environment` returns a specifier for an environment that is empty except for all bindings defined in the report that are either required or both optional and supported by the implementation. `null-environment` returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in the report that are either required or both optional and supported by the implementation.

Currently Guile does not support values of *version* for other revisions of the report.

The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (for example `car`) is unspecified. Currently the environments specified by `scheme-report-environment` are not immutable in Guile.

5.16.3 The Guile module system

The Guile module system extends the concept of environments, discussed in the previous section, with mechanisms to define, use and customise sets of bindings.

In 1996 Tom Lord implemented a full-featured module system for Guile which allows loading Scheme source files into a private name space. This system has been available since at least Guile version 1.1.

For Guile version 1.5.0 and later, the system has been improved to have better integration from C code, more fine-grained user control over interfaces, and documentation.

Although it is anticipated that the module system implementation will change in the future, the Scheme programming interface described in this manual should be considered stable. The C programming interface is considered relatively stable, although at the time of this writing, there is still some flux.

5.16.3.1 General Information about Modules

A Guile module can be thought of as a collection of named procedures, variables and macros. More precisely, it is a set of *bindings* of symbols (names) to Scheme objects.

An environment is a mapping from identifiers (or symbols) to locations, i.e., a set of bindings. There are top-level environments and lexical environments. The environment in which a lambda is executed is remembered as part of its definition.

Within a module, all bindings are visible. Certain bindings can be declared *public*, in which case they are added to the module's so-called *export list*; this set of public bindings is called the module's *public interface* (see Section 5.16.3.3 [Creating Guile Modules], page 307).

A client module *uses* a providing module's bindings by either accessing the providing module's public interface, or by building a custom interface (and then accessing that). In a custom interface, the client module can *select* which bindings to access and can also algorithmically *rename* bindings. In contrast, when using the providing module's public interface, the entire export list is available without renaming (see Section 5.16.3.2 [Using Guile Modules], page 305).

To use a module, it must be found and loaded. All Guile modules have a unique *module name*, which is a list of one or more symbols. Examples are `(ice-9 popen)` or `(srfi srfi-11)`. When Guile searches for the code of a module, it constructs the name of the file to load by concatenating the name elements with slashes between the elements and appending a number of file name extensions from the list `%load-extensions` (see Section 5.13.4 [Loading], page 292). The resulting file name is then searched in all directories in the variable `%load-path` (see Section 5.18.1 [Build Config], page 334). For example, the `(ice-9 popen)` module would result in the filename `ice-9/popen.scm` and searched in the installation directories of Guile and in all other directories in the load path.

Every module has a so-called syntax transformer associated with it. This is a procedure which performs all syntax transformation for the time the module is read in and evaluated. When working with modules, you can manipulate the current syntax transformer using the `use-syntax` syntactic form or the `#:use-syntax` module definition option (see Section 5.16.3.3 [Creating Guile Modules], page 307).

Please note that there are some problems with the current module system you should keep in mind (see Section 5.16.3.5 [Module System Quirks], page 310). We hope to address these eventually.

5.16.3.2 Using Guile Modules

To use a Guile module is to access either its public interface or a custom interface (see Section 5.16.3.1 [General Information about Modules], page 304). Both types of access are handled by the syntactic form `use-modules`, which accepts one or more interface specifications and, upon evaluation, arranges for those interfaces to be available to the current module. This process may include locating and loading code for a given module if that code has not yet been loaded, following `%load-path` (see Section 5.18.1 [Build Config], page 334).

An *interface specification* has one of two forms. The first variation is simply to name the module, in which case its public interface is the one accessed. For example:

```
(use-modules (ice-9 popen))
```

Here, the interface specification is `(ice-9 popen)`, and the result is that the current module now has access to `open-pipe`, `close-pipe`, `open-input-pipe`, and so on (see Section 5.16.3.6 [Included Guile Modules], page 311).

Note in the previous example that if the current module had already defined `open-pipe`, that definition would be overwritten by the definition in `(ice-9 popen)`. For this reason (and others), there is a second variation of interface specification that not only names a module to be accessed, but also selects bindings from it and renames them to suit the current module's needs. For example:

```
(use-modules ((ice-9 popen)
              :select ((open-pipe . pipe-open) close-pipe)
              :renamer (symbol-prefix-proc 'unixy:)))
```

Here, the interface specification is more complex than before, and the result is that a custom interface with only two bindings is created and subsequently accessed by the current module. The mapping of old to new names is as follows:

```
(ice-9 popen) sees:             current module sees:
open-pipe                       unixy:pipe-open
```

```
     close-pipe                      unixy:close-pipe
```

This example also shows how to use the convenience procedure `symbol-prefix-proc`.

You can also directly refer to bindings in a module by using the `@` syntax. For example, instead of using the `use-modules` statement from above and writing `unixy:pipe-open` to refer to the `pipe-open` from the `(ice-9 popen)`, you could also write `(@ (ice-9 popen) open-pipe)`. Thus an alternative to the complete `use-modules` statement would be

```
(define unixy:pipe-open (@ (ice-9 popen) open-pipe))
(define unixy:close-pipe (@ (ice-9 popen) close-pipe))
```

There is also `@@`, which can be used like `@`, but does not check whether the variable that is being accessed is actually exported. Thus, `@@` can be thought of as the impolite version of `@` and should only be used as a last resort or for debugging, for example.

Note that just as with a `use-modules` statement, any module that has not yet been loaded yet will be loaded when referenced by a `@` or `@@` form.

You can also use the `@` and `@@` syntaxes as the target of a `set!` when the binding refers to a variable.

`symbol-prefix-proc` *prefix-sym* [Scheme Procedure]

Return a procedure that prefixes its arg (a symbol) with *prefix-sym*.

`use-modules` *spec ...* [syntax]

Resolve each interface specification *spec* into an interface and arrange for these to be accessible by the current module. The return value is unspecified.

spec can be a list of symbols, in which case it names a module whose public interface is found and used.

spec can also be of the form:

```
(MODULE-NAME [:select SELECTION] [:renamer RENAMER])
```

in which case a custom interface is newly created and used. *module-name* is a list of symbols, as above; *selection* is a list of selection-specs; and *renamer* is a procedure that takes a symbol and returns its new name. A selection-spec is either a symbol or a pair of symbols `(ORIG . SEEN)`, where *orig* is the name in the used module and *seen* is the name in the using module. Note that *seen* is also passed through *renamer*.

The `:select` and `:renamer` clauses are optional. If both are omitted, the returned interface has no bindings. If the `:select` clause is omitted, *renamer* operates on the used module's public interface.

Signal error if module name is not resolvable.

`use-syntax` *module-name* [syntax]

Load the module `module-name` and use its system transformer as the system transformer for the currently defined module, as well as installing it as the current system transformer.

`@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must have been exported by the module.

`@@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must not have been exported by the module. This syntax is only intended for debugging purposes or as a last resort.

5.16.3.3 Creating Guile Modules

When you want to create your own modules, you have to take the following steps:

- Create a Scheme source file and add all variables and procedures you wish to export, or which are required by the exported procedures.
- Add a `define-module` form at the beginning.
- Export all bindings which should be in the public interface, either by using `define-public` or `export` (both documented below).

`define-module` *module-name* [*options* ...] [syntax]

module-name is of the form `(hierarchy file)`. One example of this is

```
(define-module (ice-9 popen))
```

`define-module` makes this module available to Guile programs under the given *module-name*.

The *options* are keyword/value pairs which specify more about the defined module. The recognized options and their meaning is shown in the following table.

`#:use-module` *`interface-specification`*

Equivalent to a `(use-modules` *`interface-specification`*`)` (see Section 5.16.3.2 [Using Guile Modules], page 305).

`#:use-syntax` *`module`*

Use *module* when loading the currently defined module, and install it as the syntax transformer.

`#:autoload` *`module symbol-list`*

Load *module* when any of *symbol-list* are accessed. For example,

```
(define-module (my mod)
  #:autoload (srfi srfi-1) (partition delete-duplicates))
...
(if something
    (set! foo (delete-duplicates ...)))
```

When a module is autoloaded, all its bindings become available. *symbol-list* is just those that will first trigger the load.

An autoload is a good way to put off loading a big module until it's really needed, for instance for faster startup or if it will only be needed in certain circumstances.

`@` can do a similar thing (see Section 5.16.3.2 [Using Guile Modules], page 305), but in that case an `@` form must be written every time a binding from the module is used.

`#:export` *`list`*

Export all identifiers in *list* which must be a list of symbols. This is equivalent to `(export` *`list`*`)` in the module body.

`#:re-export` *list*
: Re-export all identifiers in *list* which must be a list of symbols. The symbols in *list* must be imported by the current module from other modules. This is equivalent to `re-export` below.

`#:export-syntax` *list*
: Export all identifiers in *list* which must be a list of symbols. The identifiers in *list* must refer to macros (see Section 5.8.6 [Macros], page 231) defined in the current module. This is equivalent to `(export-syntax` *list*`)` in the module body.

`#:re-export-syntax` *list*
: Re-export all identifiers in *list* which must be a list of symbols. The symbols in *list* must refer to macros imported by the current module from other modules. This is equivalent to `(re-export-syntax` *list*`)` in the module body.

`#:replace` *list*
: Export all identifiers in *list* (a list of symbols) and mark them as *replacing bindings*. In the module user's name space, this will have the effect of replacing any binding with the same name that is not also "replacing". Normally a replacement results in an "override" warning message, `#:replace` avoids that.

  This is useful for modules that export bindings that have the same name as core bindings. `#:replace`, in a sense, lets Guile know that the module *purposefully* replaces a core binding. It is important to note, however, that this binding replacement is confined to the name space of the module user. In other words, the value of the core binding in question remains unchanged for other modules.

  For instance, SRFI-39 exports a binding named `current-input-port` (see Section 6.4.18 [SRFI-39], page 452) that is a function which is upwardly compatible with the core `current-input-port` function. Therefore, SRFI-39 exports its version with `#:replace`.

  SRFI-19, on the other hand, exports its own version of `current-time` (see Section 6.4.15.2 [SRFI-19 Time], page 444) which is not compatible with the core `current-time` function (see Section 6.2.5 [Time], page 389). Therefore, SRFI-19 does not use `#:replace`.

  The `#:replace` option can also be used by a module which is intentionally producing a new special kind of environment and should override any core or other bindings already in scope. For example perhaps a logic processing environment where `<=` is an inference instead of a comparison.

  The `#:duplicates` (see below) provides fine-grain control about duplicate binding handling on the module-user side.

`#:duplicates` *list*
: Tell Guile to handle duplicate bindings for the bindings imported by the current module according to the policy defined by *list*, a list of symbols. *list* must contain symbols representing a duplicate binding handling policy chosen among the following:

`check` Raises an error when a binding is imported from more than one place.

`warn` Issue a warning when a binding is imported from more than one place and leave the responsibility of actually handling the duplication to the next duplicate binding handler.

`replace` When a new binding is imported that has the same name as a previously imported binding, then do the following:

1. If the old binding was said to be *replacing* (via the **`#:replace`** option above) and the new binding is not replacing, the keep the old binding.
2. If the old binding was not said to be replacing and the new binding is replacing, then replace the old binding with the new one.
3. If neither the old nor the new binding is replacing, then keep the old one.

`warn-override-core`
Issue a warning when a core binding is being overwritten and actually override the core binding with the new one.

`first` In case of duplicate bindings, the firstly imported binding is always the one which is kept.

`last` In case of duplicate bindings, the lastly imported binding is always the one which is kept.

`noop` In case of duplicate bindings, leave the responsibility to the next duplicate handler.

If *list* contains more than one symbol, then the duplicate binding handlers which appear first will be used first when resolving a duplicate binding situation. As mentioned above, some resolution policies may explicitly leave the responsibility of handling the duplication to the next handler in *list*.

The default duplicate binding resolution policy is given by the **`default-duplicate-binding-handler`** procedure, and is

```
(replace warn-override-core warn last)
```

`#:no-backtrace`
Tell Guile not to record information for procedure backtraces when executing the procedures in this module.

`#:pure` Create a *pure* module, that is a module which does not contain any of the standard procedure bindings except for the syntax forms. This is useful if you want to create *safe* modules, that is modules which do not know anything about dangerous procedures.

`export` *variable* ... [syntax]

Add all *variable*s (which must be symbols) to the list of exported bindings of the current module.

`define-public` ... [syntax]

Equivalent to `(begin (define foo ...) (export foo))`.

`re-export` *variable ...* [syntax]

Add all *variables* (which must be symbols) to the list of re-exported bindings of the current module. Re-exported bindings must be imported by the current module from some other module.

5.16.3.4 Module System Reflection

The previous sections have described a declarative view of the module system. You can also work with it programmatically by accessing and modifying various parts of the Scheme objects that Guile uses to implement the module system.

At any time, there is a *current module*. This module is the one where a top-level `define` and similar syntax will add new bindings. You can find other module objects with `resolve-module`, for example.

These module objects can be used as the second argument to `eval`.

`current-module` [Scheme Procedure]

Return the current module object.

`set-current-module` *module* [Scheme Procedure]

Set the current module to *module* and return the previous current module.

`resolve-module` *name* [Scheme Procedure]

Find the module named *name* and return it. When it has not already been defined, try to auto-load it. When it can't be found that way either, create an empty module. The name is a list of symbols.

`resolve-interface` *name* [Scheme Procedure]

Find the module named *name* as with `resolve-module` and return its interface. The interface of a module is also a module object, but it contains only the exported bindings.

`module-use!` *module interface* [Scheme Procedure]

Add *interface* to the front of the use-list of *module*. Both arguments should be module objects, and *interface* should very likely be a module returned by `resolve-interface`.

5.16.3.5 Module System Quirks

Although the programming interfaces are relatively stable, the Guile module system itself is still evolving. Here are some situations where usage surpasses design.

- When using a module which exports a macro definition, the other module must export all bindings the macro expansion uses, too, because the expanded code would otherwise not be able to see these definitions and issue a "variable unbound" error, or worse, would use another binding which might be present in the scope of the expansion.
- When two or more used modules export bindings with the same names, the last accessed module wins, and the exported binding of that last module will silently be used. This might lead to hard-to-find errors because wrong procedures or variables are used. To avoid this kind of *name-clash* situation, use a custom interface specification (see

Section 5.16.3.2 [Using Guile Modules], page 305). (We include this entry for the possible benefit of users of Guile versions previous to 1.5.0, when custom interfaces were added to the module system.)

- [Add other quirks here.]

5.16.3.6 Included Guile Modules

Some modules are included in the Guile distribution; here are references to the entries in this manual which describe them in more detail:

boot-9
: boot-9 is Guile's initialization module, and it is always loaded when Guile starts up.

(ice-9 debug)
: Mikael Djurfeldt's source-level debugging support for Guile (see Section 3.4 [Debugging Features], page 40).

(ice-9 expect)
: Actions based on matching input from a port (see Section 6.13 [Expect], page 480).

(ice-9 format)
: Formatted output in the style of Common Lisp (see Section 6.8 [Formatted Output], page 463).

(ice-9 ftw)
: File tree walker (see Section 6.9 [File Tree Walk], page 473).

(ice-9 getopt-long)
: Command line option processing (see Section 6.3 [getopt-long], page 418).

(ice-9 history)
: Refer to previous interactive expressions (see Section 6.6 [Value History], page 461).

(ice-9 popen)
: Pipes to and from child processes (see Section 6.2.10 [Pipes], page 401).

(ice-9 pretty-print)
: Nicely formatted output of Scheme expressions and objects (see Section 6.7 [Pretty Printing], page 462).

(ice-9 q)
: First-in first-out queues (see Section 6.10 [Queues], page 475).

(ice-9 rdelim)
: Line- and character-delimited input (see Section 5.12.6 [Line/Delimited], page 276).

(ice-9 readline)
: `readline` interactive command line editing (see Section 6.5 [Readline Support], page 457).

(ice-9 receive)
: Multiple-value handling with `receive` (see Section 5.11.6 [Multiple Values], page 257).

(ice-9 regex)
Regular expression matching (see Section 5.5.6 [Regular Expressions], page 146).

(ice-9 rw) Block string input/output (see Section 5.12.7 [Block Reading and Writing], page 277).

(ice-9 streams)
Sequence of values calculated on-demand (see Section 6.11 [Streams], page 477).

(ice-9 syncase)
R5RS `syntax-rules` macro system (see Section 5.8.7 [Syntax Rules], page 232).

(ice-9 threads)
Guile's support for multi threaded execution (see Section 5.17 [Scheduling], page 323).

(ice-9 documentation)
Online documentation (REFFIXME).

(srfi srfi-1)
A library providing a lot of useful list and pair processing procedures (see Section 6.4.3 [SRFI-1], page 424).

(srfi srfi-2)
Support for `and-let*` (see Section 6.4.4 [SRFI-2], page 437).

(srfi srfi-4)
Support for homogeneous numeric vectors (see Section 6.4.5 [SRFI-4], page 438).

(srfi srfi-6)
Support for some additional string port procedures (see Section 6.4.6 [SRFI-6], page 438).

(srfi srfi-8)
Multiple-value handling with `receive` (see Section 6.4.7 [SRFI-8], page 439).

(srfi srfi-9)
Record definition with `define-record-type` (see Section 6.4.8 [SRFI-9], page 439).

(srfi srfi-10)
Read hash extension `#,()` (see Section 6.4.9 [SRFI-10], page 440).

(srfi srfi-11)
Multiple-value handling with `let-values` and `let-values*` (see Section 6.4.10 [SRFI-11], page 442).

(srfi srfi-13)
String library (see Section 6.4.11 [SRFI-13], page 442).

(srfi srfi-14)
Character-set library (see Section 6.4.12 [SRFI-14], page 442).

(srfi srfi-16)
: `case-lambda` procedures of variable arity (see Section 6.4.13 [SRFI-16], page 442).

(srfi srfi-17)
: Getter-with-setter support (see Section 6.4.14 [SRFI-17], page 443).

(srfi srfi-19)
: Time/Date library (see Section 6.4.15 [SRFI-19], page 443).

(srfi srfi-26)
: Convenient syntax for partial application (see Section 6.4.16 [SRFI-26], page 450)

(srfi srfi-31)
: `rec` convenient recursive expressions (see Section 6.4.17 [SRFI-31], page 452)

(ice-9 slib)
: This module contains hooks for using Aubrey Jaffer's portable Scheme library SLIB from Guile (see Section 6.1 [SLIB], page 372).

5.16.3.7 Accessing Modules from C

The last sections have described how modules are used in Scheme code, which is the recommended way of creating and accessing modules. You can also work with modules from C, but it is more cumbersome.

The following procedures are available.

`SCM scm_current_module` () [C Procedure]
: Return the module that is the *current module*.

`SCM scm_set_current_module` (*SCM* `module`) [C Procedure]
: Set the current module to *module* and return the previous current module.

`SCM scm_c_call_with_current_module` (*SCM* `module`, *SCM* (*`func`)(*void* *), *void* *`data`) [C Procedure]
: Call *func* and make *module* the current module during the call. The argument *data* is passed to *func*. The return value of `scm_c_call_with_current_module` is the return value of *func*.

`SCM scm_c_lookup` (*const char* *`name`) [C Procedure]
: Return the variable bound to the symbol indicated by *name* in the current module. If there is no such binding or the symbol is not bound to a variable, signal an error.

`SCM scm_lookup` (*SCM* `name`) [C Procedure]
: Like `scm_c_lookup`, but the symbol is specified directly.

`SCM scm_c_module_lookup` (*SCM* `module`, *const char* *`name`) [C Procedure]
`SCM scm_module_lookup` (*SCM* `module`, *SCM* `name`) [C Procedure]
: Like `scm_c_lookup` and `scm_lookup`, but the specified module is used instead of the current one.

`SCM scm_c_define` (*const char* *`*name`*, *SCM* *`val`*) [C Procedure]

Bind the symbol indicated by *name* to a variable in the current module and set that variable to *val*. When *name* is already bound to a variable, use that. Else create a new variable.

`SCM scm_define` (*SCM* *`name`*, *SCM* *`val`*) [C Procedure]

Like `scm_c_define`, but the symbol is specified directly.

`SCM scm_c_module_define` (*SCM* *`module`*, *const char* *`*name`*, *SCM* *`val`*) [C Procedure]

`SCM scm_module_define` (*SCM* *`module`*, *SCM* *`name`*, *SCM* *`val`*) [C Procedure]

Like `scm_c_define` and `scm_define`, but the specified module is used instead of the current one.

`SCM scm_module_reverse_lookup` (*SCM* *`module`*, *SCM* *`variable`*) [C Procedure]

Find the symbol that is bound to *variable* in *module*. When no such binding is found, return *#f*.

`SCM scm_c_define_module` (*const char* *`*name`*, *void* (*`*init`*)(*void* *), *void* *`*data`*) [C Procedure]

Define a new module named *name* and make it current while *init* is called, passing it *data*. Return the module.

The parameter *name* is a string with the symbols that make up the module name, separated by spaces. For example, ‘`"foo bar"`’ names the module ‘`(foo bar)`’.

When there already exists a module named *name*, it is used unchanged, otherwise, an empty module is created.

`SCM scm_c_resolve_module` (*const char* *`*name`*) [C Procedure]

Find the module name *name* and return it. When it has not already been defined, try to auto-load it. When it can’t be found that way either, create an empty module. The name is interpreted as for `scm_c_define_module`.

`SCM scm_resolve_module` (*SCM* *`name`*) [C Procedure]

Like `scm_c_resolve_module`, but the name is given as a real list of symbols.

`SCM scm_c_use_module` (*const char* *`*name`*) [C Procedure]

Add the module named *name* to the uses list of the current module, as with `(use-modules` *`name`*`)`. The name is interpreted as for `scm_c_define_module`.

`SCM scm_c_export` (*const char* *`*name`*, *...*) [C Procedure]

Add the bindings designated by *name, ...* to the public interface of the current module. The list of names is terminated by `NULL`.

5.16.4 Dynamic Libraries

Most modern Unices have something called *shared libraries*. This ordinarily means that they have the capability to share the executable image of a library between several running programs to save memory and disk space. But generally, shared libraries give a lot of additional flexibility compared to the traditional static libraries. In fact, calling them ‘dynamic’ libraries is as correct as calling them ‘shared’.

Shared libraries really give you a lot of flexibility in addition to the memory and disk space savings. When you link a program against a shared library, that library is not closely incorporated into the final executable. Instead, the executable of your program only contains enough information to find the needed shared libraries when the program is actually run. Only then, when the program is starting, is the final step of the linking process performed. This means that you need not recompile all programs when you install a new, only slightly modified version of a shared library. The programs will pick up the changes automatically the next time they are run.

Now, when all the necessary machinery is there to perform part of the linking at run-time, why not take the next step and allow the programmer to explicitly take advantage of it from within his program? Of course, many operating systems that support shared libraries do just that, and chances are that Guile will allow you to access this feature from within your Scheme programs. As you might have guessed already, this feature is called *dynamic linking*.[1]

As with many aspects of Guile, there is a low-level way to access the dynamic linking apparatus, and a more high-level interface that integrates dynamically linked libraries into the module system.

5.16.4.1 Low level dynamic linking

When using the low level procedures to do your dynamic linking, you have complete control over which library is loaded when and what gets done with it.

`dynamic-link` *library* [Scheme Procedure]
`scm_dynamic_link` (*library*) [C Function]

> Find the shared library denoted by *library* (a string) and link it into the running Guile application. When everything works out, return a Scheme object suitable for representing the linked object file. Otherwise an error is thrown. How object files are searched is system dependent.
>
> Normally, *library* is just the name of some shared library file that will be searched for in the places where shared libraries usually reside, such as in '`/usr/lib`' and '`/usr/local/lib`'.

`dynamic-object?` *obj* [Scheme Procedure]
`scm_dynamic_object_p` (*obj*) [C Function]

> Return `#t` if *obj* is a dynamic library handle, or `#f` otherwise.

`dynamic-unlink` *dobj* [Scheme Procedure]
`scm_dynamic_unlink` (*dobj*) [C Function]

> Unlink the indicated object file from the application. The argument *dobj* must have been obtained by a call to `dynamic-link`. After `dynamic-unlink` has been called on *dobj*, its content is no longer accessible.

`dynamic-func` *name dobj* [Scheme Procedure]

[1] Some people also refer to the final linking stage at program startup as 'dynamic linking', so if you want to make yourself perfectly clear, it is probably best to use the more technical term *dlopening*, as suggested by Gordon Matzigkeit in his libtool documentation.

`scm_dynamic_func` (*name, dobj*) [C Function]

Search the dynamic object *dobj* for the C function indicated by the string *name* and return some Scheme handle that can later be used with `dynamic-call` to actually call the function.

Regardless whether your C compiler prepends an underscore '`_`' to the global names in a program, you should **not** include this underscore in *function*. Guile knows whether the underscore is needed or not and will add it when necessary.

`dynamic-call` *func dobj* [Scheme Procedure]
`scm_dynamic_call` (*func, dobj*) [C Function]

Call the C function indicated by *func* and *dobj*. The function is passed no arguments and its return value is ignored. When *function* is something returned by `dynamic-func`, call that function and ignore *dobj*. When *func* is a string , look it up in *dynobj*; this is equivalent to

```
(dynamic-call (dynamic-func func dobj) #f)
```

Interrupts are deferred while the C function is executing (with `SCM_DEFER_INTS/SCM_ALLOW_INTS`).

`dynamic-args-call` *func dobj args* [Scheme Procedure]
`scm_dynamic_args_call` (*func, dobj, args*) [C Function]

Call the C function indicated by *func* and *dobj*, just like `dynamic-call`, but pass it some arguments and return its return value. The C function is expected to take two arguments and return an `int`, just like `main`:

```
int c_func (int argc, char **argv);
```

The parameter *args* must be a list of strings and is converted into an array of `char *`. The array is passed in *argv* and its size in *argc*. The return value is converted to a Scheme number and returned from the call to `dynamic-args-call`.

When dynamic linking is disabled or not supported on your system, the above functions throw errors, but they are still available.

Here is a small example that works on GNU/Linux:

```
(define libc-obj (dynamic-link "libc.so"))
libc-obj
⇒ #<dynamic-object "libc.so">
(dynamic-args-call 'rand libc-obj '())
⇒ 269167349
(dynamic-unlink libc-obj)
libc-obj
⇒ #<dynamic-object "libc.so" (unlinked)>
```

As you can see, after calling `dynamic-unlink` on a dynamically linked library, it is marked as '`(unlinked)`' and you are no longer able to use it with `dynamic-call`, etc. Whether the library is really removed from you program is system-dependent and will generally not happen when some other parts of your program still use it. In the example above, `libc` is almost certainly not removed from your program because it is badly needed by almost everything.

The functions to call a function from a dynamically linked library, `dynamic-call` and `dynamic-args-call`, are not very powerful. They are mostly intended to be used for calling specially written initialization functions that will then add new primitives to Guile. For example, we do not expect that you will dynamically link '`libX11`' with `dynamic-link` and then construct a beautiful graphical user interface just by using `dynamic-call` and `dynamic-args-call`. Instead, the usual way would be to write a special Guile<->X11 glue library that has intimate knowledge about both Guile and X11 and does whatever is necessary to make them inter-operate smoothly. This glue library could then be dynamically linked into a vanilla Guile interpreter and activated by calling its initialization function. That function would add all the new types and primitives to the Guile interpreter that it has to offer.

From this setup the next logical step is to integrate these glue libraries into the module system of Guile so that you can load new primitives into a running system just as you can load new Scheme code.

There is, however, another possibility to get a more thorough access to the functions contained in a dynamically linked library. Anthony Green has written '`libffi`', a library that implements a *foreign function interface* for a number of different platforms. With it, you can extend the Spartan functionality of `dynamic-call` and `dynamic-args-call` considerably. There is glue code available in the Guile contrib archive to make '`libffi`' accessible from Guile.

5.16.4.2 Putting Compiled Code into Modules

The new primitives that you add to Guile with `scm_c_define_gsubr` (see Section 5.8.2 [Primitive Procedures], page 226) or with any of the other mechanisms are placed into the `(guile-user)` module by default. However, it is also possible to put new primitives into other modules.

The mechanism for doing so is not very well thought out and is likely to change when the module system of Guile itself is revised, but it is simple and useful enough to document it as it stands.

What `scm_c_define_gsubr` and the functions used by the snarfer really do is to add the new primitives to whatever module is the *current module* when they are called. This is analogous to the way Scheme code is put into modules: the `define-module` expression at the top of a Scheme source file creates a new module and makes it the current module while the rest of the file is evaluated. The `define` expressions in that file then add their new definitions to this current module.

Therefore, all we need to do is to make sure that the right module is current when calling `scm_c_define_gsubr` for our new primitives.

5.16.4.3 Dynamic Linking and Compiled Code Modules

The most interesting application of dynamically linked libraries is probably to use them for providing *compiled code modules* to Scheme programs. As much fun as programming in Scheme is, every now and then comes the need to write some low-level C stuff to make Scheme even more fun.

Not only can you put these new primitives into their own module (see the previous section), you can even put them into a shared library that is only then linked to your running Guile image when it is actually needed.

An example will hopefully make everything clear. Suppose we want to make the Bessel functions of the C library available to Scheme in the module '`(math bessel)`'. First we need to write the appropriate glue code to convert the arguments and return values of the functions from Scheme to C and back. Additionally, we need a function that will add them to the set of Guile primitives. Because this is just an example, we will only implement this for the `j0` function.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_double2num (j0 (scm_num2dbl (x, "j0")));
}

void
init_math_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

We can already try to bring this into action by manually calling the low level functions for performing dynamic linking. The C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux, please refer to the `libtool` documentation for how to create dynamically linkable libraries portably.

```
gcc -shared -o libbessel.so -fPIC bessel.c
```

Now fire up Guile:

```
(define bessel-lib (dynamic-link "./libbessel.so"))
(dynamic-call "init_math_bessel" bessel-lib)
(j0 2)
⇒ 0.223890779141236
```

The filename '`./libbessel.so`' should be pointing to the shared library produced with the `gcc` command above, of course. The second line of the Guile interaction will call the `init_math_bessel` function which in turn will register the C function `j0_wrapper` with the Guile interpreter under the name `j0`. This function becomes immediately available and we can call it from Scheme.

Fun, isn't it? But we are only half way there. This is what `apropos` has to say about `j0`:

```
(apropos "j0")
⊣ (guile-user): j0     #<primitive-procedure j0>
```

As you can see, `j0` is contained in the root module, where all the other Guile primitives like `display`, etc live. In general, a primitive is put into whatever module is the *current module* at the time `scm_c_define_gsubr` is called.

A compiled module should have a specially named *module init function*. Guile knows about this special name and will call that function automatically after having linked in the shared library. For our example, we replace `init_math_bessel` with the following code in '`bessel.c`':

```
void
init_math_bessel (void *unused)
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
  scm_c_export ("j0", NULL);
}
```

```
void
scm_init_math_bessel_module ()
{
  scm_c_define_module ("math bessel", init_math_bessel, NULL);
}
```

The general pattern for the name of a module init function is: '`scm_init_`', followed by the name of the module where the individual hierarchical components are concatenated with underscores, followed by '`_module`'.

After '`libbessel.so`' has been rebuilt, we need to place the shared library into the right place.

Once the module has been correctly installed, it should be possible to use it like this:

```
guile> (load-extension "./libbessel.so" "scm_init_math_bessel_module")
guile> (use-modules (math bessel))
guile> (j0 2)
0.223890779141236
guile> (apropos "j0")
⊣ (math bessel): j0      #<primitive-procedure j0>
```

That's it!

`load-extension` *lib init* [Scheme Procedure]
`scm_load_extension` (*lib*, *init*) [C Function]

Load and initialize the extension designated by LIB and INIT. When there is no pre-registered function for LIB/INIT, this is equivalent to

```
(dynamic-call INIT (dynamic-link LIB))
```

When there is a pre-registered function, that function is called instead.

Normally, there is no pre-registered function. This option exists only for situations where dynamic linking is unavailable or unwanted. In that case, you would statically link your program with the desired library, and register its init function right after Guile has been initialized.

LIB should be a string denoting a shared library without any file type suffix such as ".so". The suffix is provided automatically. It should also not contain any directory components. Libraries that implement Guile Extensions should be put into the normal locations for shared libraries. We recommend to use the naming convention libguile-bla-blum for a extension related to a module '(bla blum)'.

The normal way for a extension to be used is to write a small Scheme file that defines a module, and to load the extension into this module. When the module is auto-loaded, the extension is loaded as well. For example,

```
(define-module (bla blum))

(load-extension "libguile-bla-blum" "bla_init_blum")
```

5.16.4.4 Compiled Code Installation

The simplest way to write a module using compiled C code is

```
(define-module (foo bar))
(load-extension "foobar-c-code" "foo_bar_init")
```

When loaded with `(use-modules (foo bar))`, the `load-extension` call looks for the '`foobar-c-code.so`' (etc) object file in the standard system locations, such as '`/usr/lib`' or '`/usr/local/lib`'.

If someone installs your module to a non-standard location then the object file won't be found. You can address this by inserting the install location in the '`foo/bar.scm`' file. This is convenient for the user and also guarantees the intended object is read, even if stray older or newer versions are in the loader's path.

The usual way to specify an install location is with a `prefix` at the configure stage, for instance '`./configure prefix=/opt`' results in library files as say '`/opt/lib/foobar-c-code.so`'. When using Autoconf (see section "Introduction" in *The GNU Autoconf Manual*), the library location is in a `libdir` variable. Its value is intended to be expanded by `make`, and can by substituted into a source file like '`foo.scm.in`'

```
(define-module (foo bar))
(load-extension "XXlibdirXX/foobar-c-code" "foo_bar_init")
```

with the following in a '`Makefile`', using `sed` (see section "Introduction" in *SED*),

```
foo.scm: foo.scm.in
        sed 's|XXlibdirXX|$(libdir)|' <foo.scm.in >foo.scm
```

The actual pattern `XXlibdirXX` is arbitrary, it's only something which doesn't otherwise occur. If several modules need the value, it can be easier to create one '`foo/config.scm`' with a define of the `libdir` location, and use that as required.

```
(define-module (foo config))
(define-public foo-config-libdir "XXlibdirXX"")
```

Such a file might have other locations too, for instance a data directory for auxiliary files, or `localedir` if the module has its own `gettext` message catalogue (see Section 5.20 [Internationalization], page 345).

When installing multiple C code objects, it can be convenient to put them in a subdirectory of `libdir`, thus giving for example `/usr/lib/foo/some-obj.so`. If the objects are only meant to be used through the module, then a subdirectory keeps them out of sight.

It will be noted all of the above requires that the Scheme code to be found in `%load-path` (see Section 5.18.1 [Build Config], page 334). Presently it's left up to the system administrator or each user to augment that path when installing Guile modules in non-default locations. But having reached the Scheme code, that code should take care of hitting any of its own private files etc.

Presently there's no convention for having a Guile version number in module C code filenames or directories. This is primarily because there's no established principles for two versions of Guile to be installed under the same prefix (eg. two both under '`/usr`'). Assuming upward compatibility is maintained then this should be unnecessary, and if compatibility is not maintained then it's highly likely a package will need to be revisited anyway.

The present suggestion is that modules should assume when they're installed under a particular `prefix` that there's a single version of Guile there, and the `guile-config` at build time has the necessary information about it. C code or Scheme code might adapt itself accordingly (allowing for features not available in an older version for instance).

5.16.5 Variables

Each module has its own hash table, sometimes known as an *obarray*, that maps the names defined in that module to their corresponding variable objects.

A variable is a box-like object that can hold any Scheme value. It is said to be *undefined* if its box holds a special Scheme value that denotes undefined-ness (which is different from all other Scheme values, including for example `#f`); otherwise the variable is *defined*.

On its own, a variable object is anonymous. A variable is said to be *bound* when it is associated with a name in some way, usually a symbol in a module obarray. When this happens, the relationship is mutual: the variable is bound to the name (in that module), and the name (in that module) is bound to the variable.

(That's the theory, anyway. In practice, defined-ness and bound-ness sometimes get confused, because Lisp and Scheme implementations have often conflated — or deliberately drawn no distinction between — a name that is unbound and a name that is bound to a variable whose value is undefined. We will try to be clear about the difference and explain any confusion where it is unavoidable.)

Variables do not have a read syntax. Most commonly they are created and bound implicitly by `define` expressions: a top-level `define` expression of the form

```
(define name value)
```

creates a variable with initial value *value* and binds it to the name *name* in the current module. But they can also be created dynamically by calling one of the constructor procedures `make-variable` and `make-undefined-variable`.

First-class variables are especially useful for interacting with the current module system (see Section 5.16.3 [The Guile module system], page 304).

`make-undefined-variable` [Scheme Procedure]
`scm_make_undefined_variable` () [C Function]
Return a variable that is initially unbound.

`make-variable` *init* [Scheme Procedure]
`scm_make_variable` (*init*) [C Function]
Return a variable initialized to value *init*.

`variable-bound?` *var* [Scheme Procedure]
`scm_variable_bound_p` (*var*) [C Function]
Return `#t` iff *var* is bound to a value. Throws an error if *var* is not a variable object.

`variable-ref` *var* [Scheme Procedure]
`scm_variable_ref` (*var*) [C Function]
Dereference *var* and return its value. *var* must be a variable object; see `make-variable` and `make-undefined-variable`.

`variable-set!` *var val* [Scheme Procedure]
`scm_variable_set_x` (*var*, *val*) [C Function]
Set the value of the variable *var* to *val*. *var* must be a variable object, *val* can be any value. Return an unspecified value.

`variable?` *obj* [Scheme Procedure]
`scm_variable_p` (*obj*) [C Function]

Return `#t` iff *obj* is a variable object, else return `#f`.

5.17 Threads, Mutexes, Asyncs and Dynamic Roots

[FIXME: This is pasted in from Tom Lord's original guile.texi chapter plus the Cygnus programmer's manual; it should be *very* carefully reviewed and largely reorganized.]

5.17.1 Arbiters

Arbiters are synchronization objects, they can be used by threads to control access to a shared resource. An arbiter can be locked to indicate a resource is in use, and unlocked when done.

An arbiter is like a light-weight mutex (see Section 5.17.5 [Mutexes and Condition Variables], page 326). It uses less memory and may be faster, but there's no way for a thread to block waiting on an arbiter, it can only test and get the status returned.

`make-arbiter` *name* [Scheme Procedure]
`scm_make_arbiter` (*name*) [C Function]

> Return an object of type arbiter and name *name*. Its state is initially unlocked. Arbiters are a way to achieve process synchronization.

`try-arbiter` *arb* [Scheme Procedure]
`scm_try_arbiter` (*arb*) [C Function]
`scm_try_arbiter` (*arb*) [C Function]

> If *arb* is unlocked, then lock it and return `#t`. If *arb* is already locked, then do nothing and return `#f`.

`release-arbiter` *arb* [Scheme Procedure]
`scm_release_arbiter` (*arb*) [C Function]

> If *arb* is locked, then unlock it and return `#t`. If *arb* is already unlocked, then do nothing and return `#f`.
>
> Typical usage is for the thread which locked an arbiter to later release it, but that's not required, any thread can release it.

5.17.2 Asyncs

Asyncs are a means of deferring the excution of Scheme code until it is safe to do so.

Guile provides two kinds of asyncs that share the basic concept but are otherwise quite different: system asyncs and user asyncs. System asyncs are integrated into the core of Guile and are executed automatically when the system is in a state to allow the execution of Scheme code. For example, it is not possible to execute Scheme code in a POSIX signal handler, but such a signal handler can queue a system async to be executed in the near future, when it is safe to do so.

System asyncs can also be queued for threads other than the current one. This way, you can cause threads to asynchronously execute arbitrary code.

User asyncs offer a convenient means of queueing procedures for future execution and triggering this execution. They will not be executed automatically.

5.17.2.1 System asyncs

To cause the future asynchronous execution of a procedure in a given thread, use `system-async-mark`.

Automatic invocation of system asyncs can be temporarily disabled by calling `call-with-blocked-asyncs`. This function works by temporarily increasing the *async blocking level* of the current thread while a given procedure is running. The blocking level starts out at zero, and whenever a safe point is reached, a blocking level greater than zero will prevent the execution of queued asyncs.

Analogously, the procedure `call-with-unblocked-asyncs` will temporarily decrease the blocking level of the current thread. You can use it when you want to disable asyncs by default and only allow them temporarily.

In addition to the C versions of `call-with-blocked-asyncs` and `call-with-unblocked-asyncs`, C code can use `scm_dynwind_block_asyncs` and `scm_dynwind_unblock_asyncs` inside a *dynamic context* (see Section 5.11.9 [Dynamic Wind], page 266) to block or unblock system asyncs temporarily.

`system-async-mark` *proc* [*thread*] [Scheme Procedure]
`scm_system_async_mark` (*proc*) [C Function]
`scm_system_async_mark_for_thread` (*proc, thread*) [C Function]

> Mark *proc* (a procedure with zero arguments) for future execution in *thread*. When *proc* has already been marked for *thread* but has not been executed yet, this call has no effect. When *thread* is omitted, the thread that called `system-async-mark` is used.
>
> This procedure is not safe to be called from signal handlers. Use `scm_sigaction` or `scm_sigaction_for_thread` to install signal handlers.

`call-with-blocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_blocked_asyncs` (*proc*) [C Function]
`void *` *scm_c_call_with_blocked_asyncs* (*void * (*proc) (void *data), void *data*) [C Function]

> Call *proc* and block the execution of system asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`call-with-unblocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_unblocked_asyncs` (*proc*) [C Function]
`void *` *scm_c_call_with_unblocked_asyncs* (*void *(*p) (void *d), void *d*) [C Function]

> Call *proc* and unblock the execution of system asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void scm_dynwind_block_asyncs ()` [C Function]

> This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 5.11.9 [Dynamic Wind], page 266). During the dynwind context, asyncs are blocked by one level.

`void scm_dynwind_unblock_asyncs ()` [C Function]

> This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 5.11.9 [Dynamic Wind], page 266). During the dynwind context, asyncs are unblocked by one level.

5.17.2.2 User asyncs

A user async is a pair of a thunk (a parameterless procedure) and a mark. Setting the mark on a user async will cause the thunk to be executed when the user async is passed to `run-asyncs`. Setting the mark more than once is satisfied by one execution of the thunk.

User asyncs are created with `async`. They are marked with `async-mark`.

`async` *thunk* [Scheme Procedure]
`scm_async` (*thunk*) [C Function]

Create a new user async for the procedure *thunk*.

`async-mark` *a* [Scheme Procedure]
`scm_async_mark` (*a*) [C Function]

Mark the user async *a* for future execution.

`run-asyncs` *list_of_a* [Scheme Procedure]
`scm_run_asyncs` (*list_of_a*) [C Function]

Execute all thunks from the marked asyncs of the list *list_of_a*.

5.17.3 Continuation Barriers

The non-local flow of control caused by continuations might sometimes not be wanted. You can use `with-continuation-barrier` etc to errect fences that continuations can not pass.

`with-continuation-barrier` *proc* [Scheme Procedure]
`scm_with_continuation_barrier` (*proc*) [C Function]

Call *proc* and return its result. Do not allow the invocation of continuations that would leave or enter the dynamic extent of the call to `with-continuation-barrier`. Such an attempt causes an error to be signaled.

Throws (such as errors) that are not caught from within *proc* are caught by `with-continuation-barrier`. In that case, a short message is printed to the current error port and `#f` is returned.

Thus, `with-continuation-barrier` returns exactly once.

`void * scm_c_with_continuation_barrier` (*void *(*func) (void *), void *data*) [C Function]

Like `scm_with_continuation_barrier` but call *func* on *data*. When an error is caught, `NULL` is returned.

5.17.4 Threads

`all-threads` [Scheme Procedure]
`scm_all_threads` () [C Function]

Return a list of all threads.

`current-thread` [Scheme Procedure]
`scm_current_thread` () [C Function]

Return the thread that called this function.

`call-with-new-thread` *thunk* [*handler*] [Scheme Procedure]

Call `thunk` in a new thread and with a new dynamic state, returning the new thread. The procedure *thunk* is called via `with-continuation-barrier`.

When *handler* is specified, then *thunk* is called from within a `catch` with tag `#t` that has *handler* as its handler. This catch is established inside the continuation barrier.

Once *thunk* or *handler* returns, the return value is made the *exit value* of the thread and the thread is terminated.

`SCM scm_spawn_thread` (*scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]

Call *body* in a new thread, passing it *body_data*, returning the new thread. The function *body* is called via `scm_c_with_continuation_barrier`.

When *handler* is non-`NULL`, *body* is called via `scm_internal_catch` with tag `SCM_BOOL_T` that has *handler* and *handler_data* as the handler and its data. This catch is established inside the continuation barrier.

Once *body* or *handler* returns, the return value is made the *exit value* of the thread and the thread is terminated.

`join-thread` *thread* [Scheme Procedure]

Wait for *thread* to terminate and return its exit value. Threads that have not been created with `call-with-new-thread` or `scm_spawn_thread` have an exit value of `#f`.

`thread-exited?` *thread* [Scheme Procedure]
`scm_thread_exited_p` (*thread*) [C Function]

Return `#t` iff *thread* has exited.

`yield` [Scheme Procedure]

If one or more threads are waiting to execute, calling yield forces an immediate context switch to one of them. Otherwise, yield has no effect.

Higher level thread procedures are available by loading the `(ice-9 threads)` module. These provide standardized thread creation.

`make-thread` *proc* [*args. . .*] [macro]

Apply *proc* to *args* in a new thread formed by `call-with-new-thread` using a default error handler that display the error to the current error port. The *args. . .* expressions are evaluated in the new thread.

`begin-thread` *first* [*rest. . .*] [macro]

Evaluate forms *first* and *rest* in a new thread formed by `call-with-new-thread` using a default error handler that display the error to the current error port.

5.17.5 Mutexes and Condition Variables

A mutex is a thread synchronization object, it can be used by threads to control access to a shared resource. A mutex can be locked to indicate a resource is in use, and other threads can then block on the mutex to wait for the resource (or can just test and do something else if not available). “Mutex” is short for “mutual exclusion”.

There are two types of mutexes in Guile, "standard" and "recursive". They're created by `make-mutex` and `make-recursive-mutex` respectively, the operation functions are then common to both.

Note that for both types of mutex there's no protection against a "deadly embrace". For instance if one thread has locked mutex A and is waiting on mutex B, but another thread owns B and is waiting on A, then an endless wait will occur (in the current implementation). Acquiring requisite mutexes in a fixed order (like always A before B) in all threads is one way to avoid such problems.

`make-mutex` [Scheme Procedure]
`scm_make_mutex ()` [C Function]
Return a new standard mutex. It is initially unlocked.

`make-recursive-mutex` [Scheme Procedure]
`scm_make_recursive_mutex ()` [C Function]
Create a new recursive mutex. It is initialloy unlocked.

`lock-mutex` *mutex* [Scheme Procedure]
`scm_lock_mutex` (*mutex*) [C Function]
Lock *mutex*. If the mutex is already locked by another thread then block and return only when *mutex* has been acquired.

For standard mutexes (`make-mutex`), and error is signalled if the thread has itself already locked *mutex*.

For a recursive mutex (`make-recursive-mutex`), if the thread has itself already locked *mutex*, then a further `lock-mutex` call increments the lock count. An additional `unlock-mutex` will be required to finally release.

When a system async (see Section 5.17.2.1 [System asyncs], page 323) is activated for a thread blocked in `lock-mutex`, the wait is interrupted and the async is executed. When the async returns, the wait resumes.

`void scm_dynwind_lock_mutex` (*SCM mutex*) [C Function]
Arrange for *mutex* to be locked whenever the current dynwind context is entered and to be unlocked when it is exited.

`try-mutex` *mx* [Scheme Procedure]
`scm_try_mutex` (*mx*) [C Function]
Try to lock *mutex* as per `lock-mutex`. If *mutex* can be acquired immediately then this is done and the return is `#t`. If *mutex* is locked by some other thread then nothing is done and the return is `#f`.

`unlock-mutex` *mutex* [Scheme Procedure]
`scm_unlock_mutex` (*mutex*) [C Function]
Unlock *mutex*. An error is signalled if *mutex* is not locked by the calling thread.

`make-condition-variable` [Scheme Procedure]
`scm_make_condition_variable ()` [C Function]
Return a new condition variable.

`wait-condition-variable` *condvar mutex* [*time*] [Scheme Procedure]
`scm_wait_condition_variable` (*condvar, mutex, time*) [C Function]

Wait until *condvar* has been signalled. While waiting, *mutex* is atomically unlocked (as with `unlock-mutex`) and is locked again when this function returns. When *time* is given, it specifies a point in time where the waiting should be aborted. It can be either a integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, `#f` is returned. When the condition variable has in fact been signalled, `#t` is returned. The mutex is re-locked in any case before `wait-condition-variable` returns.

When a system async is activated for a thread that is blocked in a call to `wait-condition-variable`, the waiting is interrupted, the mutex is locked, and the async is executed. When the async returns, the mutex is unlocked again and the waiting is resumed. When the thread block while re-acquiring the mutex, execution of asyncs is blocked.

`signal-condition-variable` *condvar* [Scheme Procedure]
`scm_signal_condition_variable` (*condvar*) [C Function]

Wake up one thread that is waiting for *condvar*.

`broadcast-condition-variable` *condvar* [Scheme Procedure]
`scm_broadcast_condition_variable` (*condvar*) [C Function]

Wake up all threads that are waiting for *condvar*.

The following are higher level operations on mutexes. These are available from

```
(use-modules (ice-9 threads))
```

`with-mutex` *mutex* [*body...*] [macro]

Lock *mutex*, evaluate the *body* forms, then unlock *mutex*. The return value is the return from the last *body* form.

The lock, body and unlock form the branches of a `dynamic-wind` (see Section 5.11.9 [Dynamic Wind], page 266), so *mutex* is automatically unlocked if an error or new continuation exits *body*, and is re-locked if *body* is re-entered by a captured continuation.

`monitor` *body...* [macro]

Evaluate the *body* forms, with a mutex locked so only one thread can execute that code at any one time. The return value is the return from the last *body* form.

Each `monitor` form has its own private mutex and the locking and evaluation is as per `with-mutex` above. A standard mutex (`make-mutex`) is used, which means *body* must not recursively re-enter the `monitor` form.

The term “monitor” comes from operating system theory, where it means a particular bit of code managing access to some resource and which only ever executes on behalf of one process at any one time.

5.17.6 Blocking in Guile Mode

A thread must not block outside of a libguile function while it is in guile mode. The following functions can be used to temporily leave guile mode or to perform some common blocking operations in a supported way.

`void * scm_without_guile` (*void *(*func) (void *), void *data*) [C Function]

Leave guile mode, call *func* on *data*, enter guile mode and return the result of calling *func*.

While a thread has left guile mode, it must not call any libguile functions except `scm_with_guile` or `scm_without_guile` and must not use any libguile macros. Also, local variables of type `SCM` that are allocated while not in guile mode are not protected from the garbage collector.

When used from non-guile mode, calling `scm_without_guile` is still allowed: it simply calls *func*. In that way, you can leave guile mode without having to know whether the current thread is in guile mode or not.

`int scm_pthread_mutex_lock` (*pthread_mutex_t *mutex*) [C Function]

Like `pthread_mutex_lock`, but leaves guile mode while waiting for the mutex.

`int scm_pthread_cond_wait` (*pthread_cond_t *cond, pthread_mutex_t *mutex*) [C Function]

`int scm_pthread_cond_timedwait` (*pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *abstime*) [C Function]

Like `pthread_cond_wait` and `pthread_cond_timedwait`, but leaves guile mode while waiting for the condition variable.

`int scm_std_select` (*int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout*) [C Function]

Like `select` but leaves guile mode while waiting. Also, the delivery of a system async causes this function to be interrupted with error code `EINTR`.

`unsigned int scm_std_sleep` (*unsigned int seconds*) [C Function]

Like `sleep`, but leaves guile mode while sleeping. Also, the delivery of a system async causes this function to be interrupted.

`unsigned long scm_std_usleep` (*unsigned long usecs*) [C Function]

Like `usleep`, but leaves guile mode while sleeping. Also, the delivery of a system async causes this function to be interrupted.

5.17.7 Critical Sections

`SCM_CRITICAL_SECTION_START` [C Macro]

`SCM_CRITICAL_SECTION_END` [C Macro]

These two macros can be used to delimit a critical section. Syntactically, they are both statements and need to be followed immediately by a semicolon.

Executing `SCM_CRITICAL_SECTION_START` will lock a recursive mutex and block the executing of system asyncs. Executing `SCM_CRITICAL_SECTION_END` will unblock the execution of system asyncs and unlock the mutex. Thus, the code that executes between these two macros can only be executed in one thread at any one time and no system asyncs will run. However, because the mutex is a recursive one, the code might still be reentered by the same thread. You must either allow for this or avoid it, both by careful coding.

On the other hand, critical sections delimited with these macros can be nested since the mutex is recursive.

You must make sure that for each `SCM_CRITICAL_SECTION_START`, the corresponding `SCM_CRITICAL_SECTION_END` is always executed. This means that no non-local exit (such as a signalled error) might happen, for example.

`void scm_dynwind_critical_section` (*SCM mutex*) [C Function]

Call `scm_dynwind_lock_mutex` on *mutex* and call `scm_dynwind_block_asyncs`. When *mutex* is false, a recursive mutex provided by Guile is used instead.

The effect of a call to `scm_dynwind_critical_section` is that the current dynwind context (see Section 5.11.9 [Dynamic Wind], page 266) turns into a critical section. Because of the locked mutex, no second thread can enter it concurrently and because of the blocked asyncs, no system async can reenter it from the current thread.

When the current thread reenters the critical section anyway, the kind of *mutex* determines what happens: When *mutex* is recursive, the reentry is allowed. When it is a normal mutex, an error is signalled.

5.17.8 Fluids and Dynamic States

A *fluid* is an object that can store one value per *dynamic state*. Each thread has a current dynamic state, and when accessing a fluid, this current dynamic state is used to provide the actual value. In this way, fluids can be used for thread local storage, but they are in fact more flexible: dynamic states are objects of their own and can be made current for more than one thread at the same time, or only be made current temporarily, for example.

Fluids can also be used to simulate the desirable effects of dynamically scoped variables. Dynamically scoped variables are useful when you want to set a variable to a value during some dynamic extent in the execution of your program and have them revert to their original value when the control flow is outside of this dynamic extent. See the description of `with-fluids` below for details.

New fluids are created with `make-fluid` and `fluid?` is used for testing whether an object is actually a fluid. The values stored in a fluid can be accessed with `fluid-ref` and `fluid-set!`.

`make-fluid` [Scheme Procedure]
`scm_make_fluid` () [C Function]

Return a newly created fluid. Fluids are objects that can hold one value per dynamic state. That is, modifications to this value are only visible to code that executes with the same dynamic state as the modifying code. When a new dynamic state is constructed, it inherits the values from its parent. Because each thread normally executes with its own dynamic state, you can use fluids for thread local storage.

`fluid?` *obj* [Scheme Procedure]
`scm_fluid_p` (*obj*) [C Function]

Return `#t` iff *obj* is a fluid; otherwise, return `#f`.

`fluid-ref` *fluid* [Scheme Procedure]
`scm_fluid_ref` (*fluid*) [C Function]

Return the value associated with *fluid* in the current dynamic root. If *fluid* has not been set, then return `#f`.

`fluid-set!` *fluid value* [Scheme Procedure]
`scm_fluid_set_x` (*fluid*, *value*) [C Function]

Set the value associated with *fluid* in the current dynamic root.

`with-fluids*` temporarily changes the values of one or more fluids, so that the given procedure and each procedure called by it access the given values. After the procedure returns, the old values are restored.

`with-fluid*` *fluid value thunk* [Scheme Procedure]
`scm_with_fluid` (*fluid*, *value*, *thunk*) [C Function]

Set *fluid* to *value* temporarily, and call *thunk*. *thunk* must be a procedure with no argument.

`with-fluids*` *fluids values thunk* [Scheme Procedure]
`scm_with_fluids` (*fluids*, *values*, *thunk*) [C Function]

Set *fluids* to *values* temporary, and call *thunk*. *fluids* must be a list of fluids and *values* must be the same number of their values to be applied. Each substitution is done in the order given. *thunk* must be a procedure with no argument. it is called inside a `dynamic-wind` and the fluids are set/restored when control enter or leaves the established dynamic extent.

`with-fluids` ((*fluid value*) ...) *body...* [Scheme Macro]

Execute *body...* while each *fluid* is set to the corresponding *value*. Both *fluid* and *value* are evaluated and *fluid* must yield a fluid. *body...* is executed inside a `dynamic-wind` and the fluids are set/restored when control enter or leaves the established dynamic extent.

`SCM scm_c_with_fluids` (*SCM fluids*, *SCM vals*, *SCM* (**cproc*)(*void **), *void *data*) [C Function]
`SCM scm_c_with_fluid` (*SCM fluid*, *SCM val*, *SCM* (**cproc*)(*void **), *void *data*) [C Function]

The function `scm_c_with_fluids` is like `scm_with_fluids` except that it takes a C function to call instead of a Scheme thunk.

The function `scm_c_with_fluid` is similar but only allows one fluid to be set instead of a list.

`void scm_dynwind_fluid` (*SCM fluid*, *SCM val*) [C Function]

This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 5.11.9 [Dynamic Wind], page 266). During the dynwind context, the fluid *fluid* is set to *val*.

More precisely, the value of the fluid is swapped with a 'backup' value whenever the dynwind context is entered or left. The backup value is initialized with the *val* argument.

`make-dynamic-state` [*parent*] [Scheme Procedure]
`scm_make_dynamic_state` (*parent*) [C Function]
: Return a copy of the dynamic state object *parent* or of the current dynamic state when *parent* is omitted.

`dynamic-state?` *obj* [Scheme Procedure]
`scm_dynamic_state_p` (*obj*) [C Function]
: Return `#t` if *obj* is a dynamic state object; return `#f` otherwise.

`int scm_is_dynamic_state` (*SCM obj*) [C Procedure]
: Return non-zero if *obj* is a dynamic state object; return zero otherwise.

`current-dynamic-state` [Scheme Procedure]
`scm_current_dynamic_state` () [C Function]
: Return the current dynamic state object.

`set-current-dynamic-state` *state* [Scheme Procedure]
`scm_set_current_dynamic_state` (*state*) [C Function]
: Set the current dynamic state object to *state* and return the previous current dynamic state object.

`with-dynamic-state` *state proc* [Scheme Procedure]
`scm_with_dynamic_state` (*state, proc*) [C Function]
: Call *proc* while *state* is the current dynamic state object.

`void scm_dynwind_current_dynamic_state` (*SCM state*) [C Procedure]
: Set the current dynamic state to *state* for the current dynwind context.

`void * scm_c_with_dynamic_state` (*SCM state, void *(*func)(void *), void *data*) [C Procedure]
: Like `scm_with_dynamic_state`, but call *func* with *data*.

5.17.9 Parallel forms

The functions described in this section are available from

```
(use-modules (ice-9 threads))
```

`parallel` *expr1 ... exprN* [syntax]
: Evaluate each *expr* expression in parallel, each in its own thread. Return the results as a set of *N* multiple values (see Section 5.11.6 [Multiple Values], page 257).

`letpar` ((*var1 expr1*) ... (*varN exprN*)) *body...* [syntax]
: Evaluate each *expr* in parallel, each in its own thread, then bind the results to the corresponding *var* variables and evaluate *body*.

: `letpar` is like `let` (see Section 5.10.2 [Local Bindings], page 248), but all the expressions for the bindings are evaluated in parallel.

`par-map` *proc lst1 ... lstN* [Scheme Procedure]
`par-for-each` *proc lst1 ... lstN* [Scheme Procedure]
: Call *proc* on the elements of the given lists. `par-map` returns a list comprising the return values from *proc*. `par-for-each` returns an unspecified value, but waits for all calls to complete.

The *proc* calls are `(proc elem1 ... elemN)`, where each *elem* is from the corresponding *lst*. Each *lst* must be the same length. The calls are made in parallel, each in its own thread.

These functions are like `map` and `for-each` (see Section 5.6.2.8 [List Mapping], page 174), but make their *proc* calls in parallel.

`n-par-map` *n proc lst1 . . . lstN* [Scheme Procedure]
`n-par-for-each` *n proc lst1 . . . lstN* [Scheme Procedure]

Call *proc* on the elements of the given lists, in the same way as `par-map` and `par-for-each` above, but use no more than *n* threads at any one time. The order in which calls are initiated within that threads limit is unspecified.

These functions are good for controlling resource consumption if *proc* calls might be costly, or if there are many to be made. On a dual-CPU system for instance $n = 4$ might be enough to keep the CPUs utilized, and not consume too much memory.

`n-for-each-par-map` *n sproc pproc lst1 . . . lstN* [Scheme Procedure]

Apply *pproc* to the elements of the given lists, and apply *sproc* to each result returned by *pproc*. The final return value is unspecified, but all calls will have been completed before returning.

The calls made are `(sproc (pproc elem1 ... elemN))`, where each *elem* is from the corresponding *lst*. Each *lst* must have the same number of elements.

The *pproc* calls are made in parallel, in separate threads. No more than *n* threads are used at any one time. The order in which *pproc* calls are initiated within that limit is unspecified.

The *sproc* calls are made serially, in list element order, one at a time. *pproc* calls on later elements may execute in parallel with the *sproc* calls. Exactly which thread makes each *sproc* call is unspecified.

This function is designed for individual calculations that can be done in parallel, but with results needing to be handled serially, for instance to write them to a file. The *n* limit on threads controls system resource usage when there are many calculations or when they might be costly.

It will be seen that `n-for-each-par-map` is like a combination of `n-par-map` and `for-each`,

```
(for-each sproc (n-par-map n pproc lst1 ... lstN))
```

But the actual implementation is more efficient since each *sproc* call, in turn, can be initiated once the relevant *pproc* call has completed, it doesn't need to wait for all to finish.

5.18 Configuration, Features and Runtime Options

Why is my Guile different from your Guile? There are three kinds of possible variation:

- build differences — different versions of the Guile source code, installation directories, configuration flags that control pieces of functionality being included or left out, etc.
- differences in dynamically loaded code — behaviour and features provided by modules that can be dynamically loaded into a running Guile
- different runtime options — some of the options that are provided for controlling Guile's behaviour may be set differently.

Guile provides "introspective" variables and procedures to query all of these possible variations at runtime. For runtime options, it also provides procedures to change the settings of options and to obtain documentation on what the options mean.

5.18.1 Configuration, Build and Installation

The following procedures and variables provide information about how Guile was configured, built and installed on your system.

`version` [Scheme Procedure]
`effective-version` [Scheme Procedure]
`major-version` [Scheme Procedure]
`minor-version` [Scheme Procedure]
`micro-version` [Scheme Procedure]
`scm_version ()` [C Function]
`scm_effective_version ()` [C Function]
`scm_major_version ()` [C Function]
`scm_minor_version ()` [C Function]
`scm_micro_version ()` [C Function]

Return a string describing Guile's full version number, effective version number, major, minor or micro version number, respectively. The `effective-version` function returns the version name that should remain unchanged during a stable series. Currently that means that it omits the micro version. The effective version should be used for items like the versioned share directory name i.e. '`/usr/share/guile/1.6/`'

```
(version) ⇒ "1.6.0"
(effective-version) ⇒ "1.6"
(major-version) ⇒ "1"
(minor-version) ⇒ "6"
(micro-version) ⇒ "0"
```

`%package-data-dir` [Scheme Procedure]
`scm_sys_package_data_dir ()` [C Function]

Return the name of the directory under which Guile Scheme files in general are stored. On Unix-like systems, this is usually '`/usr/local/share/guile`' or '`/usr/share/guile`'.

`%library-dir` [Scheme Procedure]
`scm_sys_library_dir ()` [C Function]

Return the name of the directory where the Guile Scheme files that belong to the core Guile installation (as opposed to files from a

3rd party package) are installed. On Unix-like systems, this is usually '`/usr/local/share/guile/<GUILE_EFFECTIVE_VERSION>`' or '`/usr/share/guile/<GUILE_EFFECTIVE_VERSION>`', for example: '`/usr/local/share/guile/1.6`'.

`%site-dir` [Scheme Procedure]
`scm_sys_site_dir` () [C Function]

Return the name of the directory where Guile Scheme files specific to your site should be installed. On Unix-like systems, this is usually '`/usr/local/share/guile/site`' or '`/usr/share/guile/site`'.

`%load-path` [Variable]

List of directories which should be searched for Scheme modules and libraries. `%load-path` is initialized when Guile starts up to `(list (%site-dir) (%library-dir) (%package-data-dir) ".")`, prepended with the contents of the GUILE_LOAD_PATH environment variable, if it is set.

`parse-path` *path* [*tail*] [Scheme Procedure]
`scm_parse_path` (*path, tail*) [C Function]

Parse *path*, which is expected to be a colon-separated string, into a list and return the resulting list with *tail* appended. If *path* is `#f`, *tail* is returned.

`search-path` *path filename* [*extensions*] [Scheme Procedure]
`scm_search_path` (*path, filename, extensions*) [C Function]

Search *path* for a directory containing a file named *filename*. The file must be readable, and not a directory. If we find one, return its full filename; otherwise, return `#f`. If *filename* is absolute, return it unchanged. If given, *extensions* is a list of strings; for each directory in *path*, we search for *filename* concatenated with each *extension*.

`%guile-build-info` [Variable]

Alist of information collected during the building of a particular Guile. Entries can be grouped into one of several categories: directories, env vars, and versioning info.

Briefly, here are the keys in `%guile-build-info`, by group:

directories	srcdir, top_srcdir, prefix, exec_prefix, bindir, sbindir, libexecdir, datadir, sysconfdir, sharedstatedir, localstatedir, libdir, infodir, mandir, includedir, pkgdatadir, pkglibdir, pkgincludedir
env vars	LIBS
versioning info	guileversion, libguileinterface, buildstamp

Values are all strings. The value for `LIBS` is typically found also as a part of "guile-config link" output. The value for `guileversion` has form X.Y.Z, and should be the same as returned by `(version)`. The value for `libguileinterface` is libtool compatible and has form CURRENT:REVISION:AGE (see section "Library interface versions" in *GNU Libtool*). The value for `buildstamp` is the output of the date(1) command.

In the source, `%guile-build-info` is initialized from libguile/libpath.h, which is completely generated, so deleting this file before a build guarantees up-to-date values for that build.

5.18.2 Feature Tracking

Guile has a Scheme level variable `*features*` that keeps track to some extent of the features that are available in a running Guile. `*features*` is a list of symbols, for example `threads`, each of which describes a feature of the running Guile process.

`*features*` [Variable]

A list of symbols describing available features of the Guile process.

You shouldn't modify the `*features*` variable directly using `set!`. Instead, see the procedures that are provided for this purpose in the following subsection.

5.18.2.1 Feature Manipulation

To check whether a particular feature is available, use the `provided?` procedure:

`provided?` *feature* [Scheme Procedure]
`feature?` *feature* [Deprecated Scheme Procedure]

Return `#t` if the specified *feature* is available, otherwise `#f`.

To advertise a feature from your own Scheme code, you can use the `provide` procedure:

`provide` *feature* [Scheme Procedure]

Add *feature* to the list of available features in this Guile process.

For C code, the equivalent function takes its feature name as a `char *` argument for convenience:

`void scm_add_feature` (*const char *str*) [C Function]

Add a symbol with name *str* to the list of available features in this Guile process.

5.18.2.2 Common Feature Symbols

In general, a particular feature may be available for one of two reasons. Either because the Guile library was configured and compiled with that feature enabled — i.e. the feature is built into the library on your system. Or because some C or Scheme code that was dynamically loaded by Guile has added that feature to the list.

In the first category, here are the features that the current version of Guile may define (depending on how it is built), and what they mean.

`array` Indicates support for arrays (see Section 5.6.7 [Arrays], page 191).

`array-for-each`
Indicates availability of `array-for-each` and other array mapping procedures (see Section 5.6.7 [Arrays], page 191).

`char-ready?`
Indicates that the `char-ready?` function is available (see Section 5.12.2 [Reading], page 272).

`complex` Indicates support for complex numbers.

`current-time`
Indicates availability of time-related functions: `times`, `get-internal-run-time` and so on (see Section 6.2.5 [Time], page 389).

`debug-extensions`
: Indicates that the debugging evaluator is available, together with the options for controlling it.

`delay`
: Indicates support for promises (see Section 5.13.5 [Delayed Evaluation], page 294).

`EIDs`
: Indicates that the `geteuid` and `getegid` really return effective user and group IDs (see Section 6.2.7 [Processes], page 393).

`inexact`
: Indicates support for inexact numbers.

`i/o-extensions`
: Indicates availability of the following extended I/O procedures: `ftell`, `redirect-port`, `dup->fdes`, `dup2`, `fileno`, `isatty?`, `fdopen`, `primitive-move->fdes` and `fdes->ports` (see Section 6.2.2 [Ports and File Descriptors], page 374).

`net-db`
: Indicates availability of network database functions: `scm_gethost`, `scm_getnet`, `scm_getproto`, `scm_getserv`, `scm_sethost`, `scm_setnet`, `scm_setproto`, `scm_setserv`, and their 'byXXX' variants (see Section 6.2.11.2 [Network Databases], page 404).

`posix`
: Indicates support for POSIX functions: `pipe`, `getgroups`, `kill`, `execl` and so on (see Section 6.2 [POSIX], page 373).

`random`
: Indicates availability of random number generation functions: `random`, `copy-random-state`, `random-uniform` and so on (see Section 5.5.2.15 [Random], page 119).

`reckless`
: Indicates that Guile was built with important checks omitted — you should never see this!

`regex`
: Indicates support for POSIX regular expressions using `make-regexp`, `regexp-exec` and friends (see Section 5.5.6.1 [Regexp Functions], page 146).

`socket`
: Indicates availability of socket-related functions: `socket`, `bind`, `connect` and so on (see Section 6.2.11.4 [Network Sockets and Communication], page 410).

`sort`
: Indicates availability of sorting and merging functions (see Section 5.9.3 [Sorting], page 239).

`system`
: Indicates that the `system` function is available (see Section 6.2.7 [Processes], page 393).

`threads`
: Indicates support for multithreading (see Section 5.17.4 [Threads], page 325).

`values`
: Indicates support for multiple return values using `values` and `call-with-values` (see Section 5.11.6 [Multiple Values], page 257).

Available features in the second category depend, by definition, on what additional code your Guile process has loaded in. The following table lists features that you might encounter for this reason.

`defmacro`
: Indicates that the `defmacro` macro is available (see Section 5.8.6 [Macros], page 231).

`describe`	Indicates that the `(oop goops describe)` module has been loaded, which provides a procedure for describing the contents of GOOPS instances.
`readline`	Indicates that Guile has loaded in Readline support, for command line editing (see Section 6.5 [Readline Support], page 457).
`record`	Indicates support for record definition using `make-record-type` and friends (see Section 5.6.8 [Records], page 204).

Although these tables may seem exhaustive, it is probably unwise in practice to rely on them, as the correspondences between feature symbols and available procedures/behaviour are not strictly defined. If you are writing code that needs to check for the existence of some procedure, it is probably safer to do so directly using the `defined?` procedure than to test for the corresponding feature using `provided?`.

5.18.3 Runtime Options

Guile's runtime behaviour can be modified by setting options. For example, is the language that Guile accepts case sensitive, or should the debugger automatically show a backtrace on error?

Guile has two levels of interface for managing options: a low-level control interface, and a user-level interface which allows the enabling or disabling of options.

Moreover, the options are classified in groups according to whether they configure *reading*, *printing*, *debugging* or *evaluating*.

5.18.3.1 Low Level Options Interfaces

`read-options-interface` [*setting*] [Scheme Procedure]
`eval-options-interface` [*setting*] [Scheme Procedure]
`print-options-interface` [*setting*] [Scheme Procedure]
`debug-options-interface` [*setting*] [Scheme Procedure]
`evaluator-traps-interface` [*setting*] [Scheme Procedure]
`scm_read_options` (*setting*) [C Function]
`scm_eval_options_interface` (*setting*) [C Function]
`scm_print_options` (*setting*) [C Function]
`scm_debug_options` (*setting*) [C Function]
`scm_evaluator_traps` (*setting*) [C Function]

> If one of these procedures is called with no arguments (or with `setting == SCM_UNDEFINED` in C code), it returns a list describing the current setting of the read, eval, print, debug or evaluator traps options respectively. The setting of a boolean option is indicated simply by the presence or absence of the option symbol in the list. The setting of a non-boolean option is indicated by the presence of the option symbol immediately followed by the option's current value.
>
> If called with a list argument, these procedures interpret the list as an option setting and modify the relevant options accordingly. [FIXME — this glosses over a lot of details!]
>
> If called with any other argument, such as `'help`, these procedures return a list of entries like `(OPTION-SYMBOL DEFAULT-VALUE DOC-STRING)`, with each entry giving the default value and documentation for each option symbol in the relevant set of options.

5.18.3.2 User Level Options Interfaces

`<group>-options` [*arg*] [Scheme Procedure]
`read-options` [*arg*] [Scheme Procedure]
`print-options` [*arg*] [Scheme Procedure]
`debug-options` [*arg*] [Scheme Procedure]
`traps` [*arg*] [Scheme Procedure]

These functions list the options in their group. The optional argument *arg* is a symbol which modifies the form in which the options are presented.

With no arguments, `<group>-options` returns the values of the options in that particular group. If *arg* is `'help`, a description of each option is given. If *arg* is `'full`, programmers' options are also shown.

arg can also be a list representing the state of all options. In this case, the list contains single symbols (for enabled boolean options) and symbols followed by values.

[FIXME: I don't think 'full is ever any different from 'help. What's up?]

`<group>-enable` *option-symbol* [Scheme Procedure]
`read-enable` *option-symbol* [Scheme Procedure]
`print-enable` *option-symbol* [Scheme Procedure]
`debug-enable` *option-symbol* [Scheme Procedure]
`trap-enable` *option-symbol* [Scheme Procedure]

These functions set the specified *option-symbol* in their options group. They only work if the option is boolean, and throw an error otherwise.

`<group>-disable` *option-symbol* [Scheme Procedure]
`read-disable` *option-symbol* [Scheme Procedure]
`print-disable` *option-symbol* [Scheme Procedure]
`debug-disable` *option-symbol* [Scheme Procedure]
`trap-disable` *option-symbol* [Scheme Procedure]

These functions turn off the specified *option-symbol* in their options group. They only work if the option is boolean, and throw an error otherwise.

`<group>-set!` *option-symbol value* [syntax]
`read-set!` *option-symbol value* [syntax]
`print-set!` *option-symbol value* [syntax]
`debug-set!` *option-symbol value* [syntax]
`trap-set!` *option-symbol value* [syntax]

These functions set a non-boolean *option-symbol* to the specified *value*.

5.18.3.3 Reader options

Here is the list of reader options generated by typing `(read-options 'full)` in Guile. You can also see the default values.

```
keywords         #f      Style of keyword recognition: #f or 'prefix
case-insensitive no      Convert symbols to lower case.
positions        yes     Record positions of source code expressions.
copy             no      Copy source code expressions.
```

Notice that while Standard Scheme is case insensitive, to ease translation of other Lisp dialects, notably Emacs Lisp, into Guile, Guile is case-sensitive by default.

To make Guile case insensitive, you can type

```
(read-enable 'case-insensitive)
```

5.18.3.4 Printing options

Here is the list of print options generated by typing `(print-options 'full)` in Guile. You can also see the default values.

```
quote-keywordish-symbols reader How to print symbols that have a colon
                                as their first or last character. The
                                value '#f' does not quote the colons;
                                '#t' quotes them; 'reader' quotes
                                them when the reader option
                                'keywords' is not '#f'.

highlight-prefix         {      The string to print before highlighted values.
highlight-suffix         }      The string to print after highlighted values.

source                   no     Print closures with source.
closure-hook             #f     Hook for printing closures.
```

5.18.3.5 Evaluator options

These are the evaluator options with their default values, as they are printed by typing `(eval-options 'full)` in Guile.

```
stack           22000   Size of thread stacks (in machine words).
```

5.18.3.6 Evaluator trap options

[FIXME: These flags, together with their corresponding handlers, are not user level options. Probably this entire section should be moved to the documentation about the low-level programmer debugging interface.]

Here is the list of evaluator trap options generated by typing `(traps 'full)` in Guile. You can also see the default values.

```
exit-frame      no      Trap when exiting eval or apply.
apply-frame     no      Trap when entering apply.
enter-frame     no      Trap when eval enters new frame.
traps yes Enable evaluator traps.
```

`key` *cont tailp* [apply-frame-handler]

Called when a procedure is being applied.

Called if:

- evaluator traps are enabled [traps interface], and
- either
  - `apply-frame` is enabled [traps interface], or
  - trace mode is on [debug-options interface], and the procedure being called has the trace property enabled.

cont is a “debug object”, which means that it can be passed to `make-stack` to discover the stack at the point of the trap. The apply frame handler’s code can capture a restartable continuation if it wants to by using `call-with-current-continuation` in the usual way.

tailp is true if this is a tail call

`key` *cont retval* [exit-frame-handler]

Called when a value is returned from a procedure.

Called if:

- evaluator traps are enabled [traps interface], and
- either
  - `exit-frame` is enabled [traps interface], or
  - trace mode is on [debug-options interface], and the procedure being called has the trace property enabled.

cont is a "debug object", which means that it can be passed to `make-stack` to discover the stack at the point of the trap. The exit frame handler's code can capture a restartable continuation if it wants to by using `call-with-current-continuation` in the usual way.

retval is the return value.

5.18.3.7 Debugger options

Here is the list of print options generated by typing `(debug-options 'full)` in Guile. You can also see the default values.

```
stack           20000   Stack size limit (0 = no check).
debug           yes     Use the debugging evaluator.
backtrace       no      Show backtrace on error.
depth           20      Maximal length of printed backtrace.
maxdepth        1000    Maximal number of stored backtrace frames.
frames          3       Maximum number of tail-recursive frames in backtrace.
indent          10      Maximal indentation in backtrace.
backwards       no      Display backtrace in anti-chronological order.
procnames       yes     Record procedure names at definition.
trace           no      *Trace mode.
breakpoints     no      *Check for breakpoints.
cheap           yes     *This option is now obsolete.  Setting it has no effect.
```

Stack overflow

Stack overflow errors are caused by a computation trying to use more stack space than has been enabled by the `stack` option. They are reported like this:

```
(non-tail-recursive-factorial 500)
⊣
ERROR: Stack overflow
ABORT: (stack-overflow)
```

If you get an error like this, you can either try rewriting your code to use less stack space, or increase the maximum stack size. To increase the maximum stack size, use `debug-set!`, for example:

```
(debug-set! stack 200000)
⇒
(show-file-name #t stack 200000 debug backtrace depth 20 maxdepth 1000 frames 3 indent

(non-tail-recursive-factorial 500)
⇒
```

```
122013682599111006870123878542304692625357434...
```

If you prefer to try rewriting your code, you may be able to save stack space by making some of your procedures *tail recursive* (see Section 3.1.3.2 [Tail Calls], page 22).

5.18.3.8 Examples of option use

Here is an example of a session in which some read and debug option handling procedures are used. In this example, the user

1. Notices that the symbols `abc` and `aBc` are not the same
2. Examines the `read-options`, and sees that `case-insensitive` is set to “no”.
3. Enables `case-insensitive`
4. Verifies that now `aBc` and `abc` are the same
5. Disables `case-insensitive` and enables debugging `backtrace`
6. Reproduces the error of displaying `aBc` with backtracing enabled [FIXME: this last example is lame because there is no depth in the backtrace. Need to give a better example, possibly putting debugging option examples in a separate session.]

```
guile> (define abc "hello")
guile> abc
"hello"
guile> aBc
ERROR: In expression aBc:
ERROR: Unbound variable: aBc
ABORT: (misc-error)

Type "(backtrace)" to get more information.
guile> (read-options 'help)
keywords #f Style of keyword recognition: #f or 'prefix
case-insensitive no Convert symbols to lower case.
positions yes Record positions of source code expressions.
copy no Copy source code expressions.
guile> (debug-options 'help)
stack 20000 Stack size limit (0 = no check).
debug yes Use the debugging evaluator.
backtrace no Show backtrace on error.
depth 20 Maximal length of printed backtrace.
maxdepth 1000 Maximal number of stored backtrace frames.
frames 3 Maximum number of tail-recursive frames in backtrace.
indent 10 Maximal indentation in backtrace.
backwards no Display backtrace in anti-chronological order.
procnames yes Record procedure names at definition.
trace no *Trace mode.
breakpoints no *Check for breakpoints.
cheap yes *This option is now obsolete.  Setting it has no effect.
guile> (read-enable 'case-insensitive)
(keywords #f case-insensitive positions)
guile> aBc
"hello"
guile> (read-disable 'case-insensitive)
(keywords #f positions)
guile> (debug-enable 'backtrace)
(stack 20000 debug backtrace depth 20 maxdepth 1000 frames 3 indent 10 procnames cheap)
guile> aBc

Backtrace:
```

```
0* aBc

ERROR: In expression aBc:
ERROR: Unbound variable: aBc
ABORT: (misc-error)
guile>
```

5.19 Support for Translating Other Languages

[Describe translation framework.]

5.19.1 Emacs Lisp Support

`nil-car` *x* [Scheme Procedure]
`scm_nil_car` (*x*) [C Function]

Return the car of *x*, but convert it to LISP nil if it is Scheme's end-of-list.

`nil-cdr` *x* [Scheme Procedure]
`scm_nil_cdr` (*x*) [C Function]

Return the cdr of *x*, but convert it to LISP nil if it is Scheme's end-of-list.

`nil-cons` *x y* [Scheme Procedure]
`scm_nil_cons` (*x*, *y*) [C Function]

Create a new cons cell with *x* as the car and *y* as the cdr, but convert *y* to Scheme's end-of-list if it is a Lisp nil.

`nil-eq` *x y* [Scheme Procedure]

Compare *x* and *y* and return Lisp's t if they are `eq?`, return Lisp's nil otherwise.

`null` *x* [Scheme Procedure]
`scm_null` (*x*) [C Function]

Return Lisp's `t` if *x* is nil in the LISP sense, return Lisp's nil otherwise.

5.20 Support for Internationalization

Guile provides an interface to GNU `gettext` for translating message strings (see section "Introduction" in *GNU `gettext` utilities*).

Messages are collected in domains, so different libraries and programs maintain different message catalogues. The *domain* parameter in the functions below is a string (it becomes part of the message catalog filename).

When `gettext` is not available, or if Guile was configured '`--without-nls`', dummy functions doing no translation are provided.

`gettext` *msg* [*domain* [*category*]] [Scheme Procedure]
`scm_gettext` (*msg, domain, category*) [C Function]

Return the translation of *msg* in *domain*. *domain* is optional and defaults to the domain set through `textdomain` below. *category* is optional and defaults to `LC_MESSAGES` (see Section 6.2.13 [Locales], page 417).

Normal usage is for *msg* to be a literal string. `xgettext` can extract those from the source to form a message catalogue ready for translators (see section "Invoking the `xgettext` Program" in *GNU `gettext` utilities*).

```
(display (gettext "You are in a maze of twisty passages."))
```

`_` is a commonly used shorthand, an application can make that an alias for `gettext`. Or a library can make a definition that uses its specific *domain* (so an application can change the default without affecting the library).

```
(define (_ msg) (gettext msg "mylibrary"))
(display (_ "File not found."))
```

`_` is also a good place to perhaps strip disambiguating extra text from the message string, as for instance in section "How to use `gettext` in GUI programs" in *GNU `gettext` utilities*.

`ngettext` *msg msgplural n* [*domain* [*category*]] [Scheme Procedure]
`scm_ngettext` (*msg, msgplural, n, domain, category*) [C Function]

Return the translation of *msg*/*msgplural* in *domain*, with a plural form chosen appropriately for the number *n*. *domain* is optional and defaults to the domain set through `textdomain` below. *category* is optional and defaults to `LC_MESSAGES` (see Section 6.2.13 [Locales], page 417).

msg is the singular form, and *msgplural* the plural. When no translation is available, *msg* is used if $n = 1$, or *msgplural* otherwise. When translated, the message catalogue can have a different rule, and can have more than two possible forms.

As per `gettext` above, normal usage is for *msg* and *msgplural* to be literal strings, since `xgettext` can extract them from the source to build a message catalogue. For example,

```
(define (done n)
  (format #t (ngettext "~a file processed\n"
                       "~a files processed\n" n)
          n))

(done 1) ⊣ 1 file processed
```

```
(done 3) ⊣ 3 files processed
```

It's important to use `ngettext` rather than plain `gettext` for plurals, since the rules for singular and plural forms in English are not the same in other languages. Only `ngettext` will allow translators to give correct forms (see section "Additional functions for plural forms" in *GNU* `gettext` *utilities*).

`textdomain` [*domain*] [Scheme Procedure]
`scm_textdomain` (*domain*) [C Function]

Get or set the default gettext domain. When called with no parameter the current domain is returned. When called with a parameter, *domain* is set as the current domain, and that new value returned. For example,

```
(textdomain "myprog")
⇒ "myprog"
```

`bindtextdomain` *domain* [*directory*] [Scheme Procedure]
`scm_bindtextdomain` (*domain, directory*) [C Function]

Get or set the directory under which to find message files for *domain*. When called without a *directory* the current setting is returned. When called with a *directory*, *directory* is set for *domain* and that new setting returned. For example,

```
(bindtextdomain "myprog" "/my/tree/share/locale")
⇒ "/my/tree/share/locale"
```

When using Autoconf/Automake, an application should arrange for the configured `localedir` to get into the program (by substituting, or by generating a config file) and set that for its domain. This ensures the catalogue can be found even when installed in a non-standard location.

`bind-textdomain-codeset` *domain* [*encoding*] [Scheme Procedure]
`scm_bind_textdomain_codeset` (*domain, encoding*) [C Function]

Get or set the text encoding to be used by `gettext` for messages from *domain*. *encoding* is a string, the name of a coding system, for instance `"8859_1"`. (On a Unix/POSIX system the `iconv` program can list all available encodings.)

When called without an *encoding* the current setting is returned, or `#f` if none yet set. When called with an *encoding*, it is set for *domain* and that new setting returned. For example,

```
(bind-textdomain-codeset "myprog")
⇒ #f
(bind-textdomain-codeset "myprog" "latin-9")
⇒ "latin-9"
```

The encoding requested can be different from the translated data file, messages will be recoded as necessary. But note that when there is no translation, `gettext` returns its *msg* unchanged, ie. without any recoding. For that reason source message strings are best as plain ASCII.

Currently Guile has no understanding of multi-byte characters, and string functions won't recognise character boundaries in multi-byte strings. An application will at least be able to pass such strings through to some output though. Perhaps this will change in the future.

5.21 Debugging Infrastructure

5.21.1 Interactive Debugging

`backtrace` [*highlights*] [Scheme Procedure]
`scm_backtrace_with_highlights` (*highlights*) [C Function]
`scm_backtrace` () [C Function]

Display a backtrace of the stack saved by the last error to the current output port. When *highlights* is given, it should be a list and all members of it are highligthed in the backtrace.

`debug` [Scheme Procedure]

Invoke the Guile debugger to explore the context of the last error.

5.21.2 Breakpoints

`set-breakpoint!` *behaviour . location-args* [Generic Function]

Set a breakpoint with behaviour *behaviour* at the location specified by *location-args*.

The form of the *location-args* depends upon what methods for `set-breakpoint!` have been provided by the implementations of subclasses of the `<breakpoint>` base class.

`get-breakpoint` *. location-args* [Generic Function]

Find and return the breakpoint instance at the location specified by *location-args*.

The form of the *location-args* depends upon what methods for `get-breakpoint` have been provided by the implementations of subclasses of the `<breakpoint>` base class.

`set-breakpoint!` *behaviour* (*proc* <*procedure*>) [Method]

Set a breakpoint with behaviour *behaviour* before applications of the procedure *proc*.

`set-breakpoint!` *behaviour x-as-read* (*x-pairified* <*pair*>) [Method]

Set a breakpoint with behaviour *behaviour* on the source expression *x-pairified*, storing *x-as-read* for use in messages describing the breakpoint.

`set-breakpoint!` *behaviour* (*number* <*integer*>) [Method]

Change the behaviour of existing breakpoint number *number* to *behaviour*.

`bp-behaviour` *breakpoint* [Accessor]

Get or set the behaviour of the breakpoint instance *breakpoint*.

`bp-enabled?` *breakpoint* [Accessor]

Get or set the enabled state of the specified *breakpoint*.

`enable-breakpoint!` *. location-args* [Procedure]
`disable-breakpoint!` *. location-args* [Procedure]

Enable or disable the breakpoint at the location specified by *location-args*.

`bp-delete!` *breakpoint* [Generic Function]

Delete breakpoint *breakpoint*. This means (1) doing whatever is needed to prevent the breakpoint from triggering again, and (2) removing it from the global list of current breakpoints.

`delete-breakpoint!` . *location-args* [Procedure]

Delete the breakpoint at the location specified by *location-args*.

`bp-describe` *breakpoint port* [Generic Function]

Print a description of *breakpoint* to the specified *port*. *port* can be `#t` for standard output, or else any output port.

`describe-breakpoint` . *location-args* [Procedure]

Print (to standard output) a description of the breakpoint at location specified by *location-args*.

`all-breakpoints` [Procedure]

Return a list of all current breakpoints, ordered by breakpoint number.

`describe-all-breakpoints` [Procedure]

Print a description of all current breakpoints to standard output.

5.21.3 Source Properties

As Guile reads in Scheme code from file or from standard input, it remembers the file name, line number and column number where each expression begins. These pieces of information are known as the *source properties* of the expression. If an expression undergoes transformation — for example, if there is a syntax transformer in effect, or the expression is a macro call — the source properties are copied from the untransformed to the transformed expression so that, if an error occurs when evaluating the transformed expression, Guile's debugger can point back to the file and location where the expression originated.

The way that source properties are stored means that Guile can only associate source properties with parenthesized expressions, and not, for example, with individual symbols, numbers or strings. The difference can be seen by typing `(xxx)` and `xxx` at the Guile prompt (where the variable `xxx` has not been defined):

```
guile> (xxx)
standard input:2:1: In expression (xxx):
standard input:2:1: Unbound variable: xxx
ABORT: (unbound-variable)
guile> xxx
<unnamed port>: In expression xxx:
<unnamed port>: Unbound variable: xxx
ABORT: (unbound-variable)
```

In the latter case, no source properties were stored, so the best that Guile could say regarding the location of the problem was "`<unnamed port>`".

The recording of source properties is controlled by the read option named "positions" (see Section 5.18.3.3 [Reader options], page 339). This option is switched *on* by default, together with the debug options "debug" and "backtrace" (see Section 5.18.3.7 [Debugger options], page 341), when Guile is run interactively; all these options are *off* by default when Guile runs a script non-interactively.

The following procedures can be used to access and set the source properties of read expressions.

`set-source-properties!` *obj plist* [Scheme Procedure]
`scm_set_source_properties_x` (*obj, plist*) [C Function]

Install the association list *plist* as the source property list for *obj*.

`set-source-property!` *obj key datum* [Scheme Procedure]
`scm_set_source_property_x` (*obj, key, datum*) [C Function]

Set the source property of object *obj*, which is specified by *key* to *datum*. Normally, the key will be a symbol.

`source-properties` *obj* [Scheme Procedure]
`scm_source_properties` (*obj*) [C Function]

Return the source property association list of *obj*.

`source-property` *obj key* [Scheme Procedure]
`scm_source_property` (*obj, key*) [C Function]

Return the source property specified by *key* from *obj*'s source property list.

In practice there are only two ways that you should use the ability to set an expression's source breakpoints.

- To set a breakpoint on an expression, use `(set-source-property! *expr* 'breakpoint #t)`. If you do this, you should also set the `traps` and `enter-frame-handler` trap options (see Section 5.18.3.6 [Evaluator trap options], page 340) and `breakpoints` debug option (see Section 5.18.3.7 [Debugger options], page 341) appropriately, and the evaluator will then call your enter frame handler whenever it is about to evaluate that expression.
- To make a read or constructed expression appear to have come from a different source than what the expression's source properties already say, you can use `set-source-property!` to set the expression's `filename`, `line` and `column` properties. The properties that you set will then show up later if that expression is involved in a backtrace or error report.

If you are looking for a way to attach arbitrary information to an expression other than these properties, you should use `make-object-property` instead (see Section 5.9.2 [Object Properties], page 238), because that will avoid bloating the source property hash table, which is really only intended for the specific purposes described in this section.

5.21.4 Using Traps

`with-traps` *thunk* [Scheme Procedure]
`scm_with_traps` (*thunk*) [C Function]

Call *thunk* with traps enabled.

`debug-object?` *obj* [Scheme Procedure]
`scm_debug_object_p` (*obj*) [C Function]

Return `#t` if *obj* is a debug object.

5.21.5 Capturing the Stack or Innermost Stack Frame

When an error occurs in a running program, or the program hits a breakpoint, its state at that point can be represented by a *stack* of all the evaluations and procedure applications that are logically in progress at that time, each of which is known as a *frame*. The programmer can learn more about the program's state at the point of interruption or error by inspecting the stack and its frames.

`make-stack` *obj . args* [Scheme Procedure]
`scm_make_stack` (*obj, args*) [C Function]

Create a new stack. If *obj* is `#t`, the current evaluation stack is used for creating the stack frames, otherwise the frames are taken from *obj* (which must be either a debug object or a continuation).

args should be a list containing any combination of integer, procedure and `#t` values.

These values specify various ways of cutting away uninteresting stack frames from the top and bottom of the stack that `make-stack` returns. They come in pairs like this: `(`***`inner_cut_1 outer_cut_1 inner_cut_2 outer_cut_2`*** `...)`.

Each *inner_cut_N* can be `#t`, an integer, or a procedure. `#t` means to cut away all frames up to but excluding the first user module frame. An integer means to cut away exactly that number of frames. A procedure means to cut away all frames up to but excluding the application frame whose procedure matches the specified one.

Each *outer_cut_N* can be an integer or a procedure. An integer means to cut away that number of frames. A procedure means to cut away frames down to but excluding the application frame whose procedure matches the specified one.

If the *outer_cut_N* of the last pair is missing, it is taken as 0.

`last-stack-frame` *obj* [Scheme Procedure]
`scm_last_stack_frame` (*obj*) [C Function]

Return a stack which consists of a single frame, which is the last stack frame for *obj*. *obj* must be either a debug object or a continuation.

5.21.6 Examining the Stack

`stack?` *obj* [Scheme Procedure]
`scm_stack_p` (*obj*) [C Function]

Return `#t` if *obj* is a calling stack.

`stack-id` *stack* [Scheme Procedure]
`scm_stack_id` (*stack*) [C Function]

Return the identifier given to *stack* by `start-stack`.

`stack-length` *stack* [Scheme Procedure]
`scm_stack_length` (*stack*) [C Function]

Return the length of *stack*.

`stack-ref` *stack index* [Scheme Procedure]
`scm_stack_ref` (*stack, index*) [C Function]

Return the *index*'th frame from *stack*.

`display-backtrace` *stack port* [*first* [*depth* [*highlights*]]] [Scheme Procedure]
`scm_display_backtrace_with_highlights` (*stack, port, first, depth, highlights*) [C Function]

`scm_display_backtrace` (*stack, port, first, depth*) [C Function]

Display a backtrace to the output port *port*. *stack* is the stack to take the backtrace from, *first* specifies where in the stack to start and *depth* how much frames to display. Both *first* and *depth* can be `#f`, which means that default values will be used. When *highlights* is given, it should be a list and all members of it are highligthed in the backtrace.

5.21.7 Examining Stack Frames

`frame?` *obj* [Scheme Procedure]
`scm_frame_p` (*obj*) [C Function]

Return `#t` if *obj* is a stack frame.

`frame-number` *frame* [Scheme Procedure]
`scm_frame_number` (*frame*) [C Function]

Return the frame number of *frame*.

`frame-previous` *frame* [Scheme Procedure]
`scm_frame_previous` (*frame*) [C Function]

Return the previous frame of *frame*, or `#f` if *frame* is the first frame in its stack.

`frame-next` *frame* [Scheme Procedure]
`scm_frame_next` (*frame*) [C Function]

Return the next frame of *frame*, or `#f` if *frame* is the last frame in its stack.

`frame-source` *frame* [Scheme Procedure]
`scm_frame_source` (*frame*) [C Function]

Return the source of *frame*.

`frame-procedure?` *frame* [Scheme Procedure]
`scm_frame_procedure_p` (*frame*) [C Function]

Return `#t` if a procedure is associated with *frame*.

`frame-procedure` *frame* [Scheme Procedure]
`scm_frame_procedure` (*frame*) [C Function]

Return the procedure for *frame*, or `#f` if no procedure is associated with *frame*.

`frame-arguments` *frame* [Scheme Procedure]
`scm_frame_arguments` (*frame*) [C Function]

Return the arguments of *frame*.

`frame-evaluating-args?` *frame* [Scheme Procedure]
`scm_frame_evaluating_args_p` (*frame*) [C Function]

Return `#t` if *frame* contains evaluated arguments.

`frame-overflow?` *frame* [Scheme Procedure]
`scm_frame_overflow_p` (*frame*) [C Function]

Return `#t` if *frame* is an overflow frame.

`frame-real?` *frame* [Scheme Procedure]
`scm_frame_real_p` (*frame*) [C Function]

Return `#t` if *frame* is a real frame.

`display-application` *frame* [*port* [*indent*]] [Scheme Procedure]
`scm_display_application` (*frame*, *port*, *indent*) [C Function]

Display a procedure application *frame* to the output port *port*. *indent* specifies the indentation of the output.

5.21.8 Decoding Memoized Source Expressions

`memoized?` *obj* [Scheme Procedure]
`scm_memoized_p` (*obj*) [C Function]

Return `#t` if *obj* is memoized.

`unmemoize` *m* [Scheme Procedure]
`scm_unmemoize` (*m*) [C Function]

Unmemoize the memoized expression *m*,

`memoized-environment` *m* [Scheme Procedure]
`scm_memoized_environment` (*m*) [C Function]

Return the environment of the memoized expression *m*.

5.21.9 Starting a New Stack

`start-stack` *id exp* [Scheme Syntax]

Evaluate *exp* on a new calling stack with identity *id*. If *exp* is interrupted during evaluation, backtraces will not display frames farther back than *exp*'s top-level form. This macro is a way of artificially limiting backtraces and stack procedures, largely as a convenience to the user.

5.22 GH: A Portable C to Scheme Interface

This chapter shows how to use the GH interface to call Guile from your application's C code, and to add new Scheme level procedures to Guile whose behaviour is specified by application specific code written in C.

Note, however, that the GH interface is now deprecated, and developers are encouraged to switch to using the scm interface instead. Therefore, for each GH feature, this chapter also documents how to achieve the same result using the scm interface.

5.22.1 Why the GH Interface is Now Deprecated

Historically, the GH interface was the product of a practical problem and a neat idea. The practical problem was that the interface of the `scm_` functions with which Guile itself was written (inherited from Aubrey Jaffer's SCM) was so closely tied to the (rather arcane) details of the internal data representation that it was extremely difficult to write a Guile extension using these functions. The neat idea was to define a high level language extension interface in such a way that other extension language projects, not just Guile, would be able to provide an implementation of that interface; then applications using this interface could be compiled with whichever of the various available implementations they chose. So the GH interface was created, and advertised both as the recommended interface for application developers wishing to use Guile, and as a portable high level interface that could theoretically be implemented by other extension language projects.

Time passed, and various things changed. Crucially, an enormous number of improvements were made to the `scm_` interface that Guile itself uses in its implementation, with the result that it is now both easy and comfortable to write a Guile extension with this interface. At the same time, the contents of the GH interface were somewhat neglected by the core Guile developers, such that some key operations — such as smob creation and management — are simply not possible using GH alone. Finally, the idea of multiple implementations of the GH interface did not really crystallize (apart, I believe, from a short lived implementation by the MzScheme project).

For all these reasons, the Guile developers have decided to deprecate the GH interface — which means that support for GH will be completely removed after the next few releases — and to focus only on the `scm_` interface, with additions to ensure that it is as easy to use in all respects as GH was.

It remains an open question whether a deep kind of interface portability would be useful for extension language-based applications, and it may still be an interesting project to attempt to define a corresponding GH-like interface, but the Guile developers no longer plan to try to do this as part of the core Guile project.

5.22.2 Transitioning away from GH

The following table summarizes how to transition from the GH to the scm interface. The replacements that are recommended are not always completely equivalent to the GH functionality that they should replace. Therefore, you should read the reference documentation of the replacements carefully if you are not yet familiar with them.

Header file

Use `#include <libguile.h>` instead of `#include <guile/gh.h>`.

Compiling and Linking

Use `guile-config` to pick up the flags required to compile C or C++ code that uses `libguile`, like so

```
$(CC) -o prog.o -c prog.c `guile-config compile`
```

If you are using libtool to link your executables, just use `-lguile` in your link command. Libtool will expand this into the needed linker options automatically. If you are not using libtool, use the `guile-config` program to query the needed options explicitly. A linker command like

```
$(CC) -o prog prog.o `guile-config link`
```

should be all that is needed. To link shared libraries that will be used as Guile Extensions, use libtool to control both the compilation and the link stage.

The `SCM` type

No change: the scm interface also uses this type to represent an arbitrary Scheme value.

`SCM_BOOL_F` and `SCM_BOOL_T`

No change.

`SCM_UNSPECIFIED` and `SCM_UNDEFINED`

No change.

`gh_enter` Use `scm_boot_guile` instead, but note that `scm_boot_guile` has a slightly different calling convention from `gh_enter`: `scm_boot_guile`, and the main program function that you specify for `scm_boot_guile` to call, both take an additional *closure* parameter. Section 4.1.1 [Guile Initialization Functions], page 55 for more details.

`gh_repl` Use `scm_shell` instead.

`gh_init` Use `scm_init_guile` instead.

`gh_catch` Use `scm_internal_catch` instead.

`gh_eval_str`

Use `scm_c_eval_string` instead.

`gh_eval_str_with_catch`

Use `scm_c_eval_string` together with `scm_internal_catch` instead.

`gh_eval_str_with_standard_handler`

Use `scm_c_eval_string` together with `scm_internal_catch` and `scm_handle_by_message_no_exit` instead.

`gh_eval_str_with_stack_saving_handler`

Use `scm_c_eval_string` together with `scm_internal_stack_catch` and `scm_handle_by_message_no_exit` instead.

`gh_eval_file` or `gh_load`

Use `scm_c_primitive_load` instead.

`gh_eval_file_with_catch`

Use `scm_c_primitive_load` together with `scm_internal_catch` instead.

`gh_eval_file_with_standard_handler`
: Use `scm_c_primitive_load` together with `scm_internal_catch` and `scm_handle_by_message_no_exit` instead.

`gh_new_procedure`
`gh_new_procedure0_0`
`gh_new_procedure0_1`
`gh_new_procedure0_2`
`gh_new_procedure1_0`
`gh_new_procedure1_1`
`gh_new_procedure1_2`
`gh_new_procedure2_0`
`gh_new_procedure2_1`
`gh_new_procedure2_2`
`gh_new_procedure3_0`
`gh_new_procedure4_0`
`gh_new_procedure5_0`
: Use `scm_c_define_gsubr` instead, but note that the arguments are in a different order: for `scm_c_define_gsubr` the C function pointer is the last argument. Section 4.2.1 [A Sample Guile Extension], page 58 for an example.

`gh_defer_ints` and `gh_allow_ints`
: Use `SCM_CRITICAL_SECTION_START` and `SCM_CRITICAL_SECTION_END` instead. Note that these macros are used without parentheses, as in `SCM_DEFER_INTS;`.

`gh_bool2scm`
: Use `scm_from_bool` instead.

`gh_int2scm`
: Use `scm_from_int` instead.

`gh_ulong2scm`
: Use `scm_from_ulong` instead.

`gh_long2scm`
: Use `scm_from_long` instead.

`gh_double2scm`
: Use `scm_make_real` instead.

`gh_char2scm`
: Use `SCM_MAKE_CHAR` instead.

`gh_str2scm`
: Use `scm_from_locale_stringn` instead.

`gh_str02scm`
: Use `scm_from_locale_string` instead.

`gh_set_substr`
: Use `scm_string_copy_x`.

`gh_symbol2scm`
: Use `scm_from_locale_symbol` instead.

`gh_ints2scm`
`gh_doubles2scm`
`gh_chars2byvect`
`gh_shorts2svect`
`gh_longs2ivect`
`gh_ulongs2uvect`
`gh_floats2fvect`
`gh_doubles2dvect`
: Use the uniform numeric vector function, See Section 5.6.4 [Uniform Numeric Vectors], page 178.

`gh_scm2bool`
: Use `scm_is_true` or `scm_to_bool` instead.

`gh_scm2int`
: Use `scm_to_int` instead.

`gh_scm2ulong`
: Use `scm_to_ulong` instead.

`gh_scm2long`
: Use `scm_to_long` instead.

`gh_scm2double`
: Use `scm_to_double` instead.

`gh_scm2char`
: Use `scm_to_char` instead.

`gh_scm2newstr`
: Use `scm_to_locale_string` or similar instead.

`gh_get_substr`
: Use `scm_c_substring` together with `scm_to_locale_string` or similar instead.

`gh_symbol2newstr`
: Use `scm_symbol_to_string` together with `scm_to_locale_string` or similar instead.

`gh_scm2chars`
: Use `scm_from_locale_string` (or similar) or the uniform numeric vector functions (see Section 5.6.4 [Uniform Numeric Vectors], page 178) instead.

`gh_scm2shorts`
`gh_scm2longs`
`gh_scm2floats`
`gh_scm2doubles`
: Use the uniform numeric vector function, See Section 5.6.4 [Uniform Numeric Vectors], page 178.

`gh_boolean_p`
: Use `scm_is_bool` instead.

`gh_symbol_p`
: Use `scm_is_symbol` instead.

`gh_char_p`
: Replace `gh_char_p (`*obj*`)` with

```
scm_is_true (scm_char_p (obj))
```

`gh_vector_p`
: Replace `gh_vector_p (`*obj*`)` with

```
scm_is_true (scm_vector_p (obj))
```

`gh_pair_p`
: Replace `gh_pair_p (`*obj*`)` with

```
scm_is_true (scm_pair_p (obj))
```

`gh_number_p`
: Use `scm_is_number` instead.

`gh_string_p`
: Use `scm_is_string` instead.

`gh_procedure_p`
: Replace `gh_procedure_p (`*obj*`)` by

```
scm_is_true (scm_procedure_p (obj))
```

`gh_list_p`
: Replace `gh_list_p (`*obj*`)` with

```
scm_is_true (scm_list_p (obj))
```

`gh_inexact_p`
: Replace `gh_inexact_p (`*obj*`)` with

```
scm_is_true (scm_inexact_p (obj))
```

`gh_exact_p`
: Replace `gh_exact_p (`*obj*`)` with

```
scm_is_true (scm_exact_p (obj))
```

`gh_eq_p` Use `scm_is_eq` instead.

`gh_eqv_p` Replace `gh_eqv_p (`*x*`, `*y*`)` with

```
scm_is_true (scm_eqv_p (x, y))
```

`gh_equal_p`
: Replace `gh_equal_p (`*x*`, `*y*`)` with

```
scm_is_true (scm_equal_p (x, y))
```

`gh_string_equal_p`
: Replace `gh_string_equal_p (`*x*`, `*y*`)` with

```
scm_is_true (scm_string_equal_p (x, y))
```

`gh_null_p`
: Use `scm_is_null` instead.

`gh_not` Use `scm_not` instead.

`gh_make_string`
: Use `scm_make_string` instead.

`gh_string_length`
: Use `scm_string_length` instead.

`gh_string_ref`
: Use `scm_string_ref` instead.

`gh_string_set_x`
: Use `scm_string_set_x` instead.

`gh_substring`
: Use `scm_substring` instead.

`gh_string_append`
: Use `scm_string_append` instead.

`gh_cons`
: Use `scm_cons` instead.

`gh_car` and `gh_cdr`
: Use `scm_car` and `scm_cdr` instead.

`gh_cxxr` and `gh_cxxxr`
: (Where each x is either ‘`a`’ or ‘`d`’.) Use the corresponding `scm_cxxr` or `scm_cxxxr` function instead.

`gh_set_car_x` and `gh_set_cdr_x`
: Use `scm_set_car_x` and `scm_set_cdr_x` instead.

`gh_list`
: Use `scm_list_n` instead.

`gh_length`
: Replace `gh_length (`*`lst`*`)` with

```
scm_to_size_t (scm_length (lst))
```

`gh_append`
: Use `scm_append` instead.

`gh_append2`, `gh_append3`, `gh_append4`
: Replace `gh_append`*`N`*` (`*`l1`*`, ..., `*`lN`*`)` by

```
scm_append (scm_list_n (l1, ..., lN, SCM_UNDEFINED))
```

`gh_reverse`
: Use `scm_reverse` instead.

`gh_list_tail` and `gh_list_ref`
: Use `scm_list_tail` and `scm_list_ref` instead.

`gh_memq`, `gh_memv` and `gh_member`
: Use `scm_memq`, `scm_memv` and `scm_member` instead.

`gh_assq`, `gh_assv` and `gh_assoc`
: Use `scm_assq`, `scm_assv` and `scm_assoc` instead.

`gh_make_vector`
: Use `scm_make_vector` instead.

`gh_vector` or `gh_list_to_vector`
: Use `scm_vector` instead.

`gh_vector_ref` and `gh_vector_set_x`
: Use `scm_vector_ref` and `scm_vector_set_x` instead.

`gh_vector_length`
: Use `scm_c_vector_length` instead.

`gh_uniform_vector_length`
: Use `scm_c_uniform_vector_length` instead.

`gh_uniform_vector_ref`
: Use `scm_c_uniform_vector_ref` instead.

`gh_vector_to_list`
: Use `scm_vector_to_list` instead.

`gh_apply` Use `scm_apply_0` instead.

`gh_call0`
`gh_call1`
`gh_call2`
`gh_call3` Use `scm_call_0`, `scm_call_1`, etc instead.

`gh_display`
`gh_write`
`gh_newline`
: Use `scm_display (obj, scm_current_output_port ())` instead, etc.

`gh_lookup`
: Use `scm_variable_ref (scm_c_lookup (name))` instead.

`gh_module_lookup`
: Use `scm_variable_ref (scm_c_module_lookup (module, name))` instead.

5.22.3 GH preliminaries

To use gh, you must have the following toward the beginning of your C source:

```
#include <guile/gh.h>
```

When you link, you will have to add at least `-lguile` to the list of libraries. If you are using more of Guile than the basic Scheme interpreter, you will have to add more libraries.

5.22.4 Data types and constants defined by GH

The following C constants and data types are defined in gh:

`SCM` is a C data type used to store all Scheme data, no matter what the Scheme type. Values are converted between C data types and the SCM type with utility functions described below (see Section 5.22.9 [Converting data between C and Scheme], page 363). [FIXME: put in references to Jim's essay and so forth.]

`SCM_BOOL_T` [Constant]
`SCM_BOOL_F` [Constant]

The *Scheme* values returned by many boolean procedures in libguile.

This can cause confusion because they are different from 0 and 1. In testing a boolean function in libguile programming, you must always make sure that you check the spec:

`gh_` and `scm_` functions will usually return `SCM_BOOL_T` and `SCM_BOOL_F`, but other C functions usually can be tested against 0 and 1, so programmers' fingers tend to just type `if (boolean_function()) { ... }`

`SCM_UNSPECIFIED` [Constant]

This is a SCM value that is not the same as any legal Scheme value. It is the value that a Scheme function returns when its specification says that its return value is unspecified.

`SCM_UNDEFINED` [Constant]

This is another SCM value that is not the same as any legal Scheme value. It is the value used to mark variables that do not yet have a value, and it is also used in C to terminate functions with variable numbers of arguments, such as `gh_list()`.

5.22.5 Starting and controlling the interpreter

In almost every case, your first `gh_` call will be:

`void gh_enter` (*int* `argc`, *char* *`argv`[], *void* (*`main_prog`)()) [Function]

Starts up a Scheme interpreter with all the builtin Scheme primitives. `gh_enter()` never exits, and the user's code should all be in the *`main_prog`*`()` function. `argc` and `argv` will be passed to *main_prog*.

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

This is the user's main program. It will be invoked by `gh_enter()` after Guile has been started up.

Note that you can use `gh_repl` inside `gh_enter` (in other words, inside the code for `main-prog`) if you want the program to be controlled by a Scheme read-eval-print loop.

A convenience routine which enters the Guile interpreter with the standard Guile read-eval-print loop (*REPL*) is:

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

Enters the Scheme interpreter giving control to the Scheme REPL. Arguments are processed as if the Guile program '`guile`' were being invoked.

Note that `gh_repl` should be used *inside* `gh_enter`, since any Guile interpreter calls are meaningless unless they happen in the context of the interpreter.

Also note that when you use `gh_repl`, your program will be controlled by Guile's REPL (which is written in Scheme and has many useful features). Use straight C code inside `gh_enter` if you want to maintain execution control in your C program.

You will typically use `gh_enter` and `gh_repl` when you want a Guile interpreter enhanced by your own libraries, but otherwise quite normal. For example, to build a Guile–derived program that includes some random number routines *GSL* (GNU Scientific Library), you would write a C program that looks like this:

```
#include <guile/gh.h>
#include <gsl_ran.h>
```

```
/* random number suite */
SCM gw_ran_seed(SCM s)
{
  gsl_ran_seed(gh_scm2int(s));
  return SCM_UNSPECIFIED;
}

SCM gw_ran_random()
{
  SCM x;

  x = gh_ulong2scm(gsl_ran_random());
  return x;
}

SCM gw_ran_uniform()
{
  SCM x;

  x = gh_double2scm(gsl_ran_uniform());
  return x;
}
SCM gw_ran_max()
{
  return gh_double2scm(gsl_ran_max());
}

void
init_gsl()
{
  /* random number suite */
  gh_new_procedure("gsl-ran-seed", gw_ran_seed, 1, 0, 0);
  gh_new_procedure("gsl-ran-random", gw_ran_random, 0, 0, 0);
  gh_new_procedure("gsl-ran-uniform", gw_ran_uniform, 0, 0, 0);
  gh_new_procedure("gsl-ran-max", gw_ran_max, 0, 0, 0);
}

void
main_prog (int argc, char *argv[])
{
  init_gsl();

  gh_repl(argc, argv);
}

int
main (int argc, char *argv[])
{
  gh_enter (argc, argv, main_prog);
}
```

Then, supposing the C program is in '`guile-gsl.c`', you could compile it with *`gcc -o guile-gsl guile-gsl.c -lguile -lgsl`*.

The resulting program '`guile-gsl`' would have new primitive procedures `gsl-ran-random`, `gsl-ran-gaussian` and so forth.

5.22.6 Error messages

[FIXME: need to fill this based on Jim's new mechanism]

5.22.7 Executing Scheme code

Once you have an interpreter running, you can ask it to evaluate Scheme code. There are two calls that implement this:

SCM **gh_eval_str** (*char *scheme_code*) [Function]

This asks the interpreter to evaluate a single string of Scheme code, and returns the result of the last expression evaluated.

Note that the line of code in *scheme_code* must be a well formed Scheme expression. If you have many lines of code before you balance parentheses, you must either concatenate them into one string, or use `gh_eval_file()`.

SCM **gh_eval_file** (*char *fname*) [Function]
SCM **gh_load** (*char *fname*) [Function]

`gh_eval_file` is completely analogous to `gh_eval_str()`, except that a whole file is evaluated instead of a string. `gh_eval_file` returns `SCM_UNSPECIFIED`.

`gh_load` is identical to `gh_eval_file` (it's a macro that calls `gh_eval_file` on its argument). It is provided to start making the `gh_` interface match the R5RS Scheme procedures closely.

5.22.8 Defining new Scheme procedures in C

The real interface between C and Scheme comes when you can write new Scheme procedures in C. This is done through the routine

SCM **gh_new_procedure** (*char *proc_name, SCM (*fn)(), int n_required_args, int n_optional_args, int restp*) [Libguile high]

`gh_new_procedure` defines a new Scheme procedure. Its Scheme name will be *proc_name*, it will be implemented by the C function (**fn*)(), it will take at least *n_required_args* arguments, and at most *n_optional_args* extra arguments.

When the *restp* parameter is 1, the procedure takes a final argument: a list of remaining parameters.

`gh_new_procedure` returns an SCM value representing the procedure.

The C function *fn* should have the form

SCM **fn** (*SCM req1, SCM req2, ..., SCM opt1, SCM opt2, ..., SCM rest_args*) [Libguile high]

The arguments are all passed as SCM values, so the user will have to use the conversion functions to convert to standard C types.

Examples of C functions used as new Scheme primitives can be found in the sample programs `learn0` and `learn1`.

Rationale: this is the correct way to define new Scheme procedures in C. The ugly mess of arguments is required because of how C handles procedures with variable numbers of arguments.

NB: what about documentation strings?

> There are several important considerations to be made when writing the C routine `(*fn)()`.
>
> First of all the C routine has to return type `SCM`.
>
> Second, all arguments passed to the C function will be of type `SCM`.
>
> Third: the C routine is now subject to Scheme flow control, which means that it could be interrupted at any point, and then reentered. This means that you have to be very careful with operations such as allocating memory, modifying static data . . .
>
> Fourth: to get around the latter issue, you can use `GH_DEFER_INTS` and `GH_ALLOW_INTS`.

`GH_DEFER_INTS` [Macro]
`GH_ALLOW_INTS` [Macro]

These macros disable and re-enable Scheme's flow control. They

5.22.9 Converting data between C and Scheme

Guile provides mechanisms to convert data between C and Scheme. This allows new builtin procedures to understand their arguments (which are of type `SCM`) and return values of type `SCM`.

5.22.9.1 C to Scheme

`SCM gh_bool2scm` (*int* `x`) [Function]

Returns `#f` if *x* is zero, `#t` otherwise.

`SCM gh_ulong2scm` (*unsigned long* `x`) [Function]
`SCM gh_long2scm` (*long* `x`) [Function]
`SCM gh_double2scm` (*double* `x`) [Function]
`SCM gh_char2scm` (*char* `x`) [Function]

Returns a Scheme object with the value of the C quantity *x*.

`SCM gh_str2scm` (*char* `*s`, *int* `len`) [Function]

Returns a new Scheme string with the (not necessarily null-terminated) C array *s* data.

`SCM gh_str02scm` (*char* `*s`) [Function]

Returns a new Scheme string with the null-terminated C string *s* data.

`SCM gh_set_substr` (*char* `*src`, *SCM* `dst`, *int* `start`, *int* `len`) [Function]

Copy *len* characters at *src* into the *existing* Scheme string *dst*, starting at *start*. *start* is an index into *dst*; zero means the beginning of the string.

If *start* + *len* is off the end of *dst*, signal an out-of-range error.

`SCM gh_symbol2scm` (*char* `*name`) [Function]

Given a null-terminated string *name*, return the symbol with that name.

`SCM gh_ints2scm` (*int* `*dptr`, *int* `n`) [Function]
`SCM gh_doubles2scm` (*double* `*dptr`, *int* `n`) [Function]

Make a scheme vector containing the *n* ints or doubles at memory location *dptr*.

SCM gh_chars2byvect (*char *dptr, int n*) [Function]
SCM gh_shorts2svect (*short *dptr, int n*) [Function]
SCM gh_longs2ivect (*long *dptr, int n*) [Function]
SCM gh_ulongs2uvect (*ulong *dptr, int n*) [Function]
SCM gh_floats2fvect (*float *dptr, int n*) [Function]
SCM gh_doubles2dvect (*double *dptr, int n*) [Function]

Make a scheme uniform vector containing the *n* chars, shorts, longs, unsigned longs, floats or doubles at memory location *dptr*.

5.22.9.2 Scheme to C

int gh_scm2bool (*SCM obj*) [Function]
unsigned long gh_scm2ulong (*SCM obj*) [Function]
long gh_scm2long (*SCM obj*) [Function]
double gh_scm2double (*SCM obj*) [Function]
int gh_scm2char (*SCM obj*) [Function]

These routines convert the Scheme object to the given C type.

char * gh_scm2newstr (*SCM str, size_t *lenp*) [Function]

Given a Scheme string *str*, return a pointer to a new copy of its contents, followed by a null byte. If *lenp* is non-null, set `*lenp` to the string's length.

This function uses malloc to obtain storage for the copy; the caller is responsible for freeing it.

Note that Scheme strings may contain arbitrary data, including null characters. This means that null termination is not a reliable way to determine the length of the returned value. However, the function always copies the complete contents of *str*, and sets **lenp* to the true length of the string (when *lenp* is non-null).

void gh_get_substr (*SCM str, char *return_str, int *lenp*) [Function]

Copy *len* characters at *start* from the Scheme string *src* to memory at *dst*. *start* is an index into *src*; zero means the beginning of the string. *dst* has already been allocated by the caller.

If *start* + *len* is off the end of *src*, signal an out-of-range error.

char * gh_symbol2newstr (*SCM sym, int *lenp*) [Function]

Takes a Scheme symbol and returns a string of the form `"'symbol-name"`. If *lenp* is non-null, the string's length is returned in `*lenp`.

This function uses malloc to obtain storage for the returned string; the caller is responsible for freeing it.

char * gh_scm2chars (*SCM vector, chars *result*) [Function]
short * gh_scm2shorts (*SCM vector, short *result*) [Function]
long * gh_scm2longs (*SCM vector, long *result*) [Function]
float * gh_scm2floats (*SCM vector, float *result*) [Function]
double * gh_scm2doubles (*SCM vector, double *result*) [Function]

Copy the numbers in *vector* to the array pointed to by *result* and return it. If *result* is NULL, allocate a double array large enough.

vector can be an ordinary vector, a weak vector, or a signed or unsigned uniform vector of the same type as the result array. For chars, *vector* can be a string or substring. For floats and doubles, *vector* can contain a mix of inexact and integer values.

If *vector* is of unsigned type and contains values too large to fit in the signed destination array, those values will be wrapped around, that is, data will be copied as if the destination array was unsigned.

5.22.10 Type predicates

These C functions mirror Scheme's type predicate procedures with one important difference. The C routines return C boolean values (0 and 1) instead of `SCM_BOOL_T` and `SCM_BOOL_F`.

The Scheme notational convention of putting a `?` at the end of predicate procedure names is mirrored in C by placing `_p` at the end of the procedure. For example, `(pair? ...)` maps to `gh_pair_p(...)`.

`int gh_boolean_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a boolean, 0 otherwise.

`int gh_symbol_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a symbol, 0 otherwise.

`int gh_char_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a char, 0 otherwise.

`int gh_vector_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a vector, 0 otherwise.

`int gh_pair_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a pair, 0 otherwise.

`int gh_procedure_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a procedure, 0 otherwise.

`int gh_list_p` (*SCM* `val`) [Function]
Returns 1 if *val* is a list, 0 otherwise.

`int gh_inexact_p` (*SCM* `val`) [Function]
Returns 1 if *val* is an inexact number, 0 otherwise.

`int gh_exact_p` (*SCM* `val`) [Function]
Returns 1 if *val* is an exact number, 0 otherwise.

5.22.11 Equality predicates

These C functions mirror Scheme's equality predicate procedures with one important difference. The C routines return C boolean values (0 and 1) instead of `SCM_BOOL_T` and `SCM_BOOL_F`.

The Scheme notational convention of putting a `?` at the end of predicate procedure names is mirrored in C by placing `_p` at the end of the procedure. For example, `(equal? ...)` maps to `gh_equal_p(...)`.

`int gh_eq_p` (*SCM x, SCM y*) [Function]
: Returns 1 if *x* and *y* are equal in the sense of Scheme's `eq?` predicate, 0 otherwise.

`int gh_eqv_p` (*SCM x, SCM y*) [Function]
: Returns 1 if *x* and *y* are equal in the sense of Scheme's `eqv?` predicate, 0 otherwise.

`int gh_equal_p` (*SCM x, SCM y*) [Function]
: Returns 1 if *x* and *y* are equal in the sense of Scheme's `equal?` predicate, 0 otherwise.

`int gh_string_equal_p` (*SCM* `s1`, *SCM* `s2`) [Function]
: Returns 1 if the strings *s1* and *s2* are equal, 0 otherwise.

`int gh_null_p` (*SCM* `l`) [Function]
: Returns 1 if *l* is an empty list or pair; 0 otherwise.

5.22.12 Memory allocation and garbage collection

5.22.13 Calling Scheme procedures from C

Many of the Scheme primitives are available in the `gh_` interface; they take and return objects of type SCM, and one could basically use them to write C code that mimics Scheme code.

I will list these routines here without much explanation, since what they do is the same as documented in section "Standard procedures" in *R5RS*. But I will point out that when a procedure takes a variable number of arguments (such as `gh_list`), you should pass the constant *SCM_UNDEFINED* from C to signify the end of the list.

`SCM gh_define` (*char* *`name`, *SCM* `val`) [Function]
: Corresponds to the Scheme `(define name val)`: it binds a value to the given name (which is a C string). Returns the new object.

Pairs and lists

`SCM gh_cons` (*SCM* `a`, *SCM* `b`) [Function]
`SCM gh_list` (*SCM l0, SCM l1, ... , SCM_UNDEFINED*) [Function]
: These correspond to the Scheme `(cons a b)` and `(list l0 l1 ...)` procedures. Note that `gh_list()` is a C macro that invokes `scm_list_n()`.

`SCM gh_car` (*SCM* `obj`) [Function]
`SCM gh_cdr` (*SCM* `obj`) [Function]
: ...

`SCM gh_c[ad][ad][ad][ad]r` (*SCM* `obj`) [Function]
: These correspond to the Scheme `(caadar ls)` procedures etc ...

`SCM gh_set_car_x` (*SCM* `pair`, *SCM* `value`) [Function]
: Modifies the CAR of *pair* to be *value*. This is equivalent to the Scheme procedure `(set-car! ...)`.

`SCM gh_set_cdr_x` (*SCM* `pair`, *SCM* `value`) [Function]
: Modifies the CDR of *pair* to be *value*. This is equivalent to the Scheme procedure `(set-cdr! ...)`.

unsigned long **gh_length** (*SCM* `ls`) [Function]

Returns the length of the list.

SCM **gh_append** (*SCM* `args`) [Function]
SCM **gh_append2** (*SCM* `l1`, *SCM* `l2`) [Function]
SCM **gh_append3** (*SCM* `l1`, *SCM* `l2`, `l3`) [Function]
SCM **gh_append4** (*SCM* `l1`, *SCM* `l2`, `l3`, `l4`) [Function]

`gh_append()` takes *args*, which is a list of lists `(list1 list2 ...)`, and returns a list containing all the elements of the individual lists.

A typical invocation of `gh_append()` to append 5 lists together would be

```
gh_append(gh_list(l1, l2, l3, l4, l5, SCM_UNDEFINED));
```

The functions `gh_append2()`, `gh_append2()`, `gh_append3()` and `gh_append4()` are convenience routines to make it easier for C programs to form the list of lists that goes as an argument to `gh_append()`.

SCM **gh_reverse** (*SCM* `ls`) [Function]

Returns a new list that has the same elements as *ls* but in the reverse order. Note that this is implemented as a macro which calls `scm_reverse()`.

SCM **gh_list_tail** (*SCM* `ls`, *SCM* `k`) [Function]

Returns the sublist of *ls* with the last *k* elements.

SCM **gh_list_ref** (*SCM* `ls`, *SCM* `k`) [Function]

Returns the *k*th element of the list *ls*.

SCM **gh_memq** (*SCM* `x`, *SCM* `ls`) [Function]
SCM **gh_memv** (*SCM* `x`, *SCM* `ls`) [Function]
SCM **gh_member** (*SCM* `x`, *SCM* `ls`) [Function]

These functions return the first sublist of *ls* whose CAR is *x*. They correspond to `(memq x ls)`, `(memv x ls)` and `(member x ls)`, and hence use (respectively) `eq?`, `eqv?` and `equal?` to do comparisons.

If *x* does not appear in *ls*, the value `SCM_BOOL_F` (not the empty list) is returned.

Note that these functions are implemented as macros which call `scm_memq()`, `scm_memv()` and `scm_member()` respectively.

SCM **gh_assq** (*SCM* `x`, *SCM* `alist`) [Function]
SCM **gh_assv** (*SCM* `x`, *SCM* `alist`) [Function]
SCM **gh_assoc** (*SCM* `x`, *SCM* `alist`) [Function]

These functions search an *association list* (list of pairs) *alist* for the first pair whose CAR is *x*, and they return that pair.

If no pair in *alist* has *x* as its CAR, the value `SCM_BOOL_F` (not the empty list) is returned.

Note that these functions are implemented as macros which call `scm_assq()`, `scm_assv()` and `scm_assoc()` respectively.

Symbols

Vectors

`SCM gh_make_vector` (*SCM* `n`, *SCM* `fill`) [Function]
`SCM gh_vector` (*SCM* `ls`) [Function]
`SCM gh_vector_ref` (*SCM* `v`, *SCM* `i`) [Function]
`SCM gh_vector_set` (*SCM* `v`, *SCM* `i`, *SCM* `val`) [Function]
`unsigned long gh_vector_length` (*SCM* `v`) [Function]
`SCM gh_list_to_vector` (*SCM* `ls`) [Function]

These correspond to the Scheme `(make-vector n fill)`, `(vector a b c ...)` `(vector-ref v i)` `(vector-set v i value)` `(vector-length v)` `(list->vector ls)` procedures.

The correspondence is not perfect for `gh_vector`: this routine takes a list *ls* instead of the individual list elements, thus making it identical to `gh_list_to_vector`.

There is also a difference in gh_vector_length: the value returned is a C `unsigned long` instead of an SCM object.

Procedures

`SCM gh_apply` (*SCM proc*, *SCM args*) [Function]

Call the Scheme procedure *proc*, with the elements of *args* as arguments. *args* must be a proper list.

`SCM gh_call0` (*SCM proc*) [Function]
`SCM gh_call1` (*SCM proc*, *SCM arg*) [Function]
`SCM gh_call2` (*SCM proc*, *SCM arg1*, *SCM arg2*) [Function]
`SCM gh_call3` (*SCM proc*, *SCM arg1*, *SCM arg2*, *SCM arg3*) [Function]

Call the Scheme procedure *proc* with no arguments (`gh_call0`), one argument (`gh_call1`), and so on. You can get the same effect by wrapping the arguments up into a list, and calling `gh_apply`; Guile provides these functions for convenience.

`SCM gh_catch` (*SCM key*, *SCM thunk*, *SCM handler*) [Function]
`SCM gh_throw` (*SCM key*, *SCM args*) [Function]

Corresponds to the Scheme `catch` and `throw` procedures, which in Guile are provided as primitives.

`SCM gh_is_eq` (*SCM a*, *SCM b*) [Function]
`SCM gh_is_eqv` (*SCM a*, *SCM b*) [Function]
`SCM gh_is_equal` (*SCM a*, *SCM b*) [Function]

These correspond to the Scheme `eq?`, `eqv?` and `equal?` predicates.

`int gh_obj_length` (*SCM* `obj`) [Function]

Returns the raw object length.

Data lookup

For now I just include Tim Pierce's comments from the '`gh_data.c`' file; it should be organized into a documentation of the two functions here.

```
/* Data lookups between C and Scheme

   Look up a symbol with a given name, and return the object to which
   it is bound.  gh_lookup examines the Guile top level, and
   gh_module_lookup checks the module name space specified by the
   ‘vec’ argument.

   The return value is the Scheme object to which SNAME is bound, or
   SCM_UNDEFINED if SNAME is not bound in the given context. [FIXME:
   should this be SCM_UNSPECIFIED?  Can a symbol ever legitimately be
   bound to SCM_UNDEFINED or SCM_UNSPECIFIED?  What is the difference?
   -twp] */
```

6 Guile Modules

6.1 SLIB

Before the SLIB facilities can be used, the following Scheme expression must be executed:

```
(use-modules (ice-9 slib))
```

`require` can then be used in the usual way (see section "Require" in *The SLIB Manual*). For example,

```
(use-modules (ice-9 slib))
(require 'primes)
(probably-prime? 13)
⇒ #t
```

Note that the following Guile core functions are overridden by `(ice-9 slib)`, to implement SLIB specified semantics.

`delete-file`
: Returns `#t` for success or `#f` for failure (see section "Input/Output" in *The SLIB Manual*), as opposed to the Guile core version unspecified for success and throwing an error for failure (see Section 6.2.3 [File System], page 381).

`provided?`
: Accepts a feature specification containing `and` and `or` forms combining symbols (see section "Feature" in *The SLIB Manual*), as opposed to the Guile core taking only plain symbols (see Section 5.18.2.1 [Feature Manipulation], page 336).

`open-file`
: Takes a symbol `r`, `rb`, `w` or `wb` for the open mode (see section "Input/Output" in *The SLIB Manual*), as opposed to the Guile core version taking a string (see Section 5.12.9.1 [File Ports], page 280).

`system`
: Returns a plain exit code 0 to 255 (see section "System Interface" in *The SLIB Manual*), as opposed to the Guile core version returning a wait status that must be examined with `status:exit-val` etc (see Section 6.2.7 [Processes], page 393).

6.1.1 SLIB installation

The following seems to work (e.g., with slib versions 2c7 and 2d2):

1. Unpack slib somewhere, e.g., '`/usr/local/share/slib`'.
2. Create a symlink in the Guile site directory to slib, e.g.,:

   ```
   ln -s /usr/local/share/slib /usr/local/share/guile/site/slib
   ```

3. Use Guile to create the catalog file, e.g.,:

   ```
   # guile
   guile> (use-modules (ice-9 slib))
   guile> (load "/usr/local/share/slib/mklibcat.scm")
   guile> (quit)
   ```

   The catalog data should now be in '`/usr/local/share/guile/site/slibcat`'.

   If instead you get an error such as:

   ```
   Unbound variable: scheme-implementation-type
   ```

   then a solution is to get a newer version of Guile, or to modify '`ice-9/slib.scm`' to use `define-public` for the offending variables.

4. Install the documentation:

```
cd /usr/local/share/slib
rm /usr/local/info/slib.info*
cp slib.info /usr/local/info
install-info slib.info /usr/local/info/dir
```

6.1.2 JACAL

Jacal is a symbolic math package written in Scheme by Aubrey Jaffer. It is usually installed as an extra package in SLIB.

You can use Guile's interface to SLIB to invoke Jacal:

```
(use-modules (ice-9 slib))
(slib:load "math")
(math)
```

For complete documentation on Jacal, please read the Jacal manual. If it has been installed on line, you can look at section "Jacal" in *JACAL Symbolic Mathematics System*. Otherwise you can find it on the web at `http://www-swiss.ai.mit.edu/~jaffer/JACAL.html`

6.2 POSIX System Calls and Networking

6.2.1 POSIX Interface Conventions

These interfaces provide access to operating system facilities. They provide a simple wrapping around the underlying C interfaces to make usage from Scheme more convenient. They are also used to implement the Guile port of scsh (see Section 6.14 [The Scheme shell (scsh)], page 483).

Generally there is a single procedure for each corresponding Unix facility. There are some exceptions, such as procedures implemented for speed and convenience in Scheme with no primitive Unix equivalent, e.g. `copy-file`.

The interfaces are intended as far as possible to be portable across different versions of Unix. In some cases procedures which can't be implemented on particular systems may become no-ops, or perform limited actions. In other cases they may throw errors.

General naming conventions are as follows:

- The Scheme name is often identical to the name of the underlying Unix facility.
- Underscores in Unix procedure names are converted to hyphens.
- Procedures which destructively modify Scheme data have exclamation marks appended, e.g., `recv!`.
- Predicates (returning only `#t` or `#f`) have question marks appended, e.g., `access?`.
- Some names are changed to avoid conflict with dissimilar interfaces defined by scsh, e.g., `primitive-fork`.
- Unix preprocessor names such as `EPERM` or `R_OK` are converted to Scheme variables of the same name (underscores are not replaced with hyphens).

Unexpected conditions are generally handled by raising exceptions. There are a few procedures which return a special value if they don't succeed, e.g., `getenv` returns `#f` if it the requested string is not found in the environment. These cases are noted in the documentation.

For ways to deal with exceptions, see Section 5.11.7 [Exceptions], page 258.

Errors which the C library would report by returning a null pointer or through some other means are reported by raising a `system-error` exception with `scm-error` (see Section 5.11.8 [Error Reporting], page 265). The *data* parameter is a list containing the Unix `errno` value (an integer). For example,

```
(define (my-handler key func fmt fmtargs data)
  (display key) (newline)
  (display func) (newline)
  (apply format #t fmt fmtargs) (newline)
  (display data) (newline))

(catch 'system-error
  (lambda () (dup2 -123 -456))
  my-handler)

 ⊣
system-error
dup2
Bad file descriptor
(9)
```

`system-error-errno` *arglist* [Function]

Return the `errno` value from a list which is the arguments to an exception handler. If the exception is not a `system-error`, then the return is `#f`. For example,

```
(catch
 'system-error
 (lambda ()
   (mkdir "/this-ought-to-fail-if-I'm-not-root"))
 (lambda stuff
   (let ((errno (system-error-errno stuff)))
     (cond
      ((= errno EACCES)
       (display "You're not allowed to do that."))
      ((= errno EEXIST)
       (display "Already exists."))
      (#t
       (display (strerror errno))))
     (newline))))
```

6.2.2 Ports and File Descriptors

Conventions generally follow those of scsh, Section 6.14 [The Scheme shell (scsh)], page 483.

File ports are implemented using low-level operating system I/O facilities, with optional buffering to improve efficiency; see Section 5.12.9.1 [File Ports], page 280.

Note that some procedures (e.g., `recv!`) will accept ports as arguments, but will actually operate directly on the file descriptor underlying the port. Any port buffering is ignored, including the buffer which implements `peek-char` and `unread-char`.

The `force-output` and `drain-input` procedures can be used to clear the buffers.

Each open file port has an associated operating system file descriptor. File descriptors are generally not useful in Scheme programs; however they may be needed when interfacing with foreign code and the Unix environment.

A file descriptor can be extracted from a port and a new port can be created from a file descriptor. However a file descriptor is just an integer and the garbage collector doesn't recognize it as a reference to the port. If all other references to the port were dropped, then it's likely that the garbage collector would free the port, with the side-effect of closing the file descriptor prematurely.

To assist the programmer in avoiding this problem, each port has an associated *revealed count* which can be used to keep track of how many times the underlying file descriptor has been stored in other places. If a port's revealed count is greater than zero, the file descriptor will not be closed when the port is garbage collected. A programmer can therefore ensure that the revealed count will be greater than zero if the file descriptor is needed elsewhere.

For the simple case where a file descriptor is "imported" once to become a port, it does not matter if the file descriptor is closed when the port is garbage collected. There is no need to maintain a revealed count. Likewise when "exporting" a file descriptor to the external environment, setting the revealed count is not required provided the port is kept open (i.e., is pointed to by a live Scheme binding) while the file descriptor is in use.

To correspond with traditional Unix behaviour, three file descriptors (0, 1, and 2) are automatically imported when a program starts up and assigned to the initial values of the current/standard input, output, and error ports, respectively. The revealed count for each is initially set to one, so that dropping references to one of these ports will not result in its garbage collection: it could be retrieved with `fdopen` or `fdes->ports`.

`port-revealed` *port* [Scheme Procedure]
`scm_port_revealed` (*port*) [C Function]

Return the revealed count for *port*.

`set-port-revealed!` *port rcount* [Scheme Procedure]
`scm_set_port_revealed_x` (*port, rcount*) [C Function]

Sets the revealed count for a *port* to *rcount*. The return value is unspecified.

`fileno` *port* [Scheme Procedure]
`scm_fileno` (*port*) [C Function]

Return the integer file descriptor underlying *port*. Does not change its revealed count.

`port->fdes` *port* [Scheme Procedure]

Returns the integer file descriptor underlying *port*. As a side effect the revealed count of *port* is incremented.

`fdopen` *fdes modes* [Scheme Procedure]
`scm_fdopen` (*fdes, modes*) [C Function]

Return a new port based on the file descriptor *fdes*. Modes are given by the string *modes*. The revealed count of the port is initialized to zero. The *modes* string is the same as that accepted by `open-file` (see Section 5.12.9.1 [File Ports], page 280).

`fdes->ports` *fd* [Scheme Procedure]
`scm_fdes_to_ports` (*fd*) [C Function]

Return a list of existing ports which have *fdes* as an underlying file descriptor, without changing their revealed counts.

`fdes->inport` *fdes* [Scheme Procedure]

Returns an existing input port which has *fdes* as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new input port with a revealed count of 1.

`fdes->outport` *fdes* [Scheme Procedure]

Returns an existing output port which has *fdes* as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new output port with a revealed count of 1.

`primitive-move->fdes` *port fd* [Scheme Procedure]
`scm_primitive_move_to_fdes` (*port, fd*) [C Function]

Moves the underlying file descriptor for *port* to the integer value *fdes* without changing the revealed count of *port*. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is `#f` if the file descriptor already had the required value or `#t` if it was moved.

`move->fdes` *port fdes* [Scheme Procedure]

Moves the underlying file descriptor for *port* to the integer value *fdes* and sets its revealed count to one. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is unspecified.

`release-port-handle` *port* [Scheme Procedure]

Decrements the revealed count for a port.

`fsync` *object* [Scheme Procedure]
`scm_fsync` (*object*) [C Function]

Copies any unwritten data for the specified output file descriptor to disk. If *port/fd* is a port, its buffer is flushed before the underlying file descriptor is fsync'd. The return value is unspecified.

`open` *path flags* [*mode*] [Scheme Procedure]
`scm_open` (*path, flags, mode*) [C Function]

Open the file named by *path* for reading and/or writing. *flags* is an integer specifying how the file should be opened. *mode* is an integer specifying the permission bits of the file, if it needs to be created, before the umask (see Section 6.2.7 [Processes], page 393) is applied. The default is 666 (Unix itself has no default).

flags can be constructed by combining variables using `logior`. Basic flags are:

`O_RDONLY` [Variable]

Open the file read-only.

`O_WRONLY` [Variable]

Open the file write-only.

`O_RDWR` [Variable]

Open the file read/write.

`O_APPEND` [Variable]

Append to the file instead of truncating.

`O_CREAT` [Variable]

Create the file if it does not already exist.

See section "File Status Flags" in *The GNU C Library Reference Manual*, for additional flags.

`open-fdes` *path flags* [*mode*] [Scheme Procedure]
`scm_open_fdes` (*path, flags, mode*) [C Function]

Similar to `open` but return a file descriptor instead of a port.

`close` *fd_or_port* [Scheme Procedure]
`scm_close` (*fd_or_port*) [C Function]

Similar to `close-port` (see Section 5.12.4 [Closing], page 275), but also works on file descriptors. A side effect of closing a file descriptor is that any ports using that file descriptor are moved to a different file descriptor and have their revealed counts set to zero.

`close-fdes` *fd* [Scheme Procedure]
`scm_close_fdes` (*fd*) [C Function]

A simple wrapper for the `close` system call. Close file descriptor *fd*, which must be an integer. Unlike `close`, the file descriptor will be closed even if a port is using it. The return value is unspecified.

`unread-char` *char* [*port*] [Scheme Procedure]
`scm_unread_char` (*char, port*) [C Function]

Place *char* in *port* so that it will be read by the next read operation on that port. If called multiple times, the unread characters will be read again in "last-in, first-out" order (i.e. a stack). If *port* is not supplied, the current input port is used.

`unread-string` *str port* [Scheme Procedure]

Place the string *str* in *port* so that its characters will be read in subsequent read operations. If called multiple times, the unread characters will be read again in last-in first-out order. If *port* is not supplied, the current-input-port is used.

`pipe` [Scheme Procedure]
`scm_pipe` () [C Function]

Return a newly created pipe: a pair of ports which are linked together on the local machine. The CAR is the input port and the CDR is the output port. Data written (and flushed) to the output port can be read from the input port. Pipes are commonly used for communication with a newly forked child process. The need to flush the output port can be avoided by making it unbuffered using `setvbuf`.

`PIPE_BUF` [Variable]

A write of up to `PIPE_BUF` many bytes to a pipe is atomic, meaning when done it goes into the pipe instantaneously and as a contiguous block (see section "Atomicity of Pipe I/O" in *The GNU C Library Reference Manual*).

Note that the output port is likely to block if too much data has been written but not yet read from the input port. Typically the capacity is `PIPE_BUF` bytes.

The next group of procedures perform a `dup2` system call, if *newfd* (an integer) is supplied, otherwise a `dup`. The file descriptor to be duplicated can be supplied as an integer or contained in a port. The type of value returned varies depending on which procedure is used.

All procedures also have the side effect when performing `dup2` that any ports using *newfd* are moved to a different file descriptor and have their revealed counts set to zero.

`dup->fdes` *fd_or_port* [*fd*] [Scheme Procedure]
`scm_dup_to_fdes` (*fd_or_port*, *fd*) [C Function]

Return a new integer file descriptor referring to the open file designated by *fd_or_port*, which must be either an open file port or a file descriptor.

`dup->inport` *port/fd* [*newfd*] [Scheme Procedure]

Returns a new input port using the new file descriptor.

`dup->outport` *port/fd* [*newfd*] [Scheme Procedure]

Returns a new output port using the new file descriptor.

`dup` *port/fd* [*newfd*] [Scheme Procedure]

Returns a new port if *port/fd* is a port, with the same mode as the supplied port, otherwise returns an integer file descriptor.

`dup->port` *port/fd mode* [*newfd*] [Scheme Procedure]

Returns a new port using the new file descriptor. *mode* supplies a mode string for the port (see Section 5.12.9.1 [File Ports], page 280).

`duplicate-port` *port modes* [Scheme Procedure]

Returns a new port which is opened on a duplicate of the file descriptor underlying *port*, with mode string *modes* as for Section 5.12.9.1 [File Ports], page 280. The two ports will share a file position and file status flags.

Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered. The mode string can include `0` to obtain an unbuffered duplicate port.

This procedure is equivalent to `(dup->port` *`port modes`*`)`.

`redirect-port` *old new* [Scheme Procedure]
`scm_redirect_port` (*old*, *new*) [C Function]

This procedure takes two ports and duplicates the underlying file descriptor from *old-port* into *new-port*. The current file descriptor in *new-port* will be closed. After the redirection the two ports will share a file position and file status flags.

The return value is unspecified.

Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered.

This procedure does not have any side effects on other ports or revealed counts.

`dup2` *oldfd newfd* [Scheme Procedure]
`scm_dup2` (*oldfd, newfd*) [C Function]

A simple wrapper for the `dup2` system call. Copies the file descriptor *oldfd* to descriptor number *newfd*, replacing the previous meaning of *newfd*. Both *oldfd* and *newfd* must be integers. Unlike for `dup->fdes` or `primitive-move->fdes`, no attempt is made to move away ports which are using *newfd*. The return value is unspecified.

`port-mode` *port* [Scheme Procedure]

Return the port modes associated with the open port *port*. These will not necessarily be identical to the modes used when the port was opened, since modes such as "append" which are used only during port creation are not retained.

`port-for-each` *proc* [Scheme Procedure]
`scm_port_for_each` (*SCM proc*) [C Function]
`scm_c_port_for_each` (*void (*proc)(void *, SCM), void *data*) [C Function]

Apply *proc* to each port in the Guile port table (FIXME: what is the Guile port table?) in turn. The return value is unspecified. More specifically, *proc* is applied exactly once to every port that exists in the system at the time `port-for-each` is invoked. Changes to the port table while `port-for-each` is running have no effect as far as `port-for-each` is concerned.

The C function `scm_port_for_each` takes a Scheme procedure encoded as a `SCM` value, while `scm_c_port_for_each` takes a pointer to a C function and passes along a arbitrary *data* cookie.

`setvbuf` *port mode* [*size*] [Scheme Procedure]
`scm_setvbuf` (*port, mode, size*) [C Function]

Set the buffering mode for *port*. *mode* can be:

`_IONBF` [Variable]

non-buffered

`_IOLBF` [Variable]

line buffered

`_IOFBF` [Variable]

block buffered, using a newly allocated buffer of *size* bytes. If *size* is omitted, a default size will be used.

`fcntl` *port/fd cmd* [*value*] [Scheme Procedure]
`scm_fcntl` (*object, cmd, value*) [C Function]

Apply *cmd* on *port/fd*, either a port or file descriptor. The *value* argument is used by the `SET` commands described below, it's an integer value.

Values for *cmd* are:

`F_DUPFD` [Variable]

Duplicate the file descriptor, the same as `dup->fdes` above does.

`F_GETFD` [Variable]
`F_SETFD` [Variable]
Get or set flags associated with the file descriptor. The only flag is the following,

`FD_CLOEXEC` [Variable]
"Close on exec", meaning the file descriptor will be closed on an `exec` call (a successful such call). For example to set that flag,

```
(fcntl port F_SETFD FD_CLOEXEC)
```

Or better, set it but leave any other possible future flags unchanged,

```
(fcntl port F_SETFD (logior FD_CLOEXEC
                            (fcntl port F_GETFD)))
```

`F_GETFL` [Variable]
`F_SETFL` [Variable]
Get or set flags associated with the open file. These flags are `O_RDONLY` etc described under `open` above.

A common use is to set `O_NONBLOCK` on a network socket. The following sets that flag, and leaves other flags unchanged.

```
(fcntl sock F_SETFL (logior O_NONBLOCK
                            (fcntl sock F_GETFL)))
```

`F_GETOWN` [Variable]
`F_SETOWN` [Variable]
Get or set the process ID of a socket's owner, for `SIGIO` signals.

`flock` *file operation* [Scheme Procedure]
`scm_flock` (*file, operation*) [C Function]
Apply or remove an advisory lock on an open file. *operation* specifies the action to be done:

`LOCK_SH` [Variable]
Shared lock. More than one process may hold a shared lock for a given file at a given time.

`LOCK_EX` [Variable]
Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

`LOCK_UN` [Variable]
Unlock the file.

`LOCK_NB` [Variable]
Don't block when locking. This is combined with one of the other operations using `logior` (see Section 5.5.2.14 [Bitwise Operations], page 117). If `flock` would block an `EWOULDBLOCK` error is thrown (see Section 6.2.1 [Conventions], page 373).

The return value is not specified. *file* may be an open file descriptor or an open file descriptor port.

Note that `flock` does not lock files across NFS.

`select` *reads writes excepts* [*secs* [*usecs*]] [Scheme Procedure]
`scm_select` (*reads, writes, excepts, secs, usecs*) [C Function]

This procedure has a variety of uses: waiting for the ability to provide input, accept output, or the existence of exceptional conditions on a collection of ports or file descriptors, or waiting for a timeout to occur. It also returns if interrupted by a signal.

reads, *writes* and *excepts* can be lists or vectors, with each member a port or a file descriptor. The value returned is a list of three corresponding lists or vectors containing only the members which meet the specified requirement. The ability of port buffers to provide input or accept output is taken into account. Ordering of the input lists or vectors is not preserved.

The optional arguments *secs* and *usecs* specify the timeout. Either *secs* can be specified alone, as either an integer or a real number, or both *secs* and *usecs* can be specified as integers, in which case *usecs* is an additional timeout expressed in microseconds. If *secs* is omitted or is `#f` then select will wait for as long as it takes for one of the other conditions to be satisfied.

The scsh version of `select` differs as follows: Only vectors are accepted for the first three arguments. The *usecs* argument is not supported. Multiple values are returned instead of a list. Duplicates in the input vectors appear only once in output. An additional `select!` interface is provided.

6.2.3 File System

These procedures allow querying and setting file system attributes (such as owner, permissions, sizes and types of files); deleting, copying, renaming and linking files; creating and removing directories and querying their contents; syncing the file system and creating special files.

`access?` *path how* [Scheme Procedure]
`scm_access` (*path, how*) [C Function]

Test accessibility of a file under the real UID and GID of the calling process. The return is `#t` if *path* exists and the permissions requested by *how* are all allowed, or `#f` if not.

how is an integer which is one of the following values, or a bitwise-OR (`logior`) of multiple values.

`R_OK` [Variable]

Test for read permission.

`W_OK` [Variable]

Test for write permission.

`X_OK` [Variable]

Test for execute permission.

`F_OK` [Variable]

Test for existence of the file. This is implied by each of the other tests, so there's no need to combine it with them.

It's important to note that `access?` does not simply indicate what will happen on attempting to read or write a file. In normal circumstances it does, but in a set-UID or set-GID program it doesn't because `access?` tests the real ID, whereas an open or execute attempt uses the effective ID.

A program which will never run set-UID/GID can ignore the difference between real and effective IDs, but for maximum generality, especially in library functions, it's best not to use `access?` to predict the result of an open or execute, instead simply attempt that and catch any exception.

The main use for `access?` is to let a set-UID/GID program determine what the invoking user would have been allowed to do, without the greater (or perhaps lesser) privileges afforded by the effective ID. For more on this, see section "Testing File Access" in *The GNU C Library Reference Manual*.

`stat` *object* [Scheme Procedure]
`scm_stat` (*object*) [C Function]

Return an object containing various information about the file determined by *obj*. *obj* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fstat` is used as the underlying system call).

The object returned by `stat` can be passed as a single parameter to the following procedures, all of which return integers:

`stat:dev` *st* [Scheme Procedure]

The device number containing the file.

`stat:ino` *st* [Scheme Procedure]

The file serial number, which distinguishes this file from all other files on the same device.

`stat:mode` *st* [Scheme Procedure]

The mode of the file. This is an integer which incorporates file type information and file permission bits. See also `stat:type` and `stat:perms` below.

`stat:nlink` *st* [Scheme Procedure]

The number of hard links to the file.

`stat:uid` *st* [Scheme Procedure]

The user ID of the file's owner.

`stat:gid` *st* [Scheme Procedure]

The group ID of the file.

`stat:rdev` *st* [Scheme Procedure]

Device ID; this entry is defined only for character or block special files. On some systems this field is not available at all, in which case `stat:rdev` returns `#f`.

`stat:size` *st* [Scheme Procedure]

The size of a regular file in bytes.

`stat:atime` *st* [Scheme Procedure]
The last access time for the file.

`stat:mtime` *st* [Scheme Procedure]
The last modification time for the file.

`stat:ctime` *st* [Scheme Procedure]
The last modification time for the attributes of the file.

`stat:blksize` *st* [Scheme Procedure]
The optimal block size for reading or writing the file, in bytes. On some systems this field is not available, in which case `stat:blksize` returns a sensible suggested block size.

`stat:blocks` *st* [Scheme Procedure]
The amount of disk space that the file occupies measured in units of 512 byte blocks. On some systems this field is not available, in which case `stat:blocks` returns `#f`.

In addition, the following procedures return the information from `stat:mode` in a more convenient form:

`stat:type` *st* [Scheme Procedure]
A symbol representing the type of file. Possible values are '`regular`', '`directory`', '`symlink`', '`block-special`', '`char-special`', '`fifo`', '`socket`', and '`unknown`'.

`stat:perms` *st* [Scheme Procedure]
An integer representing the access permission bits.

`lstat` *str* [Scheme Procedure]
`scm_lstat` (*str*) [C Function]
Similar to `stat`, but does not follow symbolic links, i.e., it will return information about a symbolic link itself, not the file it points to. *path* must be a string.

`readlink` *path* [Scheme Procedure]
`scm_readlink` (*path*) [C Function]
Return the value of the symbolic link named by *path* (a string), i.e., the file that the link points to.

`chown` *object owner group* [Scheme Procedure]
`scm_chown` (*object, owner, group*) [C Function]
Change the ownership and group of the file referred to by *object* to the integer values *owner* and *group*. *object* can be a string containing a file name or, if the platform supports `fchown` (see section "File Owner" in *The GNU C Library Reference Manual*), a port or integer file descriptor which is open on the file. The return value is unspecified.

If *object* is a symbolic link, either the ownership of the link or the ownership of the referenced file will be changed depending on the operating system (lchown is unsupported at present). If *owner* or *group* is specified as `-1`, then that ID is not changed.

`chmod` *object mode* [Scheme Procedure]
`scm_chmod` (*object, mode*) [C Function]

Changes the permissions of the file referred to by *obj*. *obj* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fchmod` is used as the underlying system call). *mode* specifies the new permissions as a decimal number, e.g., `(chmod "foo" #o755)`. The return value is unspecified.

`utime` *pathname* [*actime* [*modtime*]] [Scheme Procedure]
`scm_utime` (*pathname, actime, modtime*) [C Function]

`utime` sets the access and modification times for the file named by *path*. If *actime* or *modtime* is not supplied, then the current time is used. *actime* and *modtime* must be integer time values as returned by the `current-time` procedure.

```
(utime "foo" (- (current-time) 3600))
```

will set the access time to one hour in the past and the modification time to the current time.

`delete-file` *str* [Scheme Procedure]
`scm_delete_file` (*str*) [C Function]

Deletes (or "unlinks") the file whose path is specified by *str*.

`copy-file` *oldfile newfile* [Scheme Procedure]
`scm_copy_file` (*oldfile, newfile*) [C Function]

Copy the file specified by *oldfile* to *newfile*. The return value is unspecified.

`rename-file` *oldname newname* [Scheme Procedure]
`scm_rename` (*oldname, newname*) [C Function]

Renames the file specified by *oldname* to *newname*. The return value is unspecified.

`link` *oldpath newpath* [Scheme Procedure]
`scm_link` (*oldpath, newpath*) [C Function]

Creates a new name *newpath* in the file system for the file named by *oldpath*. If *oldpath* is a symbolic link, the link may or may not be followed depending on the system.

`symlink` *oldpath newpath* [Scheme Procedure]
`scm_symlink` (*oldpath, newpath*) [C Function]

Create a symbolic link named *newpath* with the value (i.e., pointing to) *oldpath*. The return value is unspecified.

`mkdir` *path* [*mode*] [Scheme Procedure]
`scm_mkdir` (*path, mode*) [C Function]

Create a new directory named by *path*. If *mode* is omitted then the permissions of the directory file are set using the current umask (see Section 6.2.7 [Processes], page 393). Otherwise they are set to the decimal value specified with *mode*. The return value is unspecified.

`rmdir` *path* [Scheme Procedure]
`scm_rmdir` (*path*) [C Function]

Remove the existing directory named by *path*. The directory must be empty for this to succeed. The return value is unspecified.

`opendir` *dirname* [Scheme Procedure]
`scm_opendir` (*dirname*) [C Function]

Open the directory specified by *dirname* and return a directory stream.

`directory-stream?` *object* [Scheme Procedure]
`scm_directory_stream_p` (*object*) [C Function]

Return a boolean indicating whether *object* is a directory stream as returned by `opendir`.

`readdir` *stream* [Scheme Procedure]
`scm_readdir` (*stream*) [C Function]

Return (as a string) the next directory entry from the directory stream *stream*. If there is no remaining entry to be read then the end of file object is returned.

`rewinddir` *stream* [Scheme Procedure]
`scm_rewinddir` (*stream*) [C Function]

Reset the directory port *stream* so that the next call to `readdir` will return the first directory entry.

`closedir` *stream* [Scheme Procedure]
`scm_closedir` (*stream*) [C Function]

Close the directory stream *stream*. The return value is unspecified.

Here is an example showing how to display all the entries in a directory:

```
(define dir (opendir "/usr/lib"))
(do ((entry (readdir dir) (readdir dir)))
    ((eof-object? entry))
  (display entry)(newline))
(closedir dir)
```

`sync` [Scheme Procedure]
`scm_sync` () [C Function]

Flush the operating system disk buffers. The return value is unspecified.

`mknod` *path type perms dev* [Scheme Procedure]
`scm_mknod` (*path, type, perms, dev*) [C Function]

Creates a new special file, such as a file corresponding to a device. *path* specifies the name of the file. *type* should be one of the following symbols: ‘`regular`’, ‘`directory`’, ‘`symlink`’, ‘`block-special`’, ‘`char-special`’, ‘`fifo`’, or ‘`socket`’. *perms* (an integer) specifies the file permissions. *dev* (an integer) specifies which device the special file refers to. Its exact interpretation depends on the kind of special file being created.

E.g.,

```
(mknod "/dev/fd0" 'block-special #o660 (+ (* 2 256) 2))
```

The return value is unspecified.

`tmpnam` [Scheme Procedure]
`scm_tmpnam` () [C Function]

Return an auto-generated name of a temporary file, a file which doesn’t already exist. The name includes a path, it’s usually in ‘`/tmp`’ but that’s system dependent.

Care must be taken when using `tmpnam`. In between choosing the name and creating the file another program might use that name, or an attacker might even make it a symlink pointing at something important and causing you to overwrite that.

The safe way is to create the file using `open` with `O_EXCL` to avoid any overwriting. A loop can try again with another name if the file exists (error `EEXIST`). `mkstemp!` below does that.

`mkstemp!` *tmpl* [Scheme Procedure]
`scm_mkstemp` (*tmpl*) [C Function]

Create a new unique file in the file system and return a new buffered port open for reading and writing to the file.

tmpl is a string specifying where the file should be created: it must end with '`XXXXXX`' and those '`X`'s will be changed in the string to return the name of the file. (`port-filename` on the port also gives the name.)

POSIX doesn't specify the permissions mode of the file, on GNU and most systems it's `#o600`. An application can use `chmod` to relax that if desired. For example `#o666` less `umask`, which is usual for ordinary file creation,

```
(let ((port (mkstemp! (string-copy "/tmp/myfile-XXXXXX"))))
  (chmod port (logand #o666 (lognot (umask))))
  ...)
```

`dirname` *filename* [Scheme Procedure]
`scm_dirname` (*filename*) [C Function]

Return the directory name component of the file name *filename*. If *filename* does not contain a directory component, `.` is returned.

`basename` *filename* [*suffix*] [Scheme Procedure]
`scm_basename` (*filename, suffix*) [C Function]

Return the base name of the file name *filename*. The base name is the file name without any directory components. If *suffix* is provided, and is equal to the end of *basename*, it is removed also.

```
(basename "/tmp/test.xml" ".xml")
⇒ "test"
```

6.2.4 User Information

The facilities in this section provide an interface to the user and group database. They should be used with care since they are not reentrant.

The following functions accept an object representing user information and return a selected component:

`passwd:name` *pw* [Scheme Procedure]

The name of the userid.

`passwd:passwd` *pw* [Scheme Procedure]

The encrypted passwd.

`passwd:uid` *pw* [Scheme Procedure]

The user id number.

`passwd:gid` *pw* [Scheme Procedure]
: The group id number.

`passwd:gecos` *pw* [Scheme Procedure]
: The full name.

`passwd:dir` *pw* [Scheme Procedure]
: The home directory.

`passwd:shell` *pw* [Scheme Procedure]
: The login shell.

`getpwuid` *uid* [Scheme Procedure]
: Look up an integer userid in the user database.

`getpwnam` *name* [Scheme Procedure]
: Look up a user name string in the user database.

`setpwent` [Scheme Procedure]
: Initializes a stream used by `getpwent` to read from the user database. The next use of `getpwent` will return the first entry. The return value is unspecified.

`getpwent` [Scheme Procedure]
: Read the next entry in the user database stream. The return is a passwd user object as above, or `#f` when no more entries.

`endpwent` [Scheme Procedure]
: Closes the stream used by `getpwent`. The return value is unspecified.

`setpw` [*arg*] [Scheme Procedure]
`scm_setpwent` (*arg*) [C Function]
: If called with a true argument, initialize or reset the password data stream. Otherwise, close the stream. The `setpwent` and `endpwent` procedures are implemented on top of this.

`getpw` [*user*] [Scheme Procedure]
`scm_getpwuid` (*user*) [C Function]
: Look up an entry in the user database. *obj* can be an integer, a string, or omitted, giving the behaviour of getpwuid, getpwnam or getpwent respectively.

The following functions accept an object representing group information and return a selected component:

`group:name` *gr* [Scheme Procedure]
: The group name.

`group:passwd` *gr* [Scheme Procedure]
: The encrypted group password.

`group:gid` *gr* [Scheme Procedure]
: The group id number.

`group:mem` *gr* [Scheme Procedure]

A list of userids which have this group as a supplementary group.

`getgrgid` *gid* [Scheme Procedure]

Look up an integer group id in the group database.

`getgrnam` *name* [Scheme Procedure]

Look up a group name in the group database.

`setgrent` [Scheme Procedure]

Initializes a stream used by `getgrent` to read from the group database. The next use of `getgrent` will return the first entry. The return value is unspecified.

`getgrent` [Scheme Procedure]

Return the next entry in the group database, using the stream set by `setgrent`.

`endgrent` [Scheme Procedure]

Closes the stream used by `getgrent`. The return value is unspecified.

`setgr` [*arg*] [Scheme Procedure]
`scm_setgrent` (*arg*) [C Function]

If called with a true argument, initialize or reset the group data stream. Otherwise, close the stream. The `setgrent` and `endgrent` procedures are implemented on top of this.

`getgr` [*name*] [Scheme Procedure]
`scm_getgrgid` (*name*) [C Function]

Look up an entry in the group database. *obj* can be an integer, a string, or omitted, giving the behaviour of getgrgid, getgrnam or getgrent respectively.

In addition to the accessor procedures for the user database, the following shortcut procedures are also available.

`cuserid` [Scheme Procedure]
`scm_cuserid` () [C Function]

Return a string containing a user name associated with the effective user id of the process. Return `#f` if this information cannot be obtained.

This function has been removed from the latest POSIX specification, Guile provides it only if the system has it. Using `(getpwuid (geteuid))` may be a better idea.

`getlogin` [Scheme Procedure]
`scm_getlogin` () [C Function]

Return a string containing the name of the user logged in on the controlling terminal of the process, or `#f` if this information cannot be obtained.

6.2.5 Time

`current-time` [Scheme Procedure]
`scm_current_time ()` [C Function]
: Return the number of seconds since 1970-01-01 00:00:00 UTC, excluding leap seconds.

`gettimeofday` [Scheme Procedure]
`scm_gettimeofday ()` [C Function]
: Return a pair containing the number of seconds and microseconds since 1970-01-01 00:00:00 UTC, excluding leap seconds. Note: whether true microsecond resolution is available depends on the operating system.

The following procedures either accept an object representing a broken down time and return a selected component, or accept an object representing a broken down time and a value and set the component to the value. The numbers in parentheses give the usual range.

`tm:sec` *tm* [Scheme Procedure]
`set-tm:sec` *tm val* [Scheme Procedure]
: Seconds (0-59).

`tm:min` *tm* [Scheme Procedure]
`set-tm:min` *tm val* [Scheme Procedure]
: Minutes (0-59).

`tm:hour` *tm* [Scheme Procedure]
`set-tm:hour` *tm val* [Scheme Procedure]
: Hours (0-23).

`tm:mday` *tm* [Scheme Procedure]
`set-tm:mday` *tm val* [Scheme Procedure]
: Day of the month (1-31).

`tm:mon` *tm* [Scheme Procedure]
`set-tm:mon` *tm val* [Scheme Procedure]
: Month (0-11).

`tm:year` *tm* [Scheme Procedure]
`set-tm:year` *tm val* [Scheme Procedure]
: Year (70-), the year minus 1900.

`tm:wday` *tm* [Scheme Procedure]
`set-tm:wday` *tm val* [Scheme Procedure]
: Day of the week (0-6) with Sunday represented as 0.

`tm:yday` *tm* [Scheme Procedure]
`set-tm:yday` *tm val* [Scheme Procedure]
: Day of the year (0-364, 365 in leap years).

`tm:isdst` *tm* [Scheme Procedure]
`set-tm:isdst` *tm val* [Scheme Procedure]
: Daylight saving indicator (0 for "no", greater than 0 for "yes", less than 0 for "unknown").

`tm:gmtoff` *tm* [Scheme Procedure]
`set-tm:gmtoff` *tm val* [Scheme Procedure]

Time zone offset in seconds west of UTC (-46800 to 43200). For example on East coast USA (zone '`EST+5`') this would be 18000 (ie. $5 \times 60 \times 60$) in winter, or 14400 (ie. $4 \times 60 \times 60$) during daylight savings.

Note `tm:gmtoff` is not the same as `tm_gmtoff` in the C `tm` structure. `tm_gmtoff` is seconds east and hence the negative of the value here.

`tm:zone` *tm* [Scheme Procedure]
`set-tm:zone` *tm val* [Scheme Procedure]

Time zone label (a string), not necessarily unique.

`localtime` *time* [*zone*] [Scheme Procedure]
`scm_localtime` (*time*, *zone*) [C Function]

Return an object representing the broken down components of *time*, an integer like the one returned by `current-time`. The time zone for the calculation is optionally specified by *zone* (a string), otherwise the `TZ` environment variable or the system default is used.

`gmtime` *time* [Scheme Procedure]
`scm_gmtime` (*time*) [C Function]

Return an object representing the broken down components of *time*, an integer like the one returned by `current-time`. The values are calculated for UTC.

`mktime` *sbd-time* [*zone*] [Scheme Procedure]
`scm_mktime` (*sbd_time*, *zone*) [C Function]

For a broken down time object *sbd-time*, return a pair the `car` of which is an integer time like `current-time`, and the `cdr` of which is a new broken down time with normalized fields.

zone is a timezone string, or the default is the `TZ` environment variable or the system default (see section "Specifying the Time Zone with `TZ`" in *GNU C Library Reference Manual*). *sbd-time* is taken to be in that *zone*.

The following fields of *sbd-time* are used: `tm:year`, `tm:mon`, `tm:mday`, `tm:hour`, `tm:min`, `tm:sec`, `tm:isdst`. The values can be outside their usual ranges. For example `tm:hour` normally goes up to 23, but a value say 33 would mean 9 the following day.

`tm:isdst` in *sbd-time* says whether the time given is with daylight savings or not. This is ignored if *zone* doesn't have any daylight savings adjustment amount.

The broken down time in the return normalizes the values of *sbd-time* by bringing them into their usual ranges, and using the actual daylight savings rule for that time in *zone* (which may differ from what *sbd-time* had). The easiest way to think of this is that *sbd-time* plus *zone* converts to the integer UTC time, then a `localtime` is applied to get the normal presentation of that time, in *zone*.

`tzset` [Scheme Procedure]

`scm_tzset` () [C Function]

Initialize the timezone from the `TZ` environment variable or the system default. It's not usually necessary to call this procedure since it's done automatically by other procedures that depend on the timezone.

`strftime` *format tm* [Scheme Procedure]
`scm_strftime` (*format, tm*) [C Function]

Return a string which is broken-down time structure *tm* formatted according to the given *format* string.

format contains field specifications introduced by a '`%`' character. See section "Formatting Calendar Time" in *The GNU C Library Reference Manual*, or '`man 3 strftime`', for the available formatting.

```
(strftime "%c" (localtime (current-time)))
⇒ "Mon Mar 11 20:17:43 2002"
```

If `setlocale` has been called (see Section 6.2.13 [Locales], page 417), month and day names are from the current locale and in the locale character set.

Note that '`%Z`' might print the `tm:zone` in *tm* or it might print just the current zone (`tzset` above). A GNU system prints `tm:zone`, a strict C99 system like NetBSD prints the current zone. Perhaps in the future Guile will try to get `tm:zone` used always.

`strptime` *format string* [Scheme Procedure]
`scm_strptime` (*format, string*) [C Function]

Performs the reverse action to `strftime`, parsing *string* according to the specification supplied in *template*. The interpretation of month and day names is dependent on the current locale. The value returned is a pair. The CAR has an object with time components in the form returned by `localtime` or `gmtime`, but the time zone components are not usefully set. The CDR reports the number of characters from *string* which were used for the conversion.

`internal-time-units-per-second` [Variable]

The value of this variable is the number of time units per second reported by the following procedures.

`times` [Scheme Procedure]
`scm_times` () [C Function]

Return an object with information about real and processor time. The following procedures accept such an object as an argument and return a selected component:

`tms:clock` *tms* [Scheme Procedure]

The current real time, expressed as time units relative to an arbitrary base.

`tms:utime` *tms* [Scheme Procedure]

The CPU time units used by the calling process.

`tms:stime` *tms* [Scheme Procedure]

The CPU time units used by the system on behalf of the calling process.

`tms:cutime` *tms* [Scheme Procedure]
: The CPU time units used by terminated child processes of the calling process, whose status has been collected (e.g., using `waitpid`).

`tms:cstime` *tms* [Scheme Procedure]
: Similarly, the CPU times units used by the system on behalf of terminated child processes.

`get-internal-real-time` [Scheme Procedure]
`scm_get_internal_real_time ()` [C Function]
: Return the number of time units since the interpreter was started.

`get-internal-run-time` [Scheme Procedure]
`scm_get_internal_run_time ()` [C Function]
: Return the number of time units of processor time used by the interpreter. Both *system* and *user* time are included but subprocesses are not.

6.2.6 Runtime Environment

`program-arguments` [Scheme Procedure]
`command-line` [Scheme Procedure]
`scm_program_arguments ()` [C Function]
: Return the list of command line arguments passed to Guile, as a list of strings. The list includes the invoked program name, which is usually `"guile"`, but excludes switches and parameters for command line options like `-e` and `-l`.

`getenv` *nam* [Scheme Procedure]
`scm_getenv` (*nam*) [C Function]
: Looks up the string *name* in the current environment. The return value is `#f` unless a string of the form `NAME=VALUE` is found, in which case the string `VALUE` is returned.

`setenv` *name value* [Scheme Procedure]
: Modifies the environment of the current process, which is also the default environment inherited by child processes.

If *value* is `#f`, then *name* is removed from the environment. Otherwise, the string *name=value* is added to the environment, replacing any existing string with name matching *name*.

The return value is unspecified.

`unsetenv` *name* [Scheme Procedure]
: Remove variable *name* from the environment. The name can not contain a '`=`' character.

`environ` [*env*] [Scheme Procedure]
`scm_environ` (*env*) [C Function]
: If *env* is omitted, return the current environment (in the Unix sense) as a list of strings. Otherwise set the current environment, which is also the default environment for child processes, to the supplied list of strings. Each member of *env* should be of the form *NAME=VALUE* and values of *NAME* should not be duplicated. If *env* is supplied then the return value is unspecified.

`putenv` *str* [Scheme Procedure]
`scm_putenv` (*str*) [C Function]

Modifies the environment of the current process, which is also the default environment inherited by child processes.

If *string* is of the form `NAME=VALUE` then it will be written directly into the environment, replacing any existing environment string with name matching `NAME`. If *string* does not contain an equal sign, then any existing string with name matching *string* will be removed.

The return value is unspecified.

6.2.7 Processes

`chdir` *str* [Scheme Procedure]
`scm_chdir` (*str*) [C Function]

Change the current working directory to *path*. The return value is unspecified.

`getcwd` [Scheme Procedure]
`scm_getcwd` () [C Function]

Return the name of the current working directory.

`umask` [*mode*] [Scheme Procedure]
`scm_umask` (*mode*) [C Function]

If *mode* is omitted, returns a decimal number representing the current file creation mask. Otherwise the file creation mask is set to *mode* and the previous value is returned. See section "Assigning File Permissions" in *The GNU C Library Reference Manual*, for more on how to use umasks.

E.g., `(umask #o022)` sets the mask to octal 22/decimal 18.

`chroot` *path* [Scheme Procedure]
`scm_chroot` (*path*) [C Function]

Change the root directory to that specified in *path*. This directory will be used for path names beginning with '`/`'. The root directory is inherited by all children of the current process. Only the superuser may change the root directory.

`getpid` [Scheme Procedure]
`scm_getpid` () [C Function]

Return an integer representing the current process ID.

`getgroups` [Scheme Procedure]
`scm_getgroups` () [C Function]

Return a vector of integers representing the current supplementary group IDs.

`getppid` [Scheme Procedure]
`scm_getppid` () [C Function]

Return an integer representing the process ID of the parent process.

`getuid` [Scheme Procedure]
`scm_getuid` () [C Function]

Return an integer representing the current real user ID.

`getgid` [Scheme Procedure]
`scm_getgid` () [C Function]
Return an integer representing the current real group ID.

`geteuid` [Scheme Procedure]
`scm_geteuid` () [C Function]
Return an integer representing the current effective user ID. If the system does not support effective IDs, then the real ID is returned. `(provided? 'EIDs)` reports whether the system supports effective IDs.

`getegid` [Scheme Procedure]
`scm_getegid` () [C Function]
Return an integer representing the current effective group ID. If the system does not support effective IDs, then the real ID is returned. `(provided? 'EIDs)` reports whether the system supports effective IDs.

`setgroups` *vec* [Scheme Procedure]
`scm_setgroups` (*vec*) [C Function]
Set the current set of supplementary group IDs to the integers in the given vector *vec*. The return value is unspecified.

Generally only the superuser can set the process group IDs (see section "Setting Groups" in *The GNU C Library Reference Manual*).

`setuid` *id* [Scheme Procedure]
`scm_setuid` (*id*) [C Function]
Sets both the real and effective user IDs to the integer *id*, provided the process has appropriate privileges. The return value is unspecified.

`setgid` *id* [Scheme Procedure]
`scm_setgid` (*id*) [C Function]
Sets both the real and effective group IDs to the integer *id*, provided the process has appropriate privileges. The return value is unspecified.

`seteuid` *id* [Scheme Procedure]
`scm_seteuid` (*id*) [C Function]
Sets the effective user ID to the integer *id*, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—`(provided? 'EIDs)` reports whether the system supports effective IDs. The return value is unspecified.

`setegid` *id* [Scheme Procedure]
`scm_setegid` (*id*) [C Function]
Sets the effective group ID to the integer *id*, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—`(provided? 'EIDs)` reports whether the system supports effective IDs. The return value is unspecified.

`getpgrp` [Scheme Procedure]
`scm_getpgrp` () [C Function]
Return an integer representing the current process group ID. This is the POSIX definition, not BSD.

`setpgid` *pid pgid* [Scheme Procedure]
`scm_setpgid` (*pid, pgid*) [C Function]

Move the process *pid* into the process group *pgid*. *pid* or *pgid* must be integers: they can be zero to indicate the ID of the current process. Fails on systems that do not support job control. The return value is unspecified.

`setsid` [Scheme Procedure]
`scm_setsid` () [C Function]

Creates a new session. The current process becomes the session leader and is put in a new process group. The process will be detached from its controlling terminal if it has one. The return value is an integer representing the new process group ID.

`waitpid` *pid* [*options*] [Scheme Procedure]
`scm_waitpid` (*pid, options*) [C Function]

This procedure collects status information from a child process which has terminated or (optionally) stopped. Normally it will suspend the calling process until this can be done. If more than one child process is eligible then one will be chosen by the operating system.

The value of *pid* determines the behaviour:

pid greater than 0
Request status information from the specified child process.

pid equal to -1 or `WAIT_ANY`
Request status information for any child process.

pid equal to 0 or `WAIT_MYPGRP`
Request status information for any child process in the current process group.

pid less than -1
Request status information for any child process whose process group ID is $-pid$.

The *options* argument, if supplied, should be the bitwise OR of the values of zero or more of the following variables:

`WNOHANG` [Variable]

Return immediately even if there are no child processes to be collected.

`WUNTRACED` [Variable]

Report status information for stopped processes as well as terminated processes.

The return value is a pair containing:

1. The process ID of the child process, or 0 if `WNOHANG` was specified and no process was collected.
2. The integer status value.

The following three functions can be used to decode the process status code returned by `waitpid`.

`status:exit-val` *status* [Scheme Procedure]
`scm_status_exit_val` (*status*) [C Function]

Return the exit status value, as would be set if a process ended normally through a call to `exit` or `_exit`, if any, otherwise `#f`.

`status:term-sig` *status* [Scheme Procedure]
`scm_status_term_sig` (*status*) [C Function]

Return the signal number which terminated the process, if any, otherwise `#f`.

`status:stop-sig` *status* [Scheme Procedure]
`scm_status_stop_sig` (*status*) [C Function]

Return the signal number which stopped the process, if any, otherwise `#f`.

`system` [*cmd*] [Scheme Procedure]
`scm_system` (*cmd*) [C Function]

Execute *cmd* using the operating system's "command processor". Under Unix this is usually the default shell `sh`. The value returned is *cmd*'s exit status as returned by `waitpid`, which can be interpreted using the functions above.

If `system` is called without arguments, return a boolean indicating whether the command processor is available.

`system*` . *args* [Scheme Procedure]
`scm_system_star` (*args*) [C Function]

Execute the command indicated by *args*. The first element must be a string indicating the command to be executed, and the remaining items must be strings representing each of the arguments to that command.

This function returns the exit status of the command as provided by `waitpid`. This value can be handled with `status:exit-val` and the related functions.

`system*` is similar to `system`, but accepts only one string per-argument, and performs no shell interpretation. The command is executed using fork and execlp. Accordingly this function may be safer than `system` in situations where shell interpretation is not required.

Example: (system* "echo" "foo" "bar")

`primitive-exit` [*status*] [Scheme Procedure]
`primitive-_exit` [*status*] [Scheme Procedure]
`scm_primitive_exit` (*status*) [C Function]
`scm_primitive__exit` (*status*) [C Function]

Terminate the current process without unwinding the Scheme stack. The exit status is *status* if supplied, otherwise zero.

`primitive-exit` uses the C `exit` function and hence runs usual C level cleanups (flush output streams, call `atexit` functions, etc, see section "Normal Termination" in *The GNU C Library Reference Manual*)).

`primitive-_exit` is the `_exit` system call (see section "Termination Internals" in *The GNU C Library Reference Manual*). This terminates the program immediately, with neither Scheme-level nor C-level cleanups.

The typical use for `primitive-_exit` is from a child process created with `primitive-fork`. For example in a Gdk program the child process inherits the X server connection and a C-level `atexit` cleanup which will close that connection. But closing in the child would upset the protocol in the parent, so `primitive-_exit` should be used to exit without that.

`execl` *filename . args* [Scheme Procedure]
`scm_execl` (*filename, args*) [C Function]

Executes the file named by *path* as a new process image. The remaining arguments are supplied to the process; from a C program they are accessible as the `argv` argument to `main`. Conventionally the first *arg* is the same as *path*. All arguments must be strings.

If *arg* is missing, *path* is executed with a null argument list, which may have system-dependent side-effects.

This procedure is currently implemented using the `execv` system call, but we call it `execl` because of its Scheme calling interface.

`execlp` *filename . args* [Scheme Procedure]
`scm_execlp` (*filename, args*) [C Function]

Similar to `execl`, however if *filename* does not contain a slash then the file to execute will be located by searching the directories listed in the `PATH` environment variable.

This procedure is currently implemented using the `execvp` system call, but we call it `execlp` because of its Scheme calling interface.

`execle` *filename env . args* [Scheme Procedure]
`scm_execle` (*filename, env, args*) [C Function]

Similar to `execl`, but the environment of the new process is specified by *env*, which must be a list of strings as returned by the `environ` procedure.

This procedure is currently implemented using the `execve` system call, but we call it `execle` because of its Scheme calling interface.

`primitive-fork` [Scheme Procedure]
`scm_fork` () [C Function]

Creates a new "child" process by duplicating the current "parent" process. In the child the return value is 0. In the parent the return value is the integer process ID of the child.

This procedure has been renamed from `fork` to avoid a naming conflict with the scsh fork.

`nice` *incr* [Scheme Procedure]
`scm_nice` (*incr*) [C Function]

Increment the priority of the current process by *incr*. A higher priority value means that the process runs less often. The return value is unspecified.

`setpriority` *which who prio* [Scheme Procedure]
`scm_setpriority` (*which, who, prio*) [C Function]

Set the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the variables `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user identifier for `PRIO_USER`. A zero value of *who* denotes the current process, process group, or user. *prio* is a value in the range [−20,20]. The default priority is 0; lower priorities (in numerical terms) cause more favorable scheduling. Sets the priority of all of the specified processes. Only the super-user may lower priorities. The return value is not specified.

`getpriority` *which who* [Scheme Procedure]
`scm_getpriority` (*which, who*) [C Function]

Return the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the variables `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, and *who* should be interpreted depending on *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user identifier for `PRIO_USER`). A zero value of *who* denotes the current process, process group, or user. Return the highest priority (lowest numerical value) of any of the specified processes.

6.2.8 Signals

Procedures to raise, handle and wait for signals.

`kill` *pid sig* [Scheme Procedure]
`scm_kill` (*pid, sig*) [C Function]

Sends a signal to the specified process or group of processes.

pid specifies the processes to which the signal is sent:

pid greater than 0
: The process whose identifier is *pid*.

pid equal to 0
: All processes in the current process group.

pid less than -1
: The process group whose identifier is -*pid*

pid equal to -1
: If the process is privileged, all processes except for some special system processes. Otherwise, all processes with the current effective user ID.

sig should be specified using a variable corresponding to the Unix symbolic name, e.g.,

`SIGHUP` [Variable]

Hang-up signal.

`SIGINT` [Variable]

Interrupt signal.

A full list of signals on the GNU system may be found in section “Standard Signals” in *The GNU C Library Reference Manual*.

`raise` *sig* [Scheme Procedure]
`scm_raise` (*sig*) [C Function]

Sends a specified signal *sig* to the current process, where *sig* is as described for the `kill` procedure.

`sigaction` *signum* [*handler* [*flags* [*thread*]]] [Scheme Procedure]
`scm_sigaction` (*signum, handler, flags*) [C Function]
`scm_sigaction_for_thread` (*signum, handler, flags, thread*) [C Function]

Install or report the signal handler for a specified signal.

signum is the signal number, which can be specified using the value of variables such as `SIGINT`.

If *handler* is omitted, `sigaction` returns a pair: the CAR is the current signal hander, which will be either an integer with the value `SIG_DFL` (default action) or `SIG_IGN` (ignore), or the Scheme procedure which handles the signal, or `#f` if a non-Scheme procedure handles the signal. The CDR contains the current `sigaction` flags for the handler.

If *handler* is provided, it is installed as the new handler for *signum*. *handler* can be a Scheme procedure taking one argument, or the value of `SIG_DFL` (default action) or `SIG_IGN` (ignore), or `#f` to restore whatever signal handler was installed before `sigaction` was first used. When a scheme procedure has been specified, that procedure will run in the given *thread*. When no thread has been given, the thread that made this call to `sigaction` is used.

flags is a `logior` (see Section 5.5.2.14 [Bitwise Operations], page 117) of the following (where provided by the system), or `0` for none.

`SA_NOCLDSTOP` [Variable]

By default, `SIGCHLD` is signalled when a child process stops (ie. receives `SIGSTOP`), and when a child process terminates. With the `SA_NOCLDSTOP` flag, `SIGCHLD` is only signalled for termination, not stopping.

`SA_NOCLDSTOP` has no effect on signals other than `SIGCHLD`.

`SA_RESTART` [Variable]

If a signal occurs while in a system call, deliver the signal then restart the system call (as opposed to returning an `EINTR` error from that call).

Guile always enables this flag where available, no matter what *flags* are specified. This avoids spurious error returns in low level operations.

The return value is a pair with information about the old handler as described above.

This interface does not provide access to the "signal blocking" facility. Maybe this is not needed, since the thread support may provide solutions to the problem of consistent access to data structures.

`restore-signals` [Scheme Procedure]
`scm_restore_signals` () [C Function]

Return all signal handlers to the values they had before any call to `sigaction` was made. The return value is unspecified.

`alarm` *i* [Scheme Procedure]
`scm_alarm` (*i*) [C Function]

Set a timer to raise a `SIGALRM` signal after the specified number of seconds (an integer). It's advisable to install a signal handler for `SIGALRM` beforehand, since the default action is to terminate the process.

The return value indicates the time remaining for the previous alarm, if any. The new value replaces the previous alarm. If there was no previous alarm, the return value is zero.

`pause` [Scheme Procedure]
`scm_pause` () [C Function]
Pause the current process (thread?) until a signal arrives whose action is to either terminate the current process or invoke a handler procedure. The return value is unspecified.

`sleep` *i* [Scheme Procedure]
`scm_sleep` (*i*) [C Function]
Wait for the given number of seconds (an integer) or until a signal arrives. The return value is zero if the time elapses or the number of seconds remaining otherwise.

`usleep` *i* [Scheme Procedure]
`scm_usleep` (*i*) [C Function]
Sleep for *i* microseconds. `usleep` is not available on all platforms. [FIXME: so what happens when it isn't?]

`setitimer` *which_timer interval_seconds interval_microseconds value_seconds value_microseconds* [Scheme Procedure]
`scm_setitimer` (*which_timer, interval_seconds, interval_microseconds, value_seconds, value_microseconds*) [C Function]
Set the timer specified by *which_timer* according to the given *interval_seconds*, *interval_microseconds*, *value_seconds*, and *value_microseconds* values.

Return information about the timer's previous setting.

The timers available are: `ITIMER_REAL`, `ITIMER_VIRTUAL`, and `ITIMER_PROF`.

The return value will be a list of two cons pairs representing the current state of the given timer. The first pair is the seconds and microseconds of the timer `it_interval`, and the second pair is the seconds and microseconds of the timer `it_value`.

`getitimer` *which_timer* [Scheme Procedure]
`scm_getitimer` (*which_timer*) [C Function]
Return information about the timer specified by *which_timer*.

The timers available are: `ITIMER_REAL`, `ITIMER_VIRTUAL`, and `ITIMER_PROF`.

The return value will be a list of two cons pairs representing the current state of the given timer. The first pair is the seconds and microseconds of the timer `it_interval`, and the second pair is the seconds and microseconds of the timer `it_value`.

6.2.9 Terminals and Ptys

`isatty?` *port* [Scheme Procedure]
`scm_isatty_p` (*port*) [C Function]
Return `#t` if *port* is using a serial non–file device, otherwise `#f`.

`ttyname` *port* [Scheme Procedure]
`scm_ttyname` (*port*) [C Function]
Return a string with the name of the serial terminal device underlying *port*.

`ctermid` [Scheme Procedure]
`scm_ctermid` () [C Function]

Return a string containing the file name of the controlling terminal for the current process.

`tcgetpgrp` *port* [Scheme Procedure]
`scm_tcgetpgrp` (*port*) [C Function]

Return the process group ID of the foreground process group associated with the terminal open on the file descriptor underlying *port*.

If there is no foreground process group, the return value is a number greater than 1 that does not match the process group ID of any existing process group. This can happen if all of the processes in the job that was formerly the foreground job have terminated, and no other job has yet been moved into the foreground.

`tcsetpgrp` *port pgid* [Scheme Procedure]
`scm_tcsetpgrp` (*port, pgid*) [C Function]

Set the foreground process group ID for the terminal used by the file descriptor underlying *port* to the integer *pgid*. The calling process must be a member of the same session as *pgid* and must have the same controlling terminal. The return value is unspecified.

6.2.10 Pipes

The following procedures are similar to the `popen` and `pclose` system routines. The code is in a separate "popen" module:

```
(use-modules (ice-9 popen))
```

`open-pipe` *command mode* [Scheme Procedure]
`open-pipe*` *mode prog* [*args...*] [Scheme Procedure]

Execute a command in a subprocess, with a pipe to it or from it, or with pipes in both directions.

`open-pipe` runs the shell *command* using '`/bin/sh -c`'. `open-pipe*` executes *prog* directly, with the optional *args* arguments (all strings).

mode should be one of the following values. `OPEN_READ` is an input pipe, ie. to read from the subprocess. `OPEN_WRITE` is an output pipe, ie. to write to it.

`OPEN_READ` [Variable]
`OPEN_WRITE` [Variable]
`OPEN_BOTH` [Variable]

For an input pipe, the child's standard output is the pipe and standard input is inherited from `current-input-port`. For an output pipe, the child's standard input is the pipe and standard output is inherited from `current-output-port`. In all cases cases the child's standard error is inherited from `current-error-port` (see Section 5.12.8 [Default Ports], page 278).

If those `current-X-ports` are not files of some kind, and hence don't have file descriptors for the child, then '`/dev/null`' is used instead.

Care should be taken with `OPEN_BOTH`, a deadlock will occur if both parent and child are writing, and waiting until the write completes before doing any reading.

Each direction has `PIPE_BUF` bytes of buffering (see Section 6.2.2 [Ports and File Descriptors], page 374), which will be enough for small writes, but not for say putting a big file through a filter.

`open-input-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_READ`.

```
(let* ((port (open-input-pipe "date --utc"))
       (str  (read-line port)))
  (close-pipe port)
  str)
⇒ "Mon Mar 11 20:10:44 UTC 2002"
```

`open-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_WRITE`.

```
(let ((port (open-output-pipe "lpr")))
  (display "Something for the line printer.\n" port)
  (if (not (eqv? 0 (status:exit-val (close-pipe port))))
      (error "Cannot print")))
```

`open-input-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_BOTH`.

`close-pipe` *port* [Scheme Procedure]

Close a pipe created by `open-pipe`, wait for the process to terminate, and return the wait status code. The status is as per `waitpid` and can be decoded with `status:exit-val` etc (see Section 6.2.7 [Processes], page 393)

`waitpid WAIT_ANY` should not be used when pipes are open, since it can reap a pipe's child process, causing an error from a subsequent `close-pipe`.

`close-port` (see Section 5.12.4 [Closing], page 275) can close a pipe, but it doesn't reap the child process.

The garbage collector will close a pipe no longer in use, and reap the child process with `waitpid`. If the child hasn't yet terminated the garbage collector doesn't block, but instead checks again in the next GC.

Many systems have per-user and system-wide limits on the number of processes, and a system-wide limit on the number of pipes, so pipes should be closed explicitly when no longer needed, rather than letting the garbage collector pick them up at some later time.

6.2.11 Networking

6.2.11.1 Network Address Conversion

This section describes procedures which convert internet addresses between numeric and string formats.

IPv4 Address Conversion

An IPv4 Internet address is a 4-byte value, represented in Guile as an integer in host byte order, so that say "0.0.0.1" is 1, or "1.0.0.0" is 16777216.

Some underlying C functions use network byte order for addresses, Guile converts as necessary so that at the Scheme level its host byte order everywhere.

`INADDR_ANY` [Variable]

For a server, this can be used with `bind` (see Section 6.2.11.4 [Network Sockets and Communication], page 410) to allow connections from any interface on the machine.

`INADDR_BROADCAST` [Variable]

The broadcast address on the local network.

`INADDR_LOOPBACK` [Variable]

The address of the local host using the loopback device, ie. '`127.0.0.1`'.

`inet-aton` *address* [Scheme Procedure]
`scm_inet_aton` (*address*) [C Function]

Convert an IPv4 Internet address from printable string (dotted decimal notation) to an integer. E.g.,

```
(inet-aton "127.0.0.1") ⇒ 2130706433
```

`inet-ntoa` *inetid* [Scheme Procedure]
`scm_inet_ntoa` (*inetid*) [C Function]

Convert an IPv4 Internet address to a printable (dotted decimal notation) string. E.g.,

```
(inet-ntoa 2130706433) ⇒ "127.0.0.1"
```

`inet-netof` *address* [Scheme Procedure]
`scm_inet_netof` (*address*) [C Function]

Return the network number part of the given IPv4 Internet address. E.g.,

```
(inet-netof 2130706433) ⇒ 127
```

`inet-lnaof` *address* [Scheme Procedure]
`scm_lnaof` (*address*) [C Function]

Return the local-address-with-network part of the given IPv4 Internet address, using the obsolete class A/B/C system. E.g.,

```
(inet-lnaof 2130706433) ⇒ 1
```

`inet-makeaddr` *net lna* [Scheme Procedure]
`scm_inet_makeaddr` (*net, lna*) [C Function]

Make an IPv4 Internet address by combining the network number *net* with the local-address-within-network number *lna*. E.g.,

```
(inet-makeaddr 127 1) ⇒ 2130706433
```

IPv6 Address Conversion

An IPv6 Internet address is a 16-byte value, represented in Guile as an integer in host byte order, so that say "::1" is 1.

`inet-ntop` *family address* [Scheme Procedure]
`scm_inet_ntop` (*family, address*) [C Function]

Convert a network address from an integer to a printable string. *family* can be `AF_INET` or `AF_INET6`. E.g.,

```
(inet-ntop AF_INET 2130706433) ⇒ "127.0.0.1"
(inet-ntop AF_INET6 (- (expt 2 128) 1)) ⇒
ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
```

`inet-pton` *family address* [Scheme Procedure]
`scm_inet_pton` (*family, address*) [C Function]

Convert a string containing a printable network address to an integer address. *family* can be `AF_INET` or `AF_INET6`. E.g.,

```
(inet-pton AF_INET "127.0.0.1") ⇒ 2130706433
(inet-pton AF_INET6 "::1") ⇒ 1
```

6.2.11.2 Network Databases

This section describes procedures which query various network databases. Care should be taken when using the database routines since they are not reentrant.

The Host Database

A *host object* is a structure that represents what is known about a network host, and is the usual way of representing a system's network identity inside software.

The following functions accept a host object and return a selected component:

`hostent:name` *host* [Scheme Procedure]

The "official" hostname for *host*.

`hostent:aliases` *host* [Scheme Procedure]

A list of aliases for *host*.

`hostent:addrtype` *host* [Scheme Procedure]

The host address type, one of the `AF` constants, such as `AF_INET` or `AF_INET6`.

`hostent:length` *host* [Scheme Procedure]

The length of each address for *host*, in bytes.

`hostent:addr-list` *host* [Scheme Procedure]

The list of network addresses associated with *host*. For `AF_INET` these are integer IPv4 address (see Section 6.2.11.1 [Network Address Conversion], page 402).

The following procedures are used to search the host database:

`gethost` [*host*] [Scheme Procedure]
`gethostbyname` *hostname* [Scheme Procedure]

`gethostbyaddr` *address* [Scheme Procedure]
`scm_gethost` (*host*) [C Function]

Look up a host by name or address, returning a host object. The `gethost` procedure will accept either a string name or an integer address; if given no arguments, it behaves like `gethostent` (see below). If a name or address is supplied but the address can not be found, an error will be thrown to one of the keys: `host-not-found`, `try-again`, `no-recovery` or `no-data`, corresponding to the equivalent `h_error` values. Unusual conditions may result in errors thrown to the `system-error` or `misc_error` keys.

```
(gethost "www.gnu.org")
⇒ #("www.gnu.org" () 2 4 (3353880842))

(gethostbyname "www.emacs.org")
⇒ #("emacs.org" ("www.emacs.org") 2 4 (1073448978))
```

The following procedures may be used to step through the host database from beginning to end.

`sethostent` [*stayopen*] [Scheme Procedure]

Initialize an internal stream from which host objects may be read. This procedure must be called before any calls to `gethostent`, and may also be called afterward to reset the host entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `gethostbyname` or `gethostbyaddr` calls, possibly giving an efficiency gain.

`gethostent` [Scheme Procedure]

Return the next host object from the host database, or `#f` if there are no more hosts to be found (or an error has been encountered). This procedure may not be used before `sethostent` has been called.

`endhostent` [Scheme Procedure]

Close the stream used by `gethostent`. The return value is unspecified.

`sethost` [*stayopen*] [Scheme Procedure]
`scm_sethost` (*stayopen*) [C Function]

If *stayopen* is omitted, this is equivalent to `endhostent`. Otherwise it is equivalent to `sethostent stayopen`.

The Network Database

The following functions accept an object representing a network and return a selected component:

`netent:name` *net* [Scheme Procedure]

The "official" network name.

`netent:aliases` *net* [Scheme Procedure]

A list of aliases for the network.

`netent:addrtype` *net* [Scheme Procedure]

The type of the network number. Currently, this returns only `AF_INET`.

`netent:net` *net* [Scheme Procedure]

The network number.

The following procedures are used to search the network database:

`getnet` [*net*] [Scheme Procedure]
`getnetbyname` *net-name* [Scheme Procedure]
`getnetbyaddr` *net-number* [Scheme Procedure]
`scm_getnet` (*net*) [C Function]

Look up a network by name or net number in the network database. The *net-name* argument must be a string, and the *net-number* argument must be an integer. `getnet` will accept either type of argument, behaving like `getnetent` (see below) if no arguments are given.

The following procedures may be used to step through the network database from beginning to end.

`setnetent` [*stayopen*] [Scheme Procedure]

Initialize an internal stream from which network objects may be read. This procedure must be called before any calls to `getnetent`, and may also be called afterward to reset the net entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getnetbyname` or `getnetbyaddr` calls, possibly giving an efficiency gain.

`getnetent` [Scheme Procedure]

Return the next entry from the network database.

`endnetent` [Scheme Procedure]

Close the stream used by `getnetent`. The return value is unspecified.

`setnet` [*stayopen*] [Scheme Procedure]
`scm_setnet` (*stayopen*) [C Function]

If *stayopen* is omitted, this is equivalent to `endnetent`. Otherwise it is equivalent to `setnetent stayopen`.

The Protocol Database

The following functions accept an object representing a protocol and return a selected component:

`protoent:name` *protocol* [Scheme Procedure]

The “official” protocol name.

`protoent:aliases` *protocol* [Scheme Procedure]

A list of aliases for the protocol.

`protoent:proto` *protocol* [Scheme Procedure]

The protocol number.

The following procedures are used to search the protocol database:

`getproto` [*protocol*] [Scheme Procedure]
`getprotobyname` *name* [Scheme Procedure]
`getprotobynumber` *number* [Scheme Procedure]
`scm_getproto` (*protocol*) [C Function]

Look up a network protocol by name or by number. `getprotobyname` takes a string argument, and `getprotobynumber` takes an integer argument. `getproto` will accept either type, behaving like `getprotoent` (see below) if no arguments are supplied.

The following procedures may be used to step through the protocol database from beginning to end.

`setprotoent` [*stayopen*] [Scheme Procedure]

Initialize an internal stream from which protocol objects may be read. This procedure must be called before any calls to `getprotoent`, and may also be called afterward to reset the protocol entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getprotobyname` or `getprotobynumber` calls, possibly giving an efficiency gain.

`getprotoent` [Scheme Procedure]

Return the next entry from the protocol database.

`endprotoent` [Scheme Procedure]

Close the stream used by `getprotoent`. The return value is unspecified.

`setproto` [*stayopen*] [Scheme Procedure]
`scm_setproto` (*stayopen*) [C Function]

If *stayopen* is omitted, this is equivalent to `endprotoent`. Otherwise it is equivalent to `setprotoent stayopen`.

The Service Database

The following functions accept an object representing a service and return a selected component:

`servent:name` *serv* [Scheme Procedure]

The “official” name of the network service.

`servent:aliases` *serv* [Scheme Procedure]

A list of aliases for the network service.

`servent:port` *serv* [Scheme Procedure]

The Internet port used by the service.

`servent:proto` *serv* [Scheme Procedure]

The protocol used by the service. A service may be listed many times in the database under different protocol names.

The following procedures are used to search the service database:

`getserv` [*name* [*protocol*]] [Scheme Procedure]
`getservbyname` *name protocol* [Scheme Procedure]
`getservbyport` *port protocol* [Scheme Procedure]
`scm_getserv` (*name, protocol*) [C Function]

Look up a network service by name or by service number, and return a network service object. The *protocol* argument specifies the name of the desired protocol; if the protocol found in the network service database does not match this name, a system error is signalled.

The `getserv` procedure will take either a service name or number as its first argument; if given no arguments, it behaves like `getservent` (see below).

```
(getserv "imap" "tcp")
⇒ #("imap2" ("imap") 143 "tcp")

(getservbyport 88 "udp")
⇒ #("kerberos" ("kerberos5" "krb5") 88 "udp")
```

The following procedures may be used to step through the service database from beginning to end.

`setservent` [*stayopen*] [Scheme Procedure]

Initialize an internal stream from which service objects may be read. This procedure must be called before any calls to `getservent`, and may also be called afterward to reset the service entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getservbyname` or `getservbyport` calls, possibly giving an efficiency gain.

`getservent` [Scheme Procedure]

Return the next entry from the services database.

`endservent` [Scheme Procedure]

Close the stream used by `getservent`. The return value is unspecified.

`setserv` [*stayopen*] [Scheme Procedure]
`scm_setserv` (*stayopen*) [C Function]

If *stayopen* is omitted, this is equivalent to `endservent`. Otherwise it is equivalent to `setservent stayopen`.

6.2.11.3 Network Socket Address

A *socket address* object identifies a socket endpoint for communication. In the case of `AF_INET` for instance, the socket address object comprises the host address (or interface on the host) and a port number which specifies a particular open socket in a running client or server process. A socket address object can be created with,

`make-socket-address` *AF_INET ipv4addr port* [Scheme Procedure]
`make-socket-address` *AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`make-socket-address` *AF_UNIX path* [Scheme Procedure]

`scm_make_socket_address` *family address arglist* [C Function]

Return a new socket address object. The first argument is the address family, one of the `AF` constants, then the arguments vary according to the family.

For `AF_INET` the arguments are an IPv4 network address number (see Section 6.2.11.1 [Network Address Conversion], page 402), and a port number.

For `AF_INET6` the arguments are an IPv6 network address number and a port number. Optional *flowinfo* and *scopeid* arguments may be given (both integers, default 0).

For `AF_UNIX` the argument is a filename (a string).

The C function `scm_make_socket_address` takes the *family* and *address* arguments directly, then *arglist* is a list of further arguments, being the port for IPv4, port and optional flowinfo and scopeid for IPv6, or the empty list `SCM_EOL` for Unix domain.

The following functions access the fields of a socket address object,

`sockaddr:fam` *sa* [Scheme Procedure]

Return the address family from socket address object *sa*. This is one of the `AF` constants (eg. `AF_INET`).

`sockaddr:path` *sa* [Scheme Procedure]

For an `AF_UNIX` socket address object *sa*, return the filename.

`sockaddr:addr` *sa* [Scheme Procedure]

For an `AF_INET` or `AF_INET6` socket address object *sa*, return the network address number.

`sockaddr:port` *sa* [Scheme Procedure]

For an `AF_INET` or `AF_INET6` socket address object *sa*, return the port number.

`sockaddr:flowinfo` *sa* [Scheme Procedure]

For an `AF_INET6` socket address object *sa*, return the flowinfo value.

`sockaddr:scopeid` *sa* [Scheme Procedure]

For an `AF_INET6` socket address object *sa*, return the scope ID value.

The functions below convert to and from the C `struct sockaddr` (see section "Address Formats" in *The GNU C Library Reference Manual*). That structure is a generic type, an application can cast to or from `struct sockaddr_in`, `struct sockaddr_in6` or `struct sockaddr_un` according to the address family.

In a `struct sockaddr` taken or returned, the byte ordering in the fields follows the C conventions (see section "Byte Order Conversion" in *The GNU C Library Reference Manual*). This means network byte order for `AF_INET` host address (`sin_addr.s_addr`) and port number (`sin_port`), and `AF_INET6` port number (`sin6_port`). But at the Scheme level these values are taken or returned in host byte order, so the port is an ordinary integer, and the host address likewise is an ordinary integer (as described in Section 6.2.11.1 [Network Address Conversion], page 402).

`struct sockaddr * scm_c_make_socket_address` (*SCM family, SCM address, SCM args, size_t *outsize*) [C Function]

Return a newly-`malloc`ed `struct sockaddr` created from arguments like those taken by `scm_make_socket_address` above.

The size (in bytes) of the `struct sockaddr` return is stored into `*outsize`. An application must call `free` to release the returned structure when no longer required.

`SCM scm_from_sockaddr` (*const struct sockaddr *address, unsigned address_size*) [C Function]

Return a Scheme socket address object from the C *address* structure. *address_size* is the size in bytes of *address*.

`struct sockaddr * scm_to_sockaddr` (*SCM address, size_t *address_size*) [C Function]

Return a newly-`malloc`ed `struct sockaddr` from a Scheme level socket address object.

The size (in bytes) of the `struct sockaddr` return is stored into `*outsize`. An application must call `free` to release the returned structure when no longer required.

6.2.11.4 Network Sockets and Communication

Socket ports can be created using `socket` and `socketpair`. The ports are initially unbuffered, to make reading and writing to the same port more reliable. A buffer can be added to the port using `setvbuf`; see Section 6.2.2 [Ports and File Descriptors], page 374.

Most systems have limits on how many files and sockets can be open, so it's strongly recommended that socket ports be closed explicitly when no longer required (see Section 5.12.1 [Ports], page 271).

Some of the underlying C functions take values in network byte order, but the convention in Guile is that at the Scheme level everything is ordinary host byte order and conversions are made automatically where necessary.

`socket` *family style proto* [Scheme Procedure]
`scm_socket` (*family, style, proto*) [C Function]

Return a new socket port of the type specified by *family*, *style* and *proto*. All three parameters are integers. The possible values for *family* are as follows, where supported by the system,

`PF_UNIX` [Variable]
`PF_INET` [Variable]
`PF_INET6` [Variable]

The possible values for *style* are as follows, again where supported by the system,

`SOCK_STREAM` [Variable]
`SOCK_DGRAM` [Variable]
`SOCK_RAW` [Variable]
`SOCK_RDM` [Variable]
`SOCK_SEQPACKET` [Variable]

proto can be obtained from a protocol name using `getprotobyname` (see Section 6.2.11.2 [Network Databases], page 404). A value of zero means the default protocol, which is usually right.

A socket cannot by used for communication until it has been connected somewhere, usually with either `connect` or `accept` below.

`socketpair` *family style proto* [Scheme Procedure]
`scm_socketpair` (*family, style, proto*) [C Function]

Return a pair, the `car` and `cdr` of which are two unnamed socket ports connected to each other. The connection is full-duplex, so data can be transferred in either direction between the two.

family, *style* and *proto* are as per `socket` above. But many systems only support socket pairs in the `PF_UNIX` family. Zero is likely to be the only meaningful value for *proto*.

`getsockopt` *sock level optname* [Scheme Procedure]
`setsockopt` *sock level optname value* [Scheme Procedure]
`scm_getsockopt` (*sock, level, optname*) [C Function]
`scm_setsockopt` (*sock, level, optname, value*) [C Function]

Get or set an option on socket port *sock*. `getsockopt` returns the current value. `setsockopt` sets a value and the return is unspecified.

level is an integer specifying a protocol layer, either `SOL_SOCKET` for socket level options, or a protocol number from the `IPPROTO` constants or `getprotoent` (see Section 6.2.11.2 [Network Databases], page 404).

`SOL_SOCKET` [Variable]
`IPPROTO_IP` [Variable]
`IPPROTO_TCP` [Variable]
`IPPROTO_UDP` [Variable]

optname is an integer specifying an option within the protocol layer.

For `SOL_SOCKET` level the following *optnames* are defined (when provided by the system). For their meaning see section "Socket-Level Options" in *The GNU C Library Reference Manual*, or `man 7 socket`.

`SO_DEBUG` [Variable]
`SO_REUSEADDR` [Variable]
`SO_STYLE` [Variable]
`SO_TYPE` [Variable]
`SO_ERROR` [Variable]
`SO_DONTROUTE` [Variable]
`SO_BROADCAST` [Variable]
`SO_SNDBUF` [Variable]
`SO_RCVBUF` [Variable]
`SO_KEEPALIVE` [Variable]
`SO_OOBINLINE` [Variable]
`SO_NO_CHECK` [Variable]
`SO_PRIORITY` [Variable]

The *value* taken or returned is an integer.

`SO_LINGER` [Variable]

The *value* taken or returned is a pair of integers `(`*`ENABLE`*` . `*`TIMEOUT`*`)`. On old systems without timeout support (ie. without `struct linger`), only *ENABLE* has an effect but the value in Guile is always a pair.

For IP level (`IPPROTO_IP`) the following *optnames* are defined (when provided by the system). See `man ip` for what they mean.

`IP_ADD_MEMBERSHIP` [Variable]
`IP_DROP_MEMBERSHIP` [Variable]

These can be used only with `setsockopt`, not `getsockopt`. *value* is a pair (*`MULTIADDR`* `.` *`INTERFACEADDR`*) of integer IPv4 addresses (see Section 6.2.11.1 [Network Address Conversion], page 402). *MULTIADDR* is a multicast address to be added to or dropped from the interface *INTERFACEADDR*. *INTERFACEADDR* can be `INADDR_ANY` to have the system select the interface. *INTERFACEADDR* can also be an interface index number, on systems supporting that.

`shutdown` *sock how* [Scheme Procedure]
`scm_shutdown` (*sock*, *how*) [C Function]

Sockets can be closed simply by using `close-port`. The `shutdown` procedure allows reception or transmission on a connection to be shut down individually, according to the parameter *how*:

0	Stop receiving data for this socket. If further data arrives, reject it.
1	Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don't retransmit it if it is lost.
2	Stop both reception and transmission.

The return value is unspecified.

`connect` *sock sockaddr* [Scheme Procedure]
`connect` *sock AF_INET ipv4addr port* [Scheme Procedure]
`connect` *sock AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`connect` *sock AF_UNIX path* [Scheme Procedure]
`scm_connect` (*sock*, *fam*, *address*, *args*) [C Function]

Initiate a connection on socket port *sock* to a given address. The destination is either a socket address object, or arguments the same as `make-socket-address` would take to make such an object (see Section 6.2.11.3 [Network Socket Address], page 408). The return value is unspecified.

```
(connect sock AF_INET INADDR_LOCALHOST 23)
(connect sock (make-socket-address AF_INET INADDR_LOCALHOST 23))
```

`bind` *sock sockaddr* [Scheme Procedure]
`bind` *sock AF_INET ipv4addr port* [Scheme Procedure]
`bind` *sock AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`bind` *sock AF_UNIX path* [Scheme Procedure]
`scm_bind` (*sock*, *fam*, *address*, *args*) [C Function]

Bind socket port *sock* to the given address. The address is either a socket address object, or arguments the same as `make-socket-address` would take to make such an object (see Section 6.2.11.3 [Network Socket Address], page 408). The return value is unspecified.

Generally a socket is only explicitly bound to a particular address when making a server, ie. to listen on a particular port. For an outgoing connection the system will assign a local address automatically, if not already bound.

```
(bind sock AF_INET INADDR_ANY 12345)
(bind sock (make-socket-object AF_INET INADDR_ANY 12345))
```

`listen` *sock backlog* [Scheme Procedure]
`scm_listen` (*sock, backlog*) [C Function]

Enable *sock* to accept connection requests. *backlog* is an integer specifying the maximum length of the queue for pending connections. If the queue fills, new clients will fail to connect until the server calls `accept` to accept a connection from the queue.

The return value is unspecified.

`accept` *sock* [Scheme Procedure]
`scm_accept` (*sock*) [C Function]

Accept a connection from socket port *sock* which has been enabled for listening with `listen` above. If there are no incoming connections in the queue, wait until one is available (unless `O_NONBLOCK` has been set on the socket, see Section 6.2.2 [Ports and File Descriptors], page 374).

The return value is a pair. The `car` is a new socket port, connected and ready to communicate. The `cdr` is a socket address object (see Section 6.2.11.3 [Network Socket Address], page 408) which is where the remote connection is from (like `getpeername` below).

All communication takes place using the new socket returned. The given *sock* remains bound and listening, and `accept` may be called on it again to get another incoming connection when desired.

`getsockname` *sock* [Scheme Procedure]
`scm_getsockname` (*sock*) [C Function]

Return a socket address object which is the where *sock* is bound locally. *sock* may have obtained its local address from `bind` (above), or if a `connect` is done with an otherwise unbound socket (which is usual) then the system will have assigned an address.

Note that on many systems the address of a socket in the `AF_UNIX` namespace cannot be read.

`getpeername` *sock* [Scheme Procedure]
`scm_getpeername` (*sock*) [C Function]

Return a socket address object which is where *sock* is connected to, ie. the remote endpoint.

Note that on many systems the address of a socket in the `AF_UNIX` namespace cannot be read.

`recv!` *sock buf* [*flags*] [Scheme Procedure]
`scm_recv` (*sock, buf, flags*) [C Function]

Receive data from a socket port. *sock* must already be bound to the address from which data is to be received. *buf* is a string into which the data will be written. The size of *buf* limits the amount of data which can be received: in the case of packet protocols, if a packet larger than this limit is encountered then some data will be irrevocably lost.

The optional *flags* argument is a value or bitwise OR of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes read from the socket.

Note that the data is read directly from the socket file descriptor: any unread buffered port data is ignored.

`send` *sock message* [*flags*] [Scheme Procedure]
`scm_send` (*sock, message, flags*) [C Function]

Transmit the string *message* on a socket port *sock*. *sock* must already be bound to a destination address. The value returned is the number of bytes transmitted—it's possible for this to be less than the length of *message* if the socket is set to be non-blocking. The optional *flags* argument is a value or bitwise OR of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

`recvfrom!` *sock str* [*flags* [*start* [*end*]]] [Scheme Procedure]
`scm_recvfrom` (*sock, str, flags, start, end*) [C Function]

Receive data from socket port *sock*, returning the originating address as well as the data. This function is usually for datagram sockets, but can be used on stream-oriented sockets too.

The data received is stored in the given *str*, the whole string or just the region between the optional *start* and *end* positions. The size of *str* limits the amount of data which can be received. For datagram protocols if a packet larger than this is received then excess bytes are irrevocably lost.

The return value is a pair. The `car` is the number of bytes read. The `cdr` is a socket address object (see Section 6.2.11.3 [Network Socket Address], page 408) which is where the data came from, or `#f` if the origin is unknown.

The optional *flags* argument is a or bitwise-OR (`logior`) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Data is read directly from the socket file descriptor, any buffered port data is ignored.

On a GNU/Linux system `recvfrom!` is not multi-threading, all threads stop while a `recvfrom!` call is in progress. An application may need to use `select`, `O_NONBLOCK` or `MSG_DONTWAIT` to avoid this.

`sendto` *sock message sockaddr* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET ipv4addr port* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET6 ipv6addr port* [*flowinfo* [*scopeid* [*flags*]]] [Scheme Procedure]
`sendto` *sock message AF_UNIX path* [*flags*] [Scheme Procedure]
`scm_sendto` (*sock, message, fam, address, args_and_flags*) [C Function]

Transmit the string *message* as a datagram on socket port *sock*. The destination is specified either as a socket address object, or as arguments the same as would be taken by `make-socket-address` to create such an object (see Section 6.2.11.3 [Network Socket Address], page 408).

The destination address may be followed by an optional *flags* argument which is a `logior` (see Section 5.5.2.14 [Bitwise Operations], page 117) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes transmitted – it's possible for this to be less than the length of *message* if the socket is set to be non-blocking. Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

The following functions can be used to convert short and long integers between "host" and "network" order. Although the procedures above do this automatically for addresses, the conversion will still need to be done when sending or receiving encoded integer data from the network.

`htons` *value* [Scheme Procedure]
`scm_htons` (*value*) [C Function]
Convert a 16 bit quantity from host to network byte ordering. *value* is packed into 2 bytes, which are then converted and returned as a new integer.

`ntohs` *value* [Scheme Procedure]
`scm_ntohs` (*value*) [C Function]
Convert a 16 bit quantity from network to host byte ordering. *value* is packed into 2 bytes, which are then converted and returned as a new integer.

`htonl` *value* [Scheme Procedure]
`scm_htonl` (*value*) [C Function]
Convert a 32 bit quantity from host to network byte ordering. *value* is packed into 4 bytes, which are then converted and returned as a new integer.

`ntohl` *value* [Scheme Procedure]
`scm_ntohl` (*value*) [C Function]
Convert a 32 bit quantity from network to host byte ordering. *value* is packed into 4 bytes, which are then converted and returned as a new integer.

These procedures are inconvenient to use at present, but consider:

```
(define write-network-long
  (lambda (value port)
    (let ((v (make-uniform-vector 1 1 0)))
      (uniform-vector-set! v 0 (htonl value))
      (uniform-vector-write v port))))

(define read-network-long
  (lambda (port)
    (let ((v (make-uniform-vector 1 1 0)))
      (uniform-vector-read! v port)
      (ntohl (uniform-vector-ref v 0)))))
```

6.2.11.5 Network Socket Examples

The following give examples of how to use network sockets.

Internet Socket Client Example

The following example demonstrates an Internet socket client. It connects to the HTTP daemon running on the local machine and returns the contents of the root index URL.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (connect s AF_INET (inet-aton "127.0.0.1") 80)
  (display "GET / HTTP/1.0\r\n\r\n" s)

  (do ((line (read-line s) (read-line s)))
      ((eof-object? line))
    (display line)
    (newline)))
```

Internet Socket Server Example

The following example shows a simple Internet server which listens on port 2904 for incoming connections and sends a greeting back to the client.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (setsockopt s SOL_SOCKET SO_REUSEADDR 1)
  ;; Specific address?
  ;; (bind s AF_INET (inet-aton "127.0.0.1") 2904)
  (bind s AF_INET INADDR_ANY 2904)
  (listen s 5)

  (simple-format #t "Listening for clients in pid: ~S" (getpid))
  (newline)

  (while #t
    (let* ((client-connection (accept s))
           (client-details (cdr client-connection))
           (client (car client-connection)))
      (simple-format #t "Got new client connection: ~S"
                     client-details)
      (newline)
      (simple-format #t "Client address: ~S"
                     (gethostbyaddr
                      (sockaddr:addr client-details)))
      (newline)
      ;; Send back the greeting to the client port
      (display "Hello client\r\n" client)
      (close client))))
```

6.2.12 System Identification

This section lists the various procedures Guile provides for accessing information about the system it runs on.

`uname` [Scheme Procedure]
`scm_uname ()` [C Function]

Return an object with some information about the computer system the program is running on.

The following procedures accept an object as returned by `uname` and return a selected component (all of which are strings).

`utsname:sysname` *un* [Scheme Procedure]

The name of the operating system.

`utsname:nodename` *un* [Scheme Procedure]

The network name of the computer.

`utsname:release` *un* [Scheme Procedure]

The current release level of the operating system implementation.

`utsname:version` *un* [Scheme Procedure]

The current version level within the release of the operating system.

`utsname:machine` *un* [Scheme Procedure]

A description of the hardware.

`gethostname` [Scheme Procedure]
`scm_gethostname ()` [C Function]

Return the host name of the current processor.

`sethostname` *name* [Scheme Procedure]
`scm_sethostname (`*name*`)` [C Function]

Set the host name of the current processor to *name*. May only be used by the superuser. The return value is not specified.

6.2.13 Locales

`setlocale` *category* [*locale*] [Scheme Procedure]
`scm_setlocale (`*category*, *locale*`)` [C Function]

Get or set the current locale, used for various internationalizations. Locales are strings, such as '`sv_SE`'.

If *locale* is given then the locale for the given *category* is set and the new value returned. If *locale* is not given then the current value is returned. *category* should be one of the following values

`LC_ALL` [Variable]
`LC_COLLATE` [Variable]
`LC_CTYPE` [Variable]
`LC_MESSAGES` [Variable]
`LC_MONETARY` [Variable]
`LC_NUMERIC` [Variable]
`LC_TIME` [Variable]

A common usage is '`(setlocale LC_ALL "")`', which initializes all categories based on standard environment variables (`LANG` etc). For full details on categories and locale names see section "Locales and Internationalization" in *The GNU C Library Reference Manual*.

6.2.14 Encryption

Please note that the procedures in this section are not suited for strong encryption, they are only interfaces to the well-known and common system library functions of the same name. They are just as good (or bad) as the underlying functions, so you should refer to your system documentation before using them.

`crypt` *key salt* [Scheme Procedure]
`scm_crypt` (*key*, *salt*) [C Function]

Encrypt *key* using *salt* as the salt value to the crypt(3) library call.

Although `getpass` is not an encryption procedure per se, it appears here because it is often used in combination with `crypt`:

`getpass` *prompt* [Scheme Procedure]
`scm_getpass` (*prompt*) [C Function]

Display *prompt* to the standard error output and read a password from '`/dev/tty`'. If this file is not accessible, it reads from standard input. The password may be up to 127 characters in length. Additional characters and the terminating newline character are discarded. While reading the password, echoing and the generation of signals by special characters is disabled.

6.3 The (ice-9 getopt-long) Module

The `(ice-9 getopt-long)` module exports two procedures: `getopt-long` and `option-ref`.

- `getopt-long` takes a list of strings — the command line arguments — and an *option specification*. It parses the command line arguments according to the option specification and returns a data structure that encapsulates the results of the parsing.
- `option-ref` then takes the parsed data structure and a specific option's name, and returns information about that option in particular.

To make these procedures available to your Guile script, include the expression `(use-modules (ice-9 getopt-long))` somewhere near the top, before the first usage of `getopt-long` or `option-ref`.

6.3.1 A Short getopt-long Example

This section illustrates how `getopt-long` is used by presenting and dissecting a simple example. The first thing that we need is an *option specification* that tells `getopt-long` how to parse the command line. This specification is an association list with the long option name as the key. Here is how such a specification might look:

```
(define option-spec
  '((version (single-char #\v) (value #f))
    (help    (single-char #\h) (value #f))))
```

This alist tells `getopt-long` that it should accept two long options, called *version* and *help*, and that these options can also be selected by the single-letter abbreviations *v* and *h*, respectively. The `(value #f)` clauses indicate that neither of the options accepts a value.

With this specification we can use `getopt-long` to parse a given command line:

```
(define options (getopt-long (command-line) option-spec))
```

After this call, `options` contains the parsed command line and is ready to be examined by `option-ref`. `option-ref` is called like this:

```
(option-ref options 'help #f)
```

It expects the parsed command line, a symbol indicating the option to examine, and a default value. The default value is returned if the option was not present in the command line, or if the option was present but without a value; otherwise the value from the command line is returned. Usually `option-ref` is called once for each possible option that a script supports.

The following example shows a main program which puts all this together to parse its command line and figure out what the user wanted.

```
(define (main args)
  (let* ((option-spec '((version (single-char #\v) (value #f))
                        (help    (single-char #\h) (value #f))))
         (options (getopt-long args option-spec))
         (help-wanted (option-ref options 'help #f))
         (version-wanted (option-ref options 'version #f)))
    (if (or version-wanted help-wanted)
        (begin
          (if version-wanted
              (display "getopt-long-example version 0.3\n"))
          (if help-wanted
              (display "\
getopt-long-example [options]
  -v, --version    Display version
  -h, --help       Display this help
")))
        (begin
          (display "Hello, World!") (newline)))))
```

6.3.2 How to Write an Option Specification

An option specification is an association list (see Section 5.6.11 [Association Lists], page 210) with one list element for each supported option. The key of each list element is a symbol that names the option, while the value is a list of option properties:

```
OPTION-SPEC ::=  '( (OPT-NAME1 (PROP-NAME PROP-VALUE) ...)
                    (OPT-NAME2 (PROP-NAME PROP-VALUE) ...)
                    (OPT-NAME3 (PROP-NAME PROP-VALUE) ...)
                    ...
                  )
```

Each *opt-name* specifies the long option name for that option. For example, a list element with *opt-name* `background` specifies an option that can be specified on the command line using the long option `--background`. Further information about the option — whether it takes a value, whether it is required to be present in the command line, and so on — is specified by the option properties.

In the example of the preceding section, we already saw that a long option name can have a equivalent *short option* character. The equivalent short option character can be set for an option by specifying a `single-char` property in that option's property list. For example, a list element like `'(output (single-char #\o) ...)` specifies an option with long name `--output` that can also be specified by the equivalent short name `-o`.

The `value` property specifies whether an option requires or accepts a value. If the `value` property is set to `#t`, the option requires a value: `getopt-long` will signal an error if the option name is present without a corresponding value. If set to `#f`, the option does not take a value; in this case, a non-option word that follows the option name in the command line will be treated as a non-option argument. If set to the symbol `optional`, the option accepts a value but does not require one: a non-option word that follows the option name in the command line will be interpreted as that option's value. If the option name for an option with `'(value optional)` is immediately followed in the command line by *another* option name, the value for the first option is implicitly `#t`.

The `required?` property indicates whether an option is required to be present in the command line. If the `required?` property is set to `#t`, `getopt-long` will signal an error if the option is not specified.

Finally, the `predicate` property can be used to constrain the possible values of an option. If used, the `predicate` property should be set to a procedure that takes one argument — the proposed option value as a string — and returns either `#t` or `#f` according as the proposed value is or is not acceptable. If the predicate procedure returns `#f`, `getopt-long` will signal an error.

By default, options do not have single-character equivalents, are not required, and do not take values. Where the list element for an option includes a `value` property but no `predicate` property, the option values are unconstrained.

6.3.3 Expected Command Line Format

In order for `getopt-long` to correctly parse a command line, that command line must conform to a standard set of rules for how command line options are specified. This section explains what those rules are.

`getopt-long` splits a given command line into several pieces. All elements of the argument list are classified to be either options or normal arguments. Options consist of two dashes and an option name (so-called *long* options), or of one dash followed by a single letter (*short* options).

Options can behave as switches, when they are given without a value, or they can be used to pass a value to the program. The value for an option may be specified using an equals sign, or else is simply the next word in the command line, so the following two invocations are equivalent:

```
$ ./foo.scm --output=bar.txt
$ ./foo.scm --output bar.txt
```

Short options can be used instead of their long equivalents and can be grouped together after a single dash. For example, the following commands are equivalent.

```
$ ./foo.scm --version --help
$ ./foo.scm -v --help
$ ./foo.scm -vh
```

If an option requires a value, it can only be grouped together with other short options if it is the last option in the group; the value is the next argument. So, for example, with the following option specification —

```
((apples    (single-char #\a))
 (blimps    (single-char #\b) (value #t))
 (catalexis (single-char #\c) (value #t)))
```

— the following command lines would all be acceptable:

```
$ ./foo.scm -a -b bang -c couth
$ ./foo.scm -ab bang -c couth
$ ./foo.scm -ac couth -b bang
```

But the next command line is an error, because `-b` is not the last option in its combination, and because a group of short options cannot include two options that both require values:

```
$ ./foo.scm -abc couth bang
```

If an option's value is optional, `getopt-long` decides whether the option has a value by looking at what follows it in the argument list. If the next element is a string, and it does not appear to be an option itself, then that string is the option's value.

If the option `--` appears in the argument list, argument parsing stops there and subsequent arguments are returned as ordinary arguments, even if they resemble options. So, with the command line

```
$ ./foo.scm --apples "Granny Smith" -- --blimp Goodyear
```

`getopt-long` will recognize the `--apples` option as having the value "Granny Smith", but will not treat `--blimp` as an option. The strings `--blimp` and `Goodyear` will be returned as ordinary argument strings.

6.3.4 Reference Documentation for `getopt-long`

`getopt-long` *args grammar* [Scheme Procedure]

Parse the command line given in *args* (which must be a list of strings) according to the option specification *grammar*.

The *grammar* argument is expected to be a list of this form:

`((option (property value) ...) ...)`

where each *option* is a symbol denoting the long option, but without the two leading dashes (e.g. `version` if the option is called `--version`).

For each option, there may be list of arbitrarily many property/value pairs. The order of the pairs is not important, but every property may only appear once in the property list. The following table lists the possible properties:

`(single-char char)`
: Accept `-char` as a single-character equivalent to `--option`. This is how to specify traditional Unix-style flags.

`(required? bool)`
: If *bool* is true, the option is required. `getopt-long` will raise an error if it is not found in *args*.

`(value` *`bool`*`)`
: If *bool* is `#t`, the option accepts a value; if it is `#f`, it does not; and if it is the symbol `optional`, the option may appear in *args* with or without a value.

`(predicate` *`func`*`)`
: If the option accepts a value (i.e. you specified `(value #t)` for this option), then `getopt-long` will apply *func* to the value, and throw an exception if it returns `#f`. *func* should be a procedure which accepts a string and returns a boolean value; you may need to use quasiquotes to get it into *grammar*.

`getopt-long`'s *args* parameter is expected to be a list of strings like the one returned by `command-line`, with the first element being the name of the command. Therefore `getopt-long` ignores the first element in *args* and starts argument interpretation with the second element.

`getopt-long` signals an error if any of the following conditions hold.

- The option grammar has an invalid syntax.
- One of the options in the argument list was not specified by the grammar.
- A required option is omitted.
- An option which requires an argument did not get one.
- An option that doesn't accept an argument does get one (this can only happen using the long option `--opt=`*`value`* syntax).
- An option predicate fails.

6.3.5 Reference Documentation for `option-ref`

`option-ref` *options key default* [Scheme Procedure]

Search *options* for a command line option named *key* and return its value, if found. If the option has no value, but was given, return `#t`. If the option was not given, return *default*. *options* must be the result of a call to `getopt-long`.

`option-ref` always succeeds, either by returning the requested option value from the command line, or the default value.

The special key `'()` can be used to get a list of all non-option arguments.

6.4 SRFI Support Modules

SRFI is an acronym for Scheme Request For Implementation. The SRFI documents define a lot of syntactic and procedure extensions to standard Scheme as defined in R5RS.

Guile has support for a number of SRFIs. This chapter gives an overview over the available SRFIs and some usage hints. For complete documentation, design rationales and further examples, we advise you to get the relevant SRFI documents from the SRFI home page `http://srfi.schemers.org`.

6.4.1 About SRFI Usage

SRFI support in Guile is currently implemented partly in the core library, and partly as add-on modules. That means that some SRFIs are automatically available when the interpreter is started, whereas the other SRFIs require you to use the appropriate support module explicitly.

There are several reasons for this inconsistency. First, the feature checking syntactic form `cond-expand` (see Section 6.4.2 [SRFI-0], page 423) must be available immediately, because it must be there when the user wants to check for the Scheme implementation, that is, before she can know that it is safe to use `use-modules` to load SRFI support modules. The second reason is that some features defined in SRFIs had been implemented in Guile before the developers started to add SRFI implementations as modules (for example SRFI-6 (see Section 6.4.6 [SRFI-6], page 438)). In the future, it is possible that SRFIs in the core library might be factored out into separate modules, requiring explicit module loading when they are needed. So you should be prepared to have to use `use-modules` someday in the future to access SRFI-6 bindings. If you want, you can do that already. We have included the module `(srfi srfi-6)` in the distribution, which currently does nothing, but ensures that you can write future-safe code.

Generally, support for a specific SRFI is made available by using modules named `(srfi srfi-number)`, where *number* is the number of the SRFI needed. Another possibility is to use the command line option `--use-srfi`, which will load the necessary modules automatically (see Section 3.3.2 [Invoking Guile], page 33).

6.4.2 SRFI-0 - cond-expand

This SRFI lets a portable Scheme program test for the presence of certain features, and adapt itself by using different blocks of code, or fail if the necessary features are not available. There's no module to load, this is in the Guile core.

A program designed only for Guile will generally not need this mechanism, such a program can of course directly use the various documented parts of Guile.

`cond-expand` (*feature body...*) ... [syntax]

Expand to the *body* of the first clause whose *feature* specification is satisfied. It is an error if no *feature* is satisfied.

Features are symbols such as `srfi-1`, and a feature specification can use `and`, `or` and `not` forms to test combinations. The last clause can be an `else`, to be used if no other passes.

For example, define a private version of `alist-cons` if SRFI-1 is not available.

```
          (cond-expand (srfi-1
                         )
                        (else
                         (define (alist-cons key val alist)
                           (cons (cons key val) alist))))
```

Or demand a certain set of SRFIs (list operations, string ports, `receive` and string operations), failing if they're not available.

```
          (cond-expand ((and srfi-1 srfi-6 srfi-8 srfi-13)
                         ))
```

The Guile core has the following features,

```
     guile
     r5rs
     srfi-0
     srfi-4
     srfi-6
     srfi-13
     srfi-14
```

Other SRFI feature symbols are defined once their code has been loaded with `use-modules`, since only then are their bindings available.

The '`--use-srfi`' command line option (see Section 3.3.2 [Invoking Guile], page 33) is a good way to load SRFIs to satisfy `cond-expand` when running a portable program.

Testing the `guile` feature allows a program to adapt itself to the Guile module system, but still run on other Scheme systems. For example the following demands SRFI-8 (`receive`), but also knows how to load it with the Guile mechanism.

```
     (cond-expand (srfi-8
                    )
                   (guile
                    (use-modules (srfi srfi-8))))
```

It should be noted that `cond-expand` is separate from the `*features*` mechanism (see Section 5.18.2 [Feature Tracking], page 336), feature symbols in one are unrelated to those in the other.

6.4.3 SRFI-1 - List library

The list library defined in SRFI-1 contains a lot of useful list processing procedures for construction, examining, destructuring and manipulating lists and pairs.

Since SRFI-1 also defines some procedures which are already contained in R5RS and thus are supported by the Guile core library, some list and pair procedures which appear in the SRFI-1 document may not appear in this section. So when looking for a particular list/pair processing procedure, you should also have a look at the sections Section 5.6.2 [Lists], page 168 and Section 5.6.1 [Pairs], page 166.

6.4.3.1 Constructors

New lists can be constructed by calling one of the following procedures.

`xcons` *d a* [Scheme Procedure]

Like `cons`, but with interchanged arguments. Useful mostly when passed to higher-order procedures.

`list-tabulate` *n init-proc* [Scheme Procedure]

Return an *n*-element list, where each list element is produced by applying the procedure *init-proc* to the corresponding list index. The order in which *init-proc* is applied to the indices is not specified.

`list-copy` *lst* [Scheme Procedure]

Return a new list containing the elements of the list *lst*.

This function differs from the core `list-copy` (see Section 5.6.2.3 [List Constructors], page 170) in accepting improper lists too. And if *lst* is not a pair at all then it's treated as the final tail of an improper list and simply returned.

`circular-list` *elt1 elt2 . . .* [Scheme Procedure]

Return a circular list containing the given arguments *elt1 elt2 . . .* .

`iota` *count* [*start step*] [Scheme Procedure]

Return a list containing *count* numbers, starting from *start* and adding *step* each time. The default *start* is 0, the default *step* is 1. For example,

```
(iota 6)         ⇒ (0 1 2 3 4 5)
(iota 4 2.5 -2) ⇒ (2.5 0.5 -1.5 -3.5)
```

This function takes its name from the corresponding primitive in the APL language.

6.4.3.2 Predicates

The procedures in this section test specific properties of lists.

`proper-list?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a proper list, or `#f` otherwise. This is the same as the core `list?` (see Section 5.6.2.2 [List Predicates], page 169).

A proper list is a list which ends with the empty list `()` in the usual way. The empty list `()` itself is a proper list too.

```
(proper-list? '(1 2 3))  ⇒ #t
(proper-list? '())       ⇒ #t
```

`circular-list?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a circular list, or `#f` otherwise.

A circular list is a list where at some point the `cdr` refers back to a previous pair in the list (either the start or some later point), so that following the `cdr`s takes you around in a circle, with no end.

```
(define x (list 1 2 3 4))
(set-cdr! (last-pair x) (cddr x))
x ⇒ (1 2 3 4 3 4 3 4 ...)
(circular-list? x)  ⇒ #t
```

`dotted-list?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a dotted list, or `#f` otherwise.

A dotted list is a list where the `cdr` of the last pair is not the empty list `()`. Any non-pair *obj* is also considered a dotted list, with length zero.

```
(dotted-list? '(1 2 . 3))  ⇒ #t
(dotted-list? 99)          ⇒ #t
```

It will be noted that any Scheme object passes exactly one of the above three tests `proper-list?`, `circular-list?` and `dotted-list?`. Non-lists are `dotted-list?`, finite lists are either `proper-list?` or `dotted-list?`, and infinite lists are `circular-list?`.

`null-list?` *lst* [Scheme Procedure]

Return `#t` if *lst* is the empty list `()`, `#f` otherwise. If something else than a proper or circular list is passed as *lst*, an error is signalled. This procedure is recommended for checking for the end of a list in contexts where dotted lists are not allowed.

`not-pair?` *obj* [Scheme Procedure]

Return `#t` is *obj* is not a pair, `#f` otherwise. This is shorthand notation `(not (pair? obj))` and is supposed to be used for end-of-list checking in contexts where dotted lists are allowed.

`list=` *elt= list1 . . .* [Scheme Procedure]

Return `#t` if all argument lists are equal, `#f` otherwise. List equality is determined by testing whether all lists have the same length and the corresponding elements are equal in the sense of the equality predicate *elt=*. If no or only one list is given, `#t` is returned.

6.4.3.3 Selectors

`first` *pair* [Scheme Procedure]
`second` *pair* [Scheme Procedure]
`third` *pair* [Scheme Procedure]
`fourth` *pair* [Scheme Procedure]
`fifth` *pair* [Scheme Procedure]
`sixth` *pair* [Scheme Procedure]
`seventh` *pair* [Scheme Procedure]
`eighth` *pair* [Scheme Procedure]
`ninth` *pair* [Scheme Procedure]
`tenth` *pair* [Scheme Procedure]

These are synonyms for `car`, `cadr`, `caddr`, ...

`car+cdr` *pair* [Scheme Procedure]

Return two values, the CAR and the CDR of *pair*.

`take` *lst i* [Scheme Procedure]
`take!` *lst i* [Scheme Procedure]

Return a list containing the first *i* elements of *lst*.

`take!` may modify the structure of the argument list *lst* in order to produce the result.

`drop` *lst i* [Scheme Procedure]

Return a list containing all but the first *i* elements of *lst*.

`take-right` *lst i* [Scheme Procedure]

Return the a list containing the *i* last elements of *lst*. The return shares a common tail with *lst*.

`drop-right` *lst i* [Scheme Procedure]
`drop-right!` *lst i* [Scheme Procedure]

Return the a list containing all but the *i* last elements of *lst*.

`drop-right` always returns a new list, even when *i* is zero. `drop-right!` may modify the structure of the argument list *lst* in order to produce the result.

`split-at` *lst i* [Scheme Procedure]
`split-at!` *lst i* [Scheme Procedure]

Return two values, a list containing the first *i* elements of the list *lst* and a list containing the remaining elements.

`split-at!` may modify the structure of the argument list *lst* in order to produce the result.

`last` *lst* [Scheme Procedure]

Return the last element of the non-empty, finite list *lst*.

6.4.3.4 Length, Append, Concatenate, etc.

`length+` *lst* [Scheme Procedure]

Return the length of the argument list *lst*. When *lst* is a circular list, `#f` is returned.

`concatenate` *list-of-lists* [Scheme Procedure]
`concatenate!` *list-of-lists* [Scheme Procedure]

Construct a list by appending all lists in *list-of-lists*.

`concatenate!` may modify the structure of the given lists in order to produce the result.

`concatenate` is the same as `(apply append list-of-lists)`. It exists because some Scheme implementations have a limit on the number of arguments a function takes, which the `apply` might exceed. In Guile there is no such limit.

`append-reverse` *rev-head tail* [Scheme Procedure]
`append-reverse!` *rev-head tail* [Scheme Procedure]

Reverse *rev-head*, append *tail* to it, and return the result. This is equivalent to `(append (reverse rev-head) tail)`, but its implementation is more efficient.

```
(append-reverse '(1 2 3) '(4 5 6)) ⇒ (3 2 1 4 5 6)
```

`append-reverse!` may modify *rev-head* in order to produce the result.

`zip` *lst1 lst2 ...* [Scheme Procedure]

Return a list as long as the shortest of the argument lists, where each element is a list. The first list contains the first elements of the argument lists, the second list contains the second elements, and so on.

`unzip1` *lst* [Scheme Procedure]
`unzip2` *lst* [Scheme Procedure]
`unzip3` *lst* [Scheme Procedure]
`unzip4` *lst* [Scheme Procedure]
`unzip5` *lst* [Scheme Procedure]

`unzip1` takes a list of lists, and returns a list containing the first elements of each list, `unzip2` returns two lists, the first containing the first elements of each lists and the second containing the second elements of each lists, and so on.

`count` *pred lst1 . . . lstN* [Scheme Procedure]

Return a count of the number of times *pred* returns true when called on elements from the given lists.

pred is called with *N* parameters `(pred elem1 ... elemN)`, each element being from the corresponding *lst1 . . . lstN*. The first call is with the first element of each list, the second with the second element from each, and so on.

Counting stops when the end of the shortest list is reached. At least one list must be non-circular.

6.4.3.5 Fold, Unfold & Map

`fold` *proc init lst1 . . . lstN* [Scheme Procedure]
`fold-right` *proc init lst1 . . . lstN* [Scheme Procedure]

Apply *proc* to the elements of *lst1 . . . lstN* to build a result, and return that result.

Each *proc* call is `(proc elem1 ... elemN previous)`, where *elem1* is from *lst1*, through *elemN* from *lstN*. *previous* is the return from the previous call to *proc*, or the given *init* for the first call. If any list is empty, just *init* is returned.

`fold` works through the list elements from first to last. The following shows a list reversal and the calls it makes,

```
(fold cons '() '(1 2 3))

(cons 1 '())
(cons 2 '(1))
(cons 3 '(2 1)
⇒ (3 2 1)
```

`fold-right` works through the list elements from last to first, ie. from the right. So for example the following finds the longest string, and the last among equal longest,

```
(fold-right (lambda (str prev)
              (if (> (string-length str) (string-length prev))
                  str
                  prev))
            ""
            '("x" "abc" "xyz" "jk"))
⇒ "xyz"
```

If *lst1* through *lstN* have different lengths, `fold` stops when the end of the shortest is reached; `fold-right` commences at the last element of the shortest. Ie. elements past the length of the shortest are ignored in the other *lst*s. At least one *lst* must be non-circular.

`fold` should be preferred over `fold-right` if the order of processing doesn't matter, or can be arranged either way, since `fold` is a little more efficient.

The way `fold` builds a result from iterating is quite general, it can do more than other iterations like say `map` or `filter`. The following for example removes adjacent duplicate elements from a list,

```
(define (delete-adjacent-duplicates lst)
  (fold-right (lambda (elem ret)
                (if (equal? elem (first ret))
                    ret
                    (cons elem ret)))
              (list (last lst))
              lst))
(delete-adjacent-duplicates '(1 2 3 3 4 4 4 5))
⇒ (1 2 3 4 5)
```

Clearly the same sort of thing can be done with a `for-each` and a variable in which to build the result, but a self-contained *proc* can be re-used in multiple contexts, where a `for-each` would have to be written out each time.

`pair-fold` *proc init lst1 . . . lstN* [Scheme Procedure]
`pair-fold-right` *proc init lst1 . . . lstN* [Scheme Procedure]

The same as `fold` and `fold-right`, but apply *proc* to the pairs of the lists instead of the list elements.

`reduce` *proc default lst* [Scheme Procedure]
`reduce-right` *proc default lst* [Scheme Procedure]

`reduce` is a variant of `fold`, where the first call to *proc* is on two elements from *lst*, rather than one element and a given initial value.

If *lst* is empty, `reduce` returns *default* (this is the only use for *default*). If *lst* has just one element then that's the return value. Otherwise *proc* is called on the elements of *lst*.

Each *proc* call is `(proc elem previous)`, where *elem* is from *lst* (the second and subsequent elements of *lst*), and *previous* is the return from the previous call to *proc*. The first element of *lst* is the *previous* for the first call to *proc*.

For example, the following adds a list of numbers, the calls made to `+` are shown. (Of course `+` accepts multiple arguments and can add a list directly, with `apply`.)

```
(reduce + 0 '(5 6 7)) ⇒ 18

(+ 6 5)  ⇒ 11
(+ 7 11) ⇒ 18
```

`reduce` can be used instead of `fold` where the *init* value is an "identity", meaning a value which under *proc* doesn't change the result, in this case 0 is an identity since `(+ 5 0)` is just 5. `reduce` avoids that unnecessary call.

`reduce-right` is a similar variation on `fold-right`, working from the end (ie. the right) of *lst*. The last element of *lst* is the *previous* for the first call to *proc*, and the *elem* values go from the second last.

`reduce` should be preferred over `reduce-right` if the order of processing doesn't matter, or can be arranged either way, since `reduce` is a little more efficient.

`unfold` *p f g seed* [*tail-gen*] [Scheme Procedure]

`unfold` is defined as follows:

```
(unfold p f g seed) =
   (if (p seed) (tail-gen seed)
       (cons (f seed)
             (unfold p f g (g seed))))
```

p	Determines when to stop unfolding.
f	Maps each seed value to the corresponding list element.
g	Maps each seed value to next seed valu.
seed	The state value for the unfold.
tail-gen	Creates the tail of the list; defaults to `(lambda (x) '())`.

g produces a series of seed values, which are mapped to list elements by *f*. These elements are put into a list in left-to-right order, and *p* tells when to stop unfolding.

`unfold-right` *p f g seed* [*tail*] [Scheme Procedure]

Construct a list with the following loop.

```
(let lp ((seed seed) (lis tail))
   (if (p seed) lis
       (lp (g seed)
           (cons (f seed) lis))))
```

p	Determines when to stop unfolding.
f	Maps each seed value to the corresponding list element.
g	Maps each seed value to next seed valu.
seed	The state value for the unfold.
tail-gen	Creates the tail of the list; defaults to `(lambda (x) '())`.

`map` *f lst1 lst2 ...* [Scheme Procedure]

Map the procedure over the list(s) *lst1*, *lst2*, ... and return a list containing the results of the procedure applications. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The result list will have the same length as the shortest argument lists. The order in which *f* will be applied to the list element(s) is not specified.

`for-each` *f lst1 lst2 ...* [Scheme Procedure]

Apply the procedure *f* to each pair of corresponding elements of the list(s) *lst1*, *lst2*, ... The return value is not specified. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The shortest argument list determines the number of times *f* is called. *f* will be applied to the list elements in left-to-right order.

`append-map` *f lst1 lst2 . . .* [Scheme Procedure]
`append-map!` *f lst1 lst2 . . .* [Scheme Procedure]

Equivalent to

```
(apply append (map f clist1 clist2 ...))
```

and

```
(apply append! (map f clist1 clist2 ...))
```

Map *f* over the elements of the lists, just as in the `map` function. However, the results of the applications are appended together to make the final result. `append-map` uses `append` to append the results together; `append-map!` uses `append!`.

The dynamic order in which the various applications of *f* are made is not specified.

`map!` *f lst1 lst2 . . .* [Scheme Procedure]

Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of *lst1* to construct the result list.

The dynamic order in which the various applications of *f* are made is not specified. In the n-ary case, *lst2*, *lst3*, . . . must have at least as many elements as *lst1*.

`pair-for-each` *f lst1 lst2 . . .* [Scheme Procedure]

Like `for-each`, but applies the procedure *f* to the pairs from which the argument lists are constructed, instead of the list elements. The return value is not specified.

`filter-map` *f lst1 lst2 . . .* [Scheme Procedure]

Like `map`, but only results from the applications of *f* which are true are saved in the result list.

6.4.3.6 Filtering and Partitioning

Filtering means to collect all elements from a list which satisfy a specific condition. Partitioning a list means to make two groups of list elements, one which contains the elements satisfying a condition, and the other for the elements which don't.

The `filter` and `filter!` functions are implemented in the Guile core, See Section 5.6.2.6 [List Modification], page 172.

`partition` *pred lst* [Scheme Procedure]
`partition!` *pred lst* [Scheme Procedure]

Split *lst* into those elements which do and don't satisfy the predicate *pred*.

The return is two values (see Section 5.11.6 [Multiple Values], page 257), the first being a list of all elements from *lst* which satisfy *pred*, the second a list of those which do not.

The elements in the result lists are in the same order as in *lst* but the order in which the calls `(`*`pred`*` elem)` are made on the list elements is unspecified.

`partition` does not change *lst*, but one of the returned lists may share a tail with it. `partition!` may modify *lst* to construct its return.

`remove` *pred lst* [Scheme Procedure]

`remove!` *pred lst* [Scheme Procedure]

Return a list containing all elements from *lst* which do not satisfy the predicate *pred*. The elements in the result list have the same order as in *lst*. The order in which *pred* is applied to the list elements is not specified.

`remove!` is allowed, but not required to modify the structure of the input list.

6.4.3.7 Searching

The procedures for searching elements in lists either accept a predicate or a comparison object for determining which elements are to be searched.

`find` *pred lst* [Scheme Procedure]

Return the first element of *lst* which satisfies the predicate *pred* and `#f` if no such element is found.

`find-tail` *pred lst* [Scheme Procedure]

Return the first pair of *lst* whose CAR satisfies the predicate *pred* and `#f` if no such element is found.

`take-while` *pred lst* [Scheme Procedure]
`take-while!` *pred lst* [Scheme Procedure]

Return the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`take-while!` is allowed, but not required to modify the input list while producing the result.

`drop-while` *pred lst* [Scheme Procedure]

Drop the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`span` *pred lst* [Scheme Procedure]
`span!` *pred lst* [Scheme Procedure]
`break` *pred lst* [Scheme Procedure]
`break!` *pred lst* [Scheme Procedure]

`span` splits the list *lst* into the longest initial prefix whose elements all satisfy the predicate *pred*, and the remaining tail. `break` inverts the sense of the predicate.

`span!` and `break!` are allowed, but not required to modify the structure of the input list *lst* in order to produce the result.

Note that the name `break` conflicts with the `break` binding established by `while` (see Section 5.11.4 [while do], page 252). Applications wanting to use `break` from within a `while` loop will need to make a new define under a different name.

`any` *pred lst1 lst2 . . . lstN* [Scheme Procedure]

Test whether any set of elements from *lst1* . . . lstN satisfies *pred*. If so the return value is the return from the successful *pred* call, or if not the return is `#f`.

Each *pred* call is `(pred elem1 ... elemN)` taking an element from each *lst*. The calls are made successively for the first, second, etc elements of the lists, stopping when *pred* returns non-`#f`, or when the end of the shortest list is reached.

The *pred* call on the last set of elements (ie. when the end of the shortest list has been reached), if that point is reached, is a tail call.

`every` *pred lst1 lst2 . . . lstN* [Scheme Procedure]

Test whether every set of elements from *lst1* . . . lstN satisfies *pred*. If so the return value is the return from the final *pred* call, or if not the return is `#f`.

Each *pred* call is `(pred elem1 ... elemN)` taking an element from each *lst*. The calls are made successively for the first, second, etc elements of the lists, stopping if *pred* returns `#f`, or when the end of any of the lists is reached.

The *pred* call on the last set of elements (ie. when the end of the shortest list has been reached) is a tail call.

If one of *lst1* . . . *lstN* is empty then no calls to *pred* are made, and the return is `#t`.

`list-index` *pred lst1 . . . lstN* [Scheme Procedure]

Return the index of the first set of elements, one from each of *lst1*. . .*lstN*, which satisfies *pred*.

pred is called as `(pred elem1 ... elemN)`. Searching stops when the end of the shortest *lst* is reached. The return index starts from 0 for the first set of elements. If no set of elements pass then the return is `#f`.

```
(list-index odd? '(2 4 6 9))      ⇒ 3
(list-index = '(1 2 3) '(3 1 2))  ⇒ #f
```

`member` *x lst* [=] [Scheme Procedure]

Return the first sublist of *lst* whose CAR is equal to *x*. If *x* does not appear in *lst*, return `#f`.

Equality is determined by `equal?`, or by the equality predicate = if given. = is called `(= x elem)`, ie. with the given *x* first, so for example to find the first element greater than 5,

```
(member 5 '(3 5 1 7 2 9) <) ⇒ (7 2 9)
```

This version of `member` extends the core `member` (see Section 5.6.2.7 [List Searching], page 173) by accepting an equality predicate.

6.4.3.8 Deleting

`delete` *x lst* [=] [Scheme Procedure]
`delete!` *x lst* [=] [Scheme Procedure]

Return a list containing the elements of *lst* but with those equal to *x* deleted. The returned elements will be in the same order as they were in *lst*.

Equality is determined by the = predicate, or `equal?` if not given. An equality call is made just once for each element, but the order in which the calls are made on the elements is unspecified.

The equality calls are always `(= x elem)`, ie. the given *x* is first. This means for instance elements greater than 5 can be deleted with `(delete 5 lst <)`.

`delete` does not modify *lst*, but the return might share a common tail with *lst*. `delete!` may modify the structure of *lst* to construct its return.

These functions extend the core `delete` and `delete!` (see Section 5.6.2.6 [List Modification], page 172) in accepting an equality predicate. See also `lset-difference` (see Section 6.4.3.10 [SRFI-1 Set Operations], page 435) for deleting multiple elements from a list.

`delete-duplicates` *lst* [=] [Scheme Procedure]
`delete-duplicates!` *lst* [=] [Scheme Procedure]

Return a list containing the elements of *lst* but without duplicates.

When elements are equal, only the first in *lst* is retained. Equal elements can be anywhere in *lst*, they don't have to be adjacent. The returned list will have the retained elements in the same order as they were in *lst*.

Equality is determined by the = predicate, or `equal?` if not given. Calls `(= x y)` are made with element *x* being before *y* in *lst*. A call is made at most once for each combination, but the sequence of the calls across the elements is unspecified.

`delete-duplicates` does not modify *lst*, but the return might share a common tail with *lst*. `delete-duplicates!` may modify the structure of *lst* to construct its return.

In the worst case, this is an $O(N^2)$ algorithm because it must check each element against all those preceding it. For long lists it is more efficient to sort and then compare only adjacent elements.

6.4.3.9 Association Lists

Association lists are described in detail in section Section 5.6.11 [Association Lists], page 210. The present section only documents the additional procedures for dealing with association lists defined by SRFI-1.

`assoc` *key alist* [=] [Scheme Procedure]

Return the pair from *alist* which matches *key*. This extends the core `assoc` (see Section 5.6.11.3 [Retrieving Alist Entries], page 212) by taking an optional = comparison procedure.

The default comparison is `equal?`. If an = parameter is given it's called `(=` ***key alistcar***`)`, ie. the given target *key* is the first argument, and a `car` from *alist* is second.

For example a case-insensitive string lookup,

```
(assoc "yy" '(("XX" . 1) ("YY" . 2)) string-ci=?)
⇒ ("YY" . 2)
```

`alist-cons` *key datum alist* [Scheme Procedure]

Cons a new association *key* and *datum* onto *alist* and return the result. This is equivalent to

```
(cons (cons key datum) alist)
```

`acons` (see Section 5.6.11.2 [Adding or Setting Alist Entries], page 210) in the Guile core does the same thing.

`alist-copy` *alist* [Scheme Procedure]

Return a newly allocated copy of *alist*, that means that the spine of the list as well as the pairs are copied.

`alist-delete` *key alist* [=] [Scheme Procedure]
`alist-delete!` *key alist* [=] [Scheme Procedure]

Return a list containing the elements of *alist* but with those elements whose keys are equal to *key* deleted. The returned elements will be in the same order as they were in *alist*.

Equality is determined by the = predicate, or `equal?` if not given. The order in which elements are tested is unspecified, but each equality call is made `(= key alistkey)`, ie. the given *key* parameter is first and the key from *alist* second. This means for instance all associations with a key greater than 5 can be removed with `(alist-delete 5 alist <)`.

`alist-delete` does not modify *alist*, but the return might share a common tail with *alist*. `alist-delete!` may modify the list structure of *alist* to construct its return.

6.4.3.10 Set Operations on Lists

Lists can be used to represent sets of objects. The procedures in this section operate on such lists as sets.

Note that lists are not an efficient way to implement large sets. The procedures here typically take time $m \times n$ when operating on m and n element lists. Other data structures like trees, bitsets (see Section 5.6.5 [Bit Vectors], page 187) or hash tables (see Section 5.6.12 [Hash Tables], page 215) are faster.

All these procedures take an equality predicate as the first argument. This predicate is used for testing the objects in the list sets for sameness. This predicate must be consistent with `eq?` (see Section 5.9.1 [Equality], page 236) in the sense that if two list elements are `eq?` then they must also be equal under the predicate. This simply means a given object must be equal to itself.

`lset<=` = *list1 list2 . . .* [Scheme Procedure]

Return `#t` if each list is a subset of the one following it. Ie. *list1* a subset of *list2*, *list2* a subset of *list3*, etc, for as many lists as given. If only one list or no lists are given then the return is `#t`.

A list *x* is a subset of *y* if each element of *x* is equal to some element in *y*. Elements are compared using the given = procedure, called as `(= xelem yelem)`.

```
(lset<= eq?)                    ⇒ #t
(lset<= eqv? '(1 2 3) '(1))     ⇒ #f
(lset<= eqv? '(1 3 2) '(4 3 1 2)) ⇒ #t
```

`lset=` = *list1 list2 . . .* [Scheme Procedure]

Return `#t` if all argument lists are set-equal. *list1* is compared to *list2*, *list2* to *list3*, etc, for as many lists as given. If only one list or no lists are given then the return is `#t`.

Two lists *x* and *y* are set-equal if each element of *x* is equal to some element of *y* and conversely each element of *y* is equal to some element of *x*. The order of the elements in the lists doesn't matter. Element equality is determined with the given = procedure, called as `(= xelem yelem)`, but exactly which calls are made is unspecified.

```
(lset= eq?)                     ⇒ #t
(lset= eqv? '(1 2 3) '(3 2 1))  ⇒ #t
(lset= string-ci=? '("a" "A" "b") '("B" "b" "a")) ⇒ #t
```

`lset-adjoin` = *list elem1 . . .* [Scheme Procedure]

Add to *list* any of the given *elem*s not already in the list. *elem*s are `cons`ed onto the start of *list* (so the return shares a common tail with *list*), but the order they're added is unspecified.

The given = procedure is used for comparing elements, called as `(= listelem elem)`, ie. the second argument is one of the given *elem* parameters.

```
(lset-adjoin eqv? '(1 2 3) 4 1 5) ⇒ (5 4 1 2 3)
```

`lset-union` = *list1 list2 . . .* [Scheme Procedure]
`lset-union!` = *list1 list2 . . .* [Scheme Procedure]

Return the union of the argument list sets. The result is built by taking the union of *list1* and *list2*, then the union of that with *list3*, etc, for as many lists as given. For one list argument that list itself is the result, for no list arguments the result is the empty list.

The union of two lists *x* and *y* is formed as follows. If *x* is empty then the result is *y*. Otherwise start with *x* as the result and consider each *y* element (from first to last). A *y* element not equal to something already in the result is `cons`ed onto the result.

The given = procedure is used for comparing elements, called as `(= relem yelem)`. The first argument is from the result accumulated so far, and the second is from the list being union-ed in. But exactly which calls are made is otherwise unspecified.

Notice that duplicate elements in *list1* (or the first non-empty list) are preserved, but that repeated elements in subsequent lists are only added once.

```
(lset-union eqv?)                        ⇒ ()
(lset-union eqv? '(1 2 3))               ⇒ (1 2 3)
(lset-union eqv? '(1 2 1 3) '(2 4 5) '(5)) ⇒ (5 4 1 2 1 3)
```

`lset-union` doesn't change the given lists but the result may share a tail with the first non-empty list. `lset-union!` can modify all of the given lists to form the result.

`lset-intersection` = *list1 list2 . . .* [Scheme Procedure]
`lset-intersection!` = *list1 list2 . . .* [Scheme Procedure]

Return the intersection of *list1* with the other argument lists, meaning those elements of *list1* which are also in all of *list2* etc. For one list argument, just that list is returned.

The test for an element of *list1* to be in the return is simply that it's equal to some element in each of *list2* etc. Notice this means an element appearing twice in *list1* but only once in each of *list2* etc will go into the return twice. The return has its elements in the same order as they were in *list1*.

The given = procedure is used for comparing elements, called as `(= elem1 elemN)`. The first argument is from *list1* and the second is from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

```
(lset-intersection eqv? '(x y))                        ⇒ (x y)
(lset-intersection eqv? '(1 2 3) '(4 3 2))             ⇒ (2 3)
(lset-intersection eqv? '(1 1 2 2) '(1 2) '(2 1) '(2)) ⇒ (2 2)
```

The return from `lset-intersection` may share a tail with *list1*. `lset-intersection!` may modify *list1* to form its result.

`lset-difference` = *list1 list2 . . .* [Scheme Procedure]
`lset-difference!` = *list1 list2 . . .* [Scheme Procedure]

Return *list1* with any elements in *list2*, *list3* etc removed (ie. subtracted). For one list argument, just that list is returned.

The given = procedure is used for comparing elements, called as `(= elem1 elemN)`. The first argument is from *list1* and the second from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

```
(lset-difference eqv? '(x y))             ⇒ (x y)
(lset-difference eqv? '(1 2 3) '(3 1))    ⇒ (2)
(lset-difference eqv? '(1 2 3) '(3) '(2)) ⇒ (1)
```

The return from `lset-difference` may share a tail with *list1*. `lset-difference!` may modify *list1* to form its result.

`lset-diff+intersection` = *list1 list2 . . .* [Scheme Procedure]
`lset-diff+intersection!` = *list1 list2 . . .* [Scheme Procedure]

Return two values (see Section 5.11.6 [Multiple Values], page 257), the difference and intersection of the argument lists as per `lset-difference` and `lset-intersection` above.

For two list arguments this partitions *list1* into those elements of *list1* which are in *list2* and not in *list2*. (But for more than two arguments there can be elements of *list1* which are neither part of the difference nor the intersection.)

One of the return values from `lset-diff+intersection` may share a tail with *list1*. `lset-diff+intersection!` may modify *list1* to form its results.

`lset-xor` = *list1 list2 . . .* [Scheme Procedure]
`lset-xor!` = *list1 list2 . . .* [Scheme Procedure]

Return an XOR of the argument lists. For two lists this means those elements which are in exactly one of the lists. For more than two lists it means those elements which appear in an odd number of the lists.

To be precise, the XOR of two lists *x* and *y* is formed by taking those elements of *x* not equal to any element of *y*, plus those elements of *y* not equal to any element of *x*. Equality is determined with the given = procedure, called as `(= e1 e2)`. One argument is from *x* and the other from *y*, but which way around is unspecified. Exactly which calls are made is also unspecified, as is the order of the elements in the result.

```
(lset-xor eqv? '(x y))             ⇒ (x y)
(lset-xor eqv? '(1 2 3) '(4 3 2))  ⇒ (4 1)
```

The return from `lset-xor` may share a tail with one of the list arguments. `lset-xor!` may modify *list1* to form its result.

6.4.4 SRFI-2 - and-let*

The following syntax can be obtained with

```
(use-modules (srfi srfi-2))
```

`and-let*` (*clause . . .*) *body . . .* [library syntax]

A combination of `and` and `let*`.

Each *clause* is evaluated in turn, and if `#f` is obtained then evaluation stops and `#f` is returned. If all are non-`#f` then *body* is evaluated and the last form gives the return value, or if *body* is empty then the result is `#t`. Each *clause* should be one of the following,

`(symbol expr)`
: Evaluate *expr*, check for `#f`, and bind it to *symbol*. Like `let*`, that binding is available to subsequent clauses.

`(expr)`
: Evaluate *expr* and check for `#f`.

`symbol`
: Get the value bound to *symbol* and check for `#f`.

Notice that `(expr)` has an "extra" pair of parentheses, for instance `((eq? x y))`. One way to remember this is to imagine the `symbol` in `(symbol expr)` is omitted.

`and-let*` is good for calculations where a `#f` value means termination, but where a non-`#f` value is going to be needed in subsequent expressions.

The following illustrates this, it returns text between brackets '`[...]`' in a string, or `#f` if there are no such brackets (ie. either `string-index` gives `#f`).

```
(define (extract-brackets str)
  (and-let* ((start (string-index str #\[))
             (end   (string-index str #\] start)))
    (substring str (1+ start) end)))
```

The following shows plain variables and expressions tested too. `diagnostic-levels` is taken to be an alist associating a diagnostic type with a level. `str` is printed only if the type is known and its level is high enough.

```
(define (show-diagnostic type str)
  (and-let* (want-diagnostics
             (level (assq-ref diagnostic-levels type))
             ((>= level current-diagnostic-level)))
    (display str)))
```

The advantage of `and-let*` is that an extended sequence of expressions and tests doesn't require lots of nesting as would arise from separate `and` and `let*`, or from `cond` with `=>`.

6.4.5 SRFI-4 - Homogeneous numeric vector datatypes

The SRFI-4 procedures and data types are always available, See Section 5.6.4 [Uniform Numeric Vectors], page 178.

6.4.6 SRFI-6 - Basic String Ports

SRFI-6 defines the procedures `open-input-string`, `open-output-string` and `get-output-string`. These procedures are included in the Guile core, so using this module does not make any difference at the moment. But it is possible that support for SRFI-6 will be factored out of the core library in the future, so using this module does not hurt, after all.

6.4.7 SRFI-8 - receive

`receive` is a syntax for making the handling of multiple-value procedures easier. It is documented in See Section 5.11.6 [Multiple Values], page 257.

6.4.8 SRFI-9 - define-record-type

This SRFI is a syntax for defining new record types and creating predicate, constructor, and field getter and setter functions. In Guile this is simply an alternate interface to the core record functionality (see Section 5.6.8 [Records], page 204). It can be used with,

```
(use-modules (srfi srfi-9))
```

`define-record-type` *type* [library syntax]
    (*constructor fieldname* . . .)
    predicate
    (*fieldname accessor* [*modifier*]) . . .

> Create a new record type, and make various `define`s for using it. This syntax can only occur at the top-level, not nested within some other form.
>
> *type* is bound to the record type, which is as per the return from the core `make-record-type`. *type* also provides the name for the record, as per `record-type-name`.
>
> *constructor* is bound to a function to be called as `(constructor fieldval ...)` to create a new record of this type. The arguments are initial values for the fields, one argument for each field, in the order they appear in the `define-record-type` form.
>
> The *fieldname*s provide the names for the record fields, as per the core `record-type-fields` etc, and are referred to in the subsequent accessor/modifier forms.
>
> *predictate* is bound to a function to be called as `(predicate obj)`. It returns `#t` or `#f` according to whether *obj* is a record of this type.
>
> Each *accessor* is bound to a function to be called `(accessor record)` to retrieve the respective field from a *record*. Similarly each *modifier* is bound to a function to be called `(modifier record val)` to set the respective field in a *record*.

An example will illustrate typical usage,

```
(define-record-type employee-type
  (make-employee name age salary)
  employee?
  (name    get-employee-name)
  (age     get-employee-age    set-employee-age)
  (salary  get-employee-salary set-employee-salary))
```

This creates a new employee data type, with name, age and salary fields. Accessor functions are created for each field, but no modifier function for the name (the intention in this example being that it's established only when an employee object is created). These can all then be used as for example,

```
employee-type ⇒ #<record-type employee-type>

(define fred (make-employee "Fred" 45 20000.00))
```

```
(employee? fred)        ⇒ #t
(get-employee-age fred) ⇒ 45
(set-employee-salary fred 25000.00)  ;; pay rise
```

The functions created by `define-record-type` are ordinary top-level `define`s. They can be redefined or `set!` as desired, exported from a module, etc.

6.4.9 SRFI-10 - Hash-Comma Reader Extension

This SRFI implements a reader extension `#,()` called hash-comma. It allows the reader to give new kinds of objects, for use both in data and as constants or literals in source code. This feature is available with

```
(use-modules (srfi srfi-10))
```

The new read syntax is of the form

```
#,(tag arg...)
```

where *tag* is a symbol and the *arg*s are objects taken as parameters. *tag*s are registered with the following procedure.

`define-reader-ctor` *tag proc* [Scheme Procedure]

Register *proc* as the constructor for a hash-comma read syntax starting with symbol *tag*, ie. `#,(`*tag* `arg...)`. *proc* is called with the given arguments `(`*proc* `arg...)` and the object it returns is the result of the read.

For example, a syntax giving a list of *N* copies of an object.

```
(define-reader-ctor 'repeat
  (lambda (obj reps)
    (make-list reps obj)))

(display '#,(repeat 99 3))
⊣ (99 99 99)
```

Notice the quote ' when the `#,( )` is used. The `repeat` handler returns a list and the program must quote to use it literally, the same as any other list. Ie.

```
(display '#,(repeat 99 3))
⇒
(display '(99 99 99))
```

When a handler returns an object which is self-evaluating, like a number or a string, then there's no need for quoting, just as there's no need when giving those directly as literals. For example an addition,

```
(define-reader-ctor 'sum
  (lambda (x y)
    (+ x y)))
(display #,(sum 123 456)) ⊣ 579
```

A typical use for `#,()` is to get a read syntax for objects which don't otherwise have one. For example, the following allows a hash table to be given literally, with tags and values, ready for fast lookup.

```
(define-reader-ctor 'hash
  (lambda elems
        (let ((table (make-hash-table)))
          (for-each (lambda (elem)
                      (apply hash-set! table elem))
                    elems)
          table)))

    (define (animal->family animal)
      (hash-ref '#,(hash ("tiger" "cat")
                         ("lion"  "cat")
                         ("wolf"  "dog"))
                animal))

    (animal->family "lion") ⇒ "cat"
```

Or for example the following is a syntax for a compiled regular expression (see Section 5.5.6 [Regular Expressions], page 146).

```
    (use-modules (ice-9 regex))

    (define-reader-ctor 'regexp make-regexp)

    (define (extract-angs str)
      (let ((match (regexp-exec '#,(regexp "<([A-Z0-9]+)>") str)))
        (and match
             (match:substring match 1))))

    (extract-angs "foo <BAR> quux") ⇒ "BAR"
```

`#,()` is somewhat similar to `define-macro` (see Section 5.8.6 [Macros], page 231) in that handler code is run to produce a result, but `#,()` operates at the read stage, so it can appear in data for `read` (see Section 5.13.2 [Scheme Read], page 290), not just in code to be executed.

Because `#,()` is handled at read-time it has no direct access to variables etc. A symbol in the arguments is just a symbol, not a variable reference. The arguments are essentially constants, though the handler procedure can use them in any complicated way it might want.

Once `(srfi srfi-10)` has loaded, `#,()` is available globally, there's no need to use `(srfi srfi-10)` in later modules. Similarly the tags registered are global and can be used anywhere once registered.

There's no attempt to record what previous `#,()` forms have been seen, if two identical forms occur then two calls are made to the handler procedure. The handler might like to maintain a cache or similar to avoid making copies of large objects, depending on expected usage.

In code the best uses of `#,()` are generally when there's a lot of objects of a particular kind as literals or constants. If there's just a few then some local variables and initializers are fine, but that becomes tedious and error prone when there's a lot, and the anonymous and compact syntax of `#,()` is much better.

6.4.10 SRFI-11 - let-values

This module implements the binding forms for multiple values `let-values` and `let-values*`. These forms are similar to `let` and `let*` (see Section 5.10.2 [Local Bindings], page 248), but they support binding of the values returned by multiple-valued expressions.

Write `(use-modules (srfi srfi-11))` to make the bindings available.

```
(let-values (((x y) (values 1 2))
             ((z f) (values 3 4)))
   (+ x y z f))
⇒
10
```

`let-values` performs all bindings simultaneously, which means that no expression in the binding clauses may refer to variables bound in the same clause list. `let-values*`, on the other hand, performs the bindings sequentially, just like `let*` does for single-valued expressions.

6.4.11 SRFI-13 - String Library

The SRFI-13 procedures are always available, See Section 5.5.5 [Strings], page 129.

6.4.12 SRFI-14 - Character-set Library

The SRFI-14 data type and procedures are always available, See Section 5.5.4 [Character Sets], page 123.

6.4.13 SRFI-16 - case-lambda

The syntactic form `case-lambda` creates procedures, just like `lambda`, but has syntactic extensions for writing procedures of varying arity easier.

The syntax of the `case-lambda` form is defined in the following EBNF grammar.

```
<case-lambda>
   --> (case-lambda <case-lambda-clause>)
<case-lambda-clause>
   --> (<formals> <definition-or-command>*)
<formals>
   --> (<identifier>*)
     | (<identifier>* . <identifier>)
     | <identifier>
```

The value returned by a `case-lambda` form is a procedure which matches the number of actual arguments against the formals in the various clauses, in order. *Formals* means a formal argument list just like with `lambda` (see Section 5.8.1 [Lambda], page 225). The first matching clause is selected, the corresponding values from the actual parameter list are bound to the variable names in the clauses and the body of the clause is evaluated. If no clause matches, an error is signalled.

The following (silly) definition creates a procedure *foo* which acts differently, depending on the number of actual arguments. If one argument is given, the constant `#t` is returned, two arguments are added and if more arguments are passed, their product is calculated.

```
(define foo (case-lambda
              ((x) #t)
              ((x y) (+ x y))
              (z
                (apply * z))))
(foo 'bar)
⇒
#t
(foo 2 4)
⇒
6
(foo 3 3 3)
⇒
27
(foo)
⇒
1
```

The last expression evaluates to 1 because the last clause is matched, *z* is bound to the empty list and the following multiplication, applied to zero arguments, yields 1.

6.4.14 SRFI-17 - Generalized set!

This is an implementation of SRFI-17: Generalized set!

It exports the Guile procedure `make-procedure-with-setter` under the SRFI name `getter-with-setter` and exports the standard procedures `car`, `cdr`, ..., `cdddr`, `string-ref` and `vector-ref` as procedures with setters, as required by the SRFI.

SRFI-17 was heavily criticized during its discussion period but it was finalized anyway. One issue was its concept of globally associating setter *properties* with (procedure) values, which is non-Schemy. For this reason, this implementation chooses not to provide a way to set the setter of a procedure. In fact, `(set! (setter` *proc*`)` *setter*`)` signals an error. The only way to attach a setter to a procedure is to create a new object (a *procedure with setter*) via the `getter-with-setter` procedure. This procedure is also specified in the SRFI. Using it avoids the described problems.

6.4.15 SRFI-19 - Time/Date Library

This is an implementation of the SRFI-19 time/date library. The functions and variables described here are provided by

```
(use-modules (srfi srfi-19))
```

Caution: The current code in this module incorrectly extends the Gregorian calendar leap year rule back prior to the introduction of those reforms in 1582 (or the appropriate year in various countries). The Julian calendar was used prior to 1582, and there were 10 days skipped for the reform, but the code doesn't implement that.

This will be fixed some time. Until then calculations for 1583 onwards are correct, but prior to that any day/month/year and day of the week calculations are wrong.

6.4.15.1 SRFI-19 Introduction

This module implements time and date representations and calculations, in various time systems, including universal time (UTC) and atomic time (TAI).

For those not familiar with these time systems, TAI is based on a fixed length second derived from oscillations of certain atoms. UTC differs from TAI by an integral number of seconds, which is increased or decreased at announced times to keep UTC aligned to a mean solar day (the orbit and rotation of the earth are not quite constant).

So far, only increases in the TAI $\leftrightarrow$ UTC difference have been needed. Such an increase is a "leap second", an extra second of TAI introduced at the end of a UTC day. When working entirely within UTC this is never seen, every day simply has 86400 seconds. But when converting from TAI to a UTC date, an extra 23:59:60 is present, where normally a day would end at 23:59:59. Effectively the UTC second from 23:59:59 to 00:00:00 has taken two TAI seconds.

In the current implementation, the system clock is assumed to be UTC, and a table of leap seconds in the code converts to TAI. See comments in '`srfi-19.scm`' for how to update this table.

Also, for those not familiar with the terminology, a *Julian Day* is a real number which is a count of days and fraction of a day, in UTC, starting from -4713-01-01T12:00:00Z, ie. midday Monday 1 Jan 4713 B.C. A *Modified Julian Day* is the same, but starting from 1858-11-17T00:00:00Z, ie. midnight 17 November 1858 UTC. That time is julian day 2400000.5.

6.4.15.2 SRFI-19 Time

A *time* object has type, seconds and nanoseconds fields representing a point in time starting from some epoch. This is an arbitrary point in time, not just a time of day. Although times are represented in nanoseconds, the actual resolution may be lower.

The following variables hold the possible time types. For instance `(current-time time-process)` would give the current CPU process time.

`time-utc` [Variable]

Universal Coordinated Time (UTC).

`time-tai` [Variable]

International Atomic Time (TAI).

`time-monotonic` [Variable]

Monotonic time, meaning a monotonically increasing time starting from an unspecified epoch.

Note that in the current implementation `time-monotonic` is the same as `time-tai`, and unfortunately is therefore affected by adjustments to the system clock. Perhaps this will change in the future.

`time-duration` [Variable]

A duration, meaning simply a difference between two times.

`time-process` [Variable]

CPU time spent in the current process, starting from when the process began.

`time-thread` [Variable]
CPU time spent in the current thread. Not currently implemented.

`time?` *obj* [Function]
Return `#t` if *obj* is a time object, or `#f` if not.

`make-time` *type nanoseconds seconds* [Function]
Create a time object with the given *type*, *seconds* and *nanoseconds*.

`time-type` *time* [Function]
`time-nanosecond` *time* [Function]
`time-second` *time* [Function]
`set-time-type!` *time type* [Function]
`set-time-nanosecond!` *time nsec* [Function]
`set-time-second!` *time sec* [Function]
Get or set the type, seconds or nanoseconds fields of a time object.

`set-time-type!` merely changes the field, it doesn't convert the time value. For conversions, see Section 6.4.15.4 [SRFI-19 Time/Date conversions], page 447.

`copy-time` *time* [Function]
Return a new time object, which is a copy of the given *time*.

`current-time` [*type*] [Function]
Return the current time of the given *type*. The default *type* is `time-utc`.

Note that the name `current-time` conflicts with the Guile core `current-time` function (see Section 6.2.5 [Time], page 389). Applications wanting to use both will need to use a different name for one of them.

`time-resolution` [*type*] [Function]
Return the resolution, in nanoseconds, of the given time *type*. The default *type* is `time-utc`.

`time<=?` *t1 t2* [Function]
`time<?` *t1 t2* [Function]
`time=?` *t1 t2* [Function]
`time>=?` *t1 t2* [Function]
`time>?` *t1 t2* [Function]
Return `#t` or `#f` according to the respective relation between time objects *t1* and *t2*. *t1* and *t2* must be the same time type.

`time-difference` *t1 t2* [Function]
`time-difference!` *t1 t2* [Function]
Return a time object of type `time-duration` representing the period between *t1* and *t2*. *t1* and *t2* must be the same time type.

`time-difference` returns a new time object, `time-difference!` may modify *t1* to form its return.

`add-duration` *time duration* [Function]
`add-duration!` *time duration* [Function]
`subtract-duration` *time duration* [Function]
`subtract-duration!` *time duration* [Function]

Return a time object which is *time* with the given *duration* added or subtracted. *duration* must be a time object of type `time-duration`.

`add-duration` and `subtract-duration` return a new time object. `add-duration!` and `subtract-duration!` may modify the given *time* to form their return.

6.4.15.3 SRFI-19 Date

A *date* object represents a date in the Gregorian calendar and a time of day on that date in some timezone.

The fields are year, month, day, hour, minute, second, nanoseconds and timezone. A date object is immutable, its fields can be read but they cannot be modified once the object is created.

`date?` *obj* [Function]

Return `#t` if *obj* is a date object, or `#f` if not.

`make-date` *nsecs seconds minutes hours date month year zone-offset* [Function]

Create a new date object.

`date-nanosecond` *date* [Function]

Nanoseconds, 0 to 999999999.

`date-second` *date* [Function]

Seconds, 0 to 59, or 60 for a leap second. 60 is never seen when working entirely within UTC, it's only when converting to or from TAI.

`date-minute` *date* [Function]

Minutes, 0 to 59.

`date-hour` *date* [Function]

Hour, 0 to 23.

`date-day` *date* [Function]

Day of the month, 1 to 31 (or less, according to the month).

`date-month` *date* [Function]

Month, 1 to 12.

`date-year` *date* [Function]

Year, eg. 2003. Dates B.C. are negative, eg. −46 is 46 B.C. There is no year 0, year −1 is followed by year 1.

`date-zone-offset` *date* [Function]

Time zone, an integer number of seconds east of Greenwich.

`date-year-day` *date* [Function]

Day of the year, starting from 1 for 1st January.

`date-week-day` *date* [Function]
  Day of the week, starting from 0 for Sunday.

`date-week-number` *date dstartw* [Function]
  Week of the year, ignoring a first partial week. *dstartw* is the day of the week which is taken to start a week, 0 for Sunday, 1 for Monday, etc.

`current-date` [*tz-offset*] [Function]
  Return a date object representing the current date/time, in UTC offset by *tz-offset*. *tz-offset* is seconds east of Greenwich and defaults to the local timezone.

`current-julian-day` [Function]
  Return the current Julian Day.

`current-modified-julian-day` [Function]
  Return the current Modified Julian Day.

6.4.15.4 SRFI-19 Time/Date conversions

`date->julian-day` *date* [Function]
`date->modified-julian-day` *date* [Function]
`date->time-monotonic` *date* [Function]
`date->time-tai` *date* [Function]
`date->time-utc` *date* [Function]

`julian-day->date` *jdn* [*tz-offset*] [Function]
`julian-day->time-monotonic` *jdn* [Function]
`julian-day->time-tai` *jdn* [Function]
`julian-day->time-utc` *jdn* [Function]

`modified-julian-day->date` *jdn* [*tz-offset*] [Function]
`modified-julian-day->time-monotonic` *jdn* [Function]
`modified-julian-day->time-tai` *jdn* [Function]
`modified-julian-day->time-utc` *jdn* [Function]

`time-monotonic->date` *time* [*tz-offset*] [Function]
`time-monotonic->time-tai` *time* [Function]
`time-monotonic->time-tai!` *time* [Function]
`time-monotonic->time-utc` *time* [Function]
`time-monotonic->time-utc!` *time* [Function]

`time-tai->date` *time* [*tz-offset*] [Function]
`time-tai->julian-day` *time* [Function]
`time-tai->modified-julian-day` *time* [Function]
`time-tai->time-monotonic` *time* [Function]
`time-tai->time-monotonic!` *time* [Function]
`time-tai->time-utc` *time* [Function]
`time-tai->time-utc!` *time* [Function]

`time-utc->date` *time* [*tz-offset*] [Function]
`time-utc->julian-day` *time* [Function]

`time-utc->modified-julian-day` *time* [Function]
`time-utc->time-monotonic` *time* [Function]
`time-utc->time-monotonic!` *time* [Function]
`time-utc->time-tai` *time* [Function]
`time-utc->time-tai!` *time* [Function]

Convert between dates, times and days of the respective types. For instance `time-tai->time-utc` accepts a *time* object of type `time-tai` and returns an object of type `time-utc`.

The `!` variants may modify their *time* argument to form their return. The plain functions create a new object.

For conversions to dates, *tz-offset* is seconds east of Greenwich. The default is the local timezone, at the given time, as provided by the system, using `localtime` (see Section 6.2.5 [Time], page 389).

On 32-bit systems, `localtime` is limited to a 32-bit `time_t`, so a default *tz-offset* is only available for times between Dec 1901 and Jan 2038. For prior dates an application might like to use the value in 1902, though some locations have zone changes prior to that. For future dates an application might like to assume today's rules extend indefinitely. But for correct daylight savings transitions it will be necessary to take an offset for the same day and time but a year in range and which has the same starting weekday and same leap/non-leap (to support rules like last Sunday in October).

6.4.15.5 SRFI-19 Date to string

`date->string` *date* [*format*] [Function]

Convert a date to a string under the control of a format. *format* should be a string containing '`~`' escapes, which will be expanded as per the following conversion table. The default *format* is '`~c`', a locale-dependent date and time.

Many of these conversion characters are the same as POSIX `strftime` (see Section 6.2.5 [Time], page 389), but there are some extras and some variations.

`~~`	literal `~`
`~a`	locale abbreviated weekday, eg. '`Sun`'
`~A`	locale full weekday, eg. '`Sunday`'
`~b`	locale abbreviated month, eg. '`Jan`'
`~B`	locale full month, eg. '`January`'
`~c`	locale date and time, eg. '`Fri Jul 14 20:28:42-0400 2000`'
`~d`	day of month, zero padded, '`01`' to '`31`'
`~e`	day of month, blank padded, '` 1`' to '`31`'
`~f`	seconds and fractional seconds, with locale decimal point, eg. '`5.2`'
`~h`	same as `~b`
`~H`	hour, 24-hour clock, zero padded, '`00`' to '`23`'
`~I`	hour, 12-hour clock, zero padded, '`01`' to '`12`'
`~j`	day of year, zero padded, '`001`' to '`366`'
`~k`	hour, 24-hour clock, blank padded, '` 0`' to '`23`'

`~l`	hour, 12-hour clock, blank padded, '` 1`' to '`12`'
`~m`	month, zero padded, '`01`' to '`12`'
`~M`	minute, zero padded, '`00`' to '`59`'
`~n`	newline
`~N`	nanosecond, zero padded, '`000000000`' to '`999999999`'
`~p`	locale AM or PM
`~r`	time, 12 hour clock, '`~I:~M:~S ~p`'
`~s`	number of full seconds since "the epoch" in UTC
`~S`	second, zero padded '`00`' to '`60`' (usual limit is 59, 60 is a leap second)
`~t`	horizontal tab character
`~T`	time, 24 hour clock, '`~H:~M:~S`'
`~U`	week of year, Sunday first day of week, '`00`' to '`52`'
`~V`	week of year, Monday first day of week, '`01`' to '`53`'
`~w`	day of week, 0 for Sunday, '`0`' to '`6`'
`~W`	week of year, Monday first day of week, '`00`' to '`52`'
`~y`	year, two digits, '`00`' to '`99`'
`~Y`	year, full, eg. '`2003`'
`~z`	time zone, RFC-822 style
`~Z`	time zone symbol (not currently implemented)
`~1`	ISO-8601 date, '`~Y-~m-~d`'
`~2`	ISO-8601 time+zone, '`~k:~M:~S~z`'
`~3`	ISO-8601 time, '`~k:~M:~S`'
`~4`	ISO-8601 date/time+zone, '`~Y-~m-~dT~k:~M:~S~z`'
`~5`	ISO-8601 date/time, '`~Y-~m-~dT~k:~M:~S`'

Conversions '`~D`', '`~x`' and '`~X`' are not currently described here, since the specification and reference implementation differ.

Currently Guile doesn't implement any localizations for the above, all outputs are in English, and the '`~c`' conversion is POSIX `ctime` style '`~a ~b ~d ~H:~M:~S~z ~Y`'. This may change in the future.

6.4.15.6 SRFI-19 String to date

`string->date` *input template* [Function]

Convert an *input* string to a date under the control of a *template* string. Return a newly created date object.

Literal characters in *template* must match characters in *input* and '`~`' escapes must match the input forms described in the table below. "Skip to" means characters up to one of the given type are ignored, or "no skip" for no skipping. "Read" is what's then read, and "Set" is the field affected in the date object.

For example '`~Y`' skips input characters until a digit is reached, at which point it expects a year and stores that to the year field of the date.

	Skip to	Read	Set

`~~`	no skip	literal ~	nothing
`~a`	`char-alphabetic?`	locale abbreviated weekday name	nothing
`~A`	`char-alphabetic?`	locale full weekday name	nothing
`~b`	`char-alphabetic?`	locale abbreviated month name	`date-month`
`~B`	`char-alphabetic?`	locale full month name	`date-month`
`~d`	`char-numeric?`	day of month	`date-day`
`~e`	no skip	day of month, blank padded	`date-day`
`~h`	same as ‘`~b`’		
`~H`	`char-numeric?`	hour	`date-hour`
`~k`	no skip	hour, blank padded	`date-hour`
`~m`	`char-numeric?`	month	`date-month`
`~M`	`char-numeric?`	minute	`date-minute`
`~S`	`char-numeric?`	second	`date-second`
`~y`	no skip	2-digit year	`date-year` 50 years
`~Y`	`char-numeric?`	year	`date-year`
`~z`	no skip	time zone	date-zone-offs

Notice that the weekday matching forms don’t affect the date object returned, instead the weekday will be derived from the day, month and year.

Currently Guile doesn’t implement any localizations for the above, month and weekday names are always expected in English. This may change in the future.

6.4.16 SRFI-26 - specializing parameters

This SRFI provides a syntax for conveniently specializing selected parameters of a function. It can be used with,

```
(use-modules (srfi srfi-26))
```

`cut` *slot* . . . [library syntax]
`cute` *slot* . . . [library syntax]

Return a new procedure which will make a call (*slot* . . .) but with selected parameters specialized to given expressions.

An example will illustrate the idea. The following is a specialization of `write`, sending output to `my-output-port`,

```
(cut write <> my-output-port)
⇒
(lambda (obj) (write obj my-output-port))
```

The special symbol `<>` indicates a slot to be filled by an argument to the new procedure. `my-output-port` on the other hand is an expression to be evaluated and passed, ie. it specializes the behaviour of `write`.

`<>` A slot to be filled by an argument from the created procedure. Arguments are assigned to `<>` slots in the order they appear in the `cut` form, there's no way to re-arrange arguments.

The first argument to `cut` is usually a procedure (or expression giving a procedure), but `<>` is allowed there too. For example,

```
(cut <> 1 2 3)
⇒
(lambda (proc) (proc 1 2 3))
```

`<...>` A slot to be filled by all remaining arguments from the new procedure. This can only occur at the end of a `cut` form.

For example, a procedure taking a variable number of arguments like `max` but in addition enforcing a lower bound,

```
(define my-lower-bound 123)

(cut max my-lower-bound <...>)
⇒
(lambda arglist (apply max my-lower-bound arglist))
```

For `cut` the specializing expressions are evaluated each time the new procedure is called. For `cute` they're evaluated just once, when the new procedure is created. The name `cute` stands for "`cut` with evaluated arguments". In all cases the evaluations take place in an unspecified order.

The following illustrates the difference between `cut` and `cute`,

```
(cut format <> "the time is ~s" (current-time))
⇒
(lambda (port) (format port "the time is ~s" (current-time)))

(cute format <> "the time is ~s" (current-time))
⇒
(let ((val (current-time)))
  (lambda (port) (format port "the time is ~s" val))
```

(There's no provision for a mixture of `cut` and `cute` where some expressions would be evaluated every time but others evaluated only once.)

`cut` is really just a shorthand for the sort of `lambda` forms shown in the above examples. But notice `cut` avoids the need to name unspecialized parameters, and is more compact. Use in functional programming style or just with `map`, `for-each` or similar is typical.

```
          (map (cut * 2 <>) '(1 2 3 4))

          (for-each (cut write <> my-port) my-list)
```

6.4.17 SRFI-31 - A special form 'rec' for recursive evaluation

SRFI-31 defines a special form that can be used to create self-referential expressions more conveniently. The syntax is as follows:

```
<rec expression> --> (rec <variable> <expression>)
<rec expression> --> (rec (<variable>+) <body>)
```

The first syntax can be used to create self-referential expressions, for example:

```
  guile> (define tmp (rec ones (cons 1 (delay ones))))
```

The second syntax can be used to create anonymous recursive functions:

```
  guile> (define tmp (rec (display-n item n)
                       (if (positive? n)
                           (begin (display n) (display-n (- n 1))))))
  guile> (tmp 42 3)
  424242
  guile>
```

6.4.18 SRFI-39 - Parameters

This SRFI provides parameter objects, which implement dynamically bound locations for values. The functions below are available from

```
(use-modules (srfi srfi-39))
```

A parameter object is a procedure. Called with no arguments it returns its value, called with one argument it sets the value.

```
(define my-param (make-parameter 123))
(my-param) ⇒ 123
(my-param 456)
(my-param) ⇒ 456
```

The `parameterize` special form establishes new locations for parameters, those new locations having effect within the dynamic scope of the `parameterize` body. Leaving restores the previous locations, or re-entering through a saved continuation will again use the new locations.

```
(parameterize ((my-param 789))
  (my-param) ⇒ 789
  )
(my-param) ⇒ 456
```

Parameters are like dynamically bound variables in other Lisp dialets. They allow an application to establish parameter settings (as the name suggests) just for the execution of a particular bit of code, restoring when done. Examples of such parameters might be case-sensitivity for a search, or a prompt for user input.

Global variables are not as good as parameter objects for this sort of thing. Changes to them are visible to all threads, but in Guile parameter object locations are per-thread, thereby truely limiting the effect of `parameterize` to just its dynamic execution.

Passing arguments to functions is thread-safe, but that soon becomes tedious when there's more than a few or when they need to pass down through several layers of calls before reaching the point they should affect. And introducing a new setting to existing code is often easier with a parameter object than adding arguments.

`make-parameter` *init* [*converter*] [Function]

Return a new parameter object, with initial value *init*.

A parameter object is a procedure. When called `(param)` it returns its value, or a call `(param val)` sets its value. For example,

```
(define my-param (make-parameter 123))
(my-param) ⇒ 123

(my-param 456)
(my-param) ⇒ 456
```

If a *converter* is given, then a call `(converter val)` is made for each value set, its return is the value stored. Such a call is made for the *init* initial value too.

A *converter* allows values to be validated, or put into a canonical form. For example,

```
(define my-param (make-parameter 123
                   (lambda (val)
                     (if (not (number? val))
                         (error "must be a number"))
                     (inexact->exact val))))
(my-param 0.75)
(my-param) ⇒ 3/4
```

`parameterize` ((*param value*) . . .) *body* . . . [library syntax]

Establish a new dynamic scope with the given *param*s bound to new locations and set to the given *value*s. *body* is evaluated in that environment, the result is the return from the last form in *body*.

Each *param* is an expression which is evaluated to get the parameter object. Often this will just be the name of a variable holding the object, but it can be anything that evaluates to a parameter.

The *param* expressions and *value* expressions are all evaluated before establishing the new dynamic bindings, and they're evaluated in an unspecified order.

For example,

```
(define prompt (make-parameter "Type something: "))
(define (get-input)
  (display (prompt))
  ...)

(parameterize ((prompt "Type a number: "))
  (get-input)
  ...)
```

`current-input-port` [*new-port*] [Parameter object]

`current-output-port` [*new-port*] [Parameter object]
`current-error-port` [*new-port*] [Parameter object]

> This SRFI extends the core `current-input-port` and `current-output-port`, making them parameter objects. The Guile-specific `current-error-port` is extended too, for consistency. (see Section 5.12.8 [Default Ports], page 278.)
>
> This is an upwardly compatible extension, a plain call like `(current-input-port)` still returns the current input port, and `set-current-input-port` can still be used. But the port can now also be set with `(current-input-port my-port)` and bound dynamically with `parameterize`.

`with-parameters*` *param-list value-list thunk* [Function]

> Establish a new dynamic scope, as per `parameterize` above, taking parameters from *param-list* and corresponding values from *values-list*. A call `(thunk)` is made in the new scope and the result from that *thunk* is the return from `with-parameters*`.
>
> This function is a Guile-specific addition to the SRFI, it's similar to the core `with-fluids*` (see Section 5.17.8 [Fluids and Dynamic States], page 330).

Parameter objects are implemented using fluids (see Section 5.17.8 [Fluids and Dynamic States], page 330), so each dynamic state has it's own parameter locations. That includes the separate locations when outside any `parameterize` form. When a parameter is created it gets a separate initial location in each dynamic state, all initialized to the given *init* value.

As alluded to above, because each thread usually has a separate dynamic state, each thread has it's own locations behind parameter objects, and changes in one thread are not visible to any other. When a new dynamic state or thread is created, the values of parameters in the originating context are copied, into new locations.

SRFI-39 doesn't specify the interaction between parameter objects and threads, so the threading behaviour described here should be regarded as Guile-specific.

6.4.19 SRFI-55 - Requiring Features

SRFI-55 provides `require-extension` which is a portable mechanism to load selected SRFI modules. This is implemented in the Guile core, there's no module needed to get SRFI-55 itself.

`require-extension` *clause...* [library syntax]

> Require each of the given *clause* features, throwing an error if any are unavailable.
>
> A *clause* is of the form `(identifier arg...)`. The only *identifier* currently supported is `srfi` and the arguments are SRFI numbers. For example to get SRFI-1 and SRFI-6,
>
> ```
> (require-extension (srfi 1 6))
> ```
>
> `require-extension` can only be used at the top-level.
>
> A Guile-specific program can simply `use-modules` to load SRFIs not already in the core, `require-extension` is for programs designed to be portable to other Scheme implementations.

6.4.20 SRFI-60 - Integers as Bits

This SRFI provides various functions for treating integers as bits and for bitwise manipulations. These functions can be obtained with,

```
(use-modules (srfi srfi-60))
```

Integers are treated as infinite precision twos-complement, the same as in the core logical functions (see Section 5.5.2.14 [Bitwise Operations], page 117). And likewise bit indexes start from 0 for the least significant bit. The following functions in this SRFI are already in the Guile core,

> `logand`, `logior`, `logxor`, `lognot`, `logtest`, `logcount`, `integer-length`, `logbit?`, `ash`

`bitwise-and` *n1 ...* [Function]
`bitwise-ior` *n1 ...* [Function]
`bitwise-xor` *n1 ...* [Function]
`bitwise-not` *n* [Function]
`any-bits-set?` *j k* [Function]
`bit-set?` *index n* [Function]
`arithmetic-shift` *n count* [Function]
`bit-field` *n start end* [Function]
`bit-count` *n* [Function]

> Aliases for `logand`, `logior`, `logxor`, `lognot`, `logtest`, `logbit?`, `ash`, `bit-extract` and `logcount` respectively.
>
> Note that the name `bit-count` conflicts with `bit-count` in the core (see Section 5.6.5 [Bit Vectors], page 187).

`bitwise-if` *mask n1 n0* [Function]
`bitwise-merge` *mask n1 n0* [Function]

> Return an integer with bits selected from *n1* and *n0* according to *mask*. Those bits where *mask* has 1s are taken from *n1*, and those where *mask* has 0s are taken from *n0*.
>
> ```
> (bitwise-if 3 #b0101 #b1010) ⇒ 9
> ```

`log2-binary-factors` *n* [Function]
`first-set-bit` *n* [Function]

> Return a count of how many factors of 2 are present in *n*. This is also the bit index of the lowest 1 bit in *n*. If *n* is 0, the return is -1.
>
> ```
> (log2-binary-factors 6) ⇒ 1
> (log2-binary-factors -8) ⇒ 3
> ```

`copy-bit` *index n newbit* [Function]

> Return *n* with the bit at *index* set according to *newbit*. *newbit* should be `#t` to set the bit to 1, or `#f` to set it to 0. Bits other than at *index* are unchanged in the return.
>
> ```
> (copy-bit 1 #b0101 #t) ⇒ 7
> ```

`copy-bit-field` *n newbits start end* [Function]

> Return *n* with the bits from *start* (inclusive) to *end* (exclusive) changed to the value *newbits*.

The least significant bit in *newbits* goes to *start*, the next to *start* + 1, etc. Anything in *newbits* past the *end* given is ignored.

```
(copy-bit-field #b10000 #b11 1 3) ⇒ #b10110
```

`rotate-bit-field` *n count start end* [Function]

Return *n* with the bit field from *start* (inclusive) to *end* (exclusive) rotated upwards by *count* bits.

count can be positive or negative, and it can be more than the field width (it'll be reduced modulo the width).

```
(rotate-bit-field #b0110 2 1 4) ⇒ #b1010
```

`reverse-bit-field` *n start end* [Function]

Return *n* with the bits from *start* (inclusive) to *end* (exclusive) reversed.

```
(reverse-bit-field #b101001 2 4) ⇒ #b100101
```

`integer->list` *n* [*len*] [Function]

Return bits from *n* in the form of a list of `#t` for 1 and `#f` for 0. The least significant *len* bits are returned, and the first list element is the most significant of those bits. If *len* is not given, the default is `(integer-length n)` (see Section 5.5.2.14 [Bitwise Operations], page 117).

```
(integer->list 6)   ⇒ (#t #t #f)
(integer->list 1 4) ⇒ (#f #f #f #t)
```

`list->integer` *lst* [Function]
`booleans->integer` *bool. . .* [Function]

Return an integer formed bitwise from the given *lst* list of booleans, or for `booleans->integer` from the *bool* arguments.

Each boolean is `#t` for a 1 and `#f` for a 0. The first element becomes the most significant bit in the return.

```
(list->integer '(#t #f #t #f)) ⇒ 10
```

6.4.21 SRFI-61 - A more general `cond` clause

This SRFI extends RnRS `cond` to support test expressions that return multiple values, as well as arbitrary definitions of test success. SRFI 61 is implemented in the Guile core; there's no module needed to get SRFI-61 itself. Extended `cond` is documented in Section 5.11.2 [Simple Conditional Evaluation], page 251.

6.5 Readline Support

Guile comes with an interface module to the readline library (see section "Top" in *GNU Readline Library*). This makes interactive use much more convenient, because of the command-line editing features of readline. Using `(ice-9 readline)`, you can navigate through the current input line with the cursor keys, retrieve older command lines from the input history and even search through the history entries.

6.5.1 Loading Readline Support

The module is not loaded by default and so has to be loaded and activated explicitly. This is done with two simple lines of code:

```
(use-modules (ice-9 readline))
(activate-readline)
```

The first line will load the necessary code, and the second will activate readline's features for the REPL. If you plan to use this module often, you should save these to lines to your '`.guile`' personal startup file.

You will notice that the REPL's behaviour changes a bit when you have loaded the readline module. For example, when you press Enter before typing in the closing parentheses of a list, you will see the *continuation* prompt, three dots: `...` This gives you a nice visual feedback when trying to match parentheses. To make this even easier, *bouncing parentheses* are implemented. That means that when you type in a closing parentheses, the cursor will jump to the corresponding opening parenthesis for a short time, making it trivial to make them match.

Once the readline module is activated, all lines entered interactively will be stored in a history and can be recalled later using the cursor-up and -down keys. Readline also understands the Emacs keys for navigating through the command line and history.

When you quit your Guile session by evaluating `(quit)` or pressing Ctrl-D, the history will be saved to the file '`.guile_history`' and read in when you start Guile for the next time. Thus you can start a new Guile session and still have the (probably long-winded) definition expressions available.

You can specify a different history file by setting the environment variable `GUILE_HISTORY`. And you can make Guile specific customizations to your '`.inputrc`' by testing for application '`Guile`' (see section "Conditional Init Constructs" in *GNU Readline Library*). For instance to define a key inserting a matched pair of parentheses,

```
$if Guile
  "\C-o": "()\C-b"
$endif
```

6.5.2 Readline Options

The readline interface module can be configured in several ways to better suit the user's needs. Configuration is done via the readline module's options interface, in a similar way to the evaluator and debugging options (see Section 5.18.3 [Runtime Options], page 338).

Here is the list of readline options generated by typing `(readline-options 'full)` in Guile. You can also see the default values.

```
bounce-parens   500   Time (ms) to show matching opening parenthesis (0 = off).
```

```
history-length  200   History length.
history-file    yes   Use history file.
```

The history length specifies how many input lines will be remembered. If the history contains that many lines and additional lines are entered, the oldest lines will be lost. You can switch on/off the usage of the history file using the following call.

```
(readline-disable 'history)
```

The readline options interface can only be used *after* loading the readline module, because it is defined in that module.

6.5.3 Readline Functions

The following functions are provided by

```
(use-modules (ice-9 readline))
```

There are two ways to use readline from Scheme code, either make calls to `readline` directly to get line by line input, or use the readline port below with all the usual reading functions.

`readline` [*prompt*] [Function]

Read a line of input from the user and return it as a string (without a newline at the end). *prompt* is the prompt to show, or the default is the string set in `set-readline-prompt!` below.

```
(readline "Type something: ") ⇒ "hello"
```

`set-readline-input-port!` *port* [Function]
`set-readline-output-port!` *port* [Function]

Set the input and output port the readline function should read from and write to. *port* must be a file port (see Section 5.12.9.1 [File Ports], page 280), and should usually be a terminal.

The default is the `current-input-port` and `current-output-port` (see Section 5.12.8 [Default Ports], page 278) when `(ice-9 readline)` loads, which in an interactive user session means the Unix "standard input" and "standard output".

6.5.3.1 Readline Port

`readline-port` [Function]

Return a buffered input port (see Section 6.12 [Buffered Input], page 479) which calls the `readline` function above to get input. This port can be used with all the usual reading functions (`read`, `read-char`, etc), and the user gets the interactive editing features of readline.

There's only a single readline port created. `readline-port` creates it when first called, and on subsequent calls just returns what it previously made.

`activate-readline` [Function]

If the `current-input-port` is a terminal (see Section 6.2.9 [`isatty?`], page 400) then enable readline for all reading from `current-input-port` (see Section 5.12.8 [Default Ports], page 278) and enable readline features in the interactive REPL (see Section 3.1.3.3 [The REPL], page 23).

```
(activate-readline)
(read-char)
```

`activate-readline` enables readline on `current-input-port` simply by a `set-current-input-port` to the `readline-port` above. An application can do that directly if the extra REPL features that `activate-readline` adds are not wanted.

`set-readline-prompt!` *prompt1* [*prompt2*] [Function]

Set the prompt string to print when reading input. This is used when reading through `readline-port`, and is also the default prompt for the `readline` function above.

prompt1 is the initial prompt shown. If a user might enter an expression across multiple lines, then *prompt2* is a different prompt to show further input required. In the Guile REPL for instance this is an ellipsis ('`...`').

See `set-buffered-input-continuation?!` (see Section 6.12 [Buffered Input], page 479) for an application to indicate the boundaries of logical expressions (assuming of course an application has such a notion).

6.5.3.2 Completion

`with-readline-completion-function` *completer thunk* [Function]

Call `(`*thunk*`)` with *completer* as the readline tab completion function to be used in any readline calls within that *thunk*. *completer* can be `#f` for no completion.

completer will be called as `(`*completer* `text state)`, as described in (see section "How Completing Works" in *GNU Readline Library*). *text* is a partial word to be completed, and each *completer* call should return a possible completion string or `#f` when no more. *state* is `#f` for the first call asking about a new *text* then `#t` while getting further completions of that *text*.

Here's an example *completer* for user login names from the password file (see Section 6.2.4 [User Information], page 386), much like readline's own `rl_username_completion_function`,

```
(define (username-completer-function text state)
  (if (not state)
      (setpwent))  ;; new, go to start of database
  (let more ((pw (getpwent)))
    (if pw
        (if (string-prefix? text (passwd:name pw))
            (passwd:name pw)     ;; this name matches, return it
            (more (getpwent)))   ;; doesn't match, look at next
        (begin
          ;; end of database, close it and return #f
          (endpwent)
          #f))))
```

`apropos-completion-function` *text state* [Function]

A completion function offering completions for Guile functions and variables (all `define`s). This is the default completion function.

`filename-completion-function` *text state* [Function]

A completion function offering filename completions. This is readline's `rl_filename_completion_function` (see section "Completion Functions" in *GNU Readline Library*).

`make-completion-function` *string-list* [Function]

Return a completion function which offers completions from the possibilities in *string-list*. Matching is case-sensitive.

6.6 Value History

Another module which makes command line usage more convenient is `(ice-9 history)`. This module will change the REPL so that each value which is evaluated and printed will be remembered under a name constructed from the dollar character (`$`) and the number of the evaluated expression.

Consider an example session.

```
guile> (use-modules (ice-9 history))
guile> 1
$1 = 1
guile> (+ $1 $1)
$2 = 2
guile> (* $2 $2)
$3 = 4
```

After loading the value history module `(ice-9 history)`, one (trivial) expression is evaluated. The result is stored into the variable `$1`. This fact is indicated by the output `$1 =` , which is also caused by `(ice-9 history)`. In the next line, this variable is used two times, to produce the value `$2`, which in turn is used in the calculation for `$3`.

6.7 Pretty Printing

The module `(ice-9 pretty-print)` provides the procedure `pretty-print`, which provides nicely formatted output of Scheme objects. This is especially useful for deeply nested or complex data structures, such as lists and vectors.

The module is loaded by simply saying.

```
(use-modules (ice-9 pretty-print))
```

This makes the procedure `pretty-print` available. As an example how `pretty-print` will format the output, see the following:

```
(pretty-print '(define (foo) (lambda (x)
(cond ((zero? x) #t) ((negative? x) -x) (else
(if (= x 1) 2 (* x x x)))))))
⊣
(define (foo)
  (lambda (x)
    (cond ((zero? x) #t)
          ((negative? x) -x)
          (else (if (= x 1) 2 (* x x x))))))
```

`pretty-print` *obj* [*port*] [*keyword-options*] [Scheme Procedure]

Print the textual representation of the Scheme object *obj* to *port*. *port* defaults to the current output port, if not given.

The further *keyword-options* are keywords and parameters as follows,

`#:display?` *flag*
: If *flag* is true then print using `display`. The default is `#f` which means use `write` style. (see Section 5.12.3 [Writing], page 273)

`#:per-line-prefix` *string*
: Print the given *string* as a prefix on each line. The default is no prefix.

`#:width` *columns*
: Print within the given *columns*. The default is 79.

6.8 Formatted Output

The `format` function is a powerful way to print numbers, strings and other objects together with literal text under the control of a format string. This function is available from

```
(use-modules (ice-9 format))
```

A format string is generally more compact and easier than using just the standard procedures like `display`, `write` and `newline`. Parameters in the output string allow various output styles, and parameters can be taken from the arguments for runtime flexibility.

`format` is similar to the Common Lisp procedure of the same name, but it's not identical and doesn't have quite all the features found in Common Lisp.

C programmers will note the similarity between `format` and `printf`, though escape sequences are marked with `~` instead of `%`, and are more powerful.

`format` *dest fmt* [*args...*] [Scheme Procedure]

Write output specified by the *fmt* string to *dest*. *dest* can be an output port, `#t` for `current-output-port` (see Section 5.12.8 [Default Ports], page 278), a number for `current-error-port`, or `#f` to return the output as a string.

fmt can contain literal text to be output, and `~` escapes. Each escape has the form

```
~ [param [, param...] [:] [@] code
```

`code` is a character determining the escape sequence. The `:` and `@` characters are optional modifiers, one or both of which change the way various codes operate. Optional parameters are accepted by some codes too. Parameters have the following forms,

`[+/-]number`
: An integer, with optional `+` or `-`.

`'` (apostrophe)
: The following character in the format string, for instance `'z` for `z`.

`v`
: The next function argument as the parameter. `v` stands for "variable", a parameter can be calculated at runtime and included in the arguments. Upper case `V` can be used too.

`#`
: The number of arguments remaining. (See `~*` below for some usages.)

Parameters are separated by commas (`,`). A parameter can be left empty to keep its default value when supplying later parameters.

The following escapes are available. The code letters are not case-sensitive, upper and lower case are the same.

`~a`
`~s`
: Object output. Parameters: *minwidth*, *padinc*, *minpad*, *padchar*.

`~a` outputs an argument like `display`, `~s` outputs an argument like `write` (see Section 5.12.3 [Writing], page 273).

```
(format #t "~a" "foo") ⊣ foo
(format #t "~s" "foo") ⊣ "foo"
```

`~:a` and `~:s` put objects that don't have an external representation in quotes like a string.

```
(format #t "~:a" car) ⊣ "#<primitive-procedure car>"
```

If the output is less than *minwidth* characters (default 0), it's padded on the right with *padchar* (default space). `~@a` and `~@s` put the padding on the left instead.

```
(format #f "~5a" 'abc)       ⇒ "abc  "
(format #f "~5,,,'-@a" 'abc) ⇒ "--abc"
```

minpad is a minimum for the padding then plus a multiple of *padinc*. Ie. the padding is *minpad*+N∗*padinc*, where *n* is the smallest integer making the total object plus padding greater than or equal to *minwidth*. The default *minpad* is 0 and the default *padinc* is 1 (imposing no minimum or multiple).

```
(format #f "~5,1,4a" 'abc) ⇒ "abc    "
```

`~c` Character. Parameter: *charnum*.

Output a character. The default is to simply output, as per `write-char` (see Section 5.12.3 [Writing], page 273). `~@c` prints in `write` style. `~:c` prints control characters (ASCII 0 to 31) in `^X` form.

```
(format #t "~c" #\z)        ⊣ z
(format #t "~@c" #\z)       ⊣ #\z
(format #t "~:c" #\newline) ⊣ ^J
```

If the *charnum* parameter is given then an argument is not taken but instead the character is `(integer->char` *charnum*`)` (see Section 5.5.3 [Characters], page 121). This can be used for instance to output characters given by their ASCII code.

```
(format #t "~65c")  ⊣ A
```

`~d`
`~x`
`~o`
`~b` Integer. Parameters: *minwidth*, *padchar*, *commachar*, *commawidth*.

Output an integer argument as a decimal, hexadecimal, octal or binary integer (respectively).

```
(format #t "~d" 123) ⊣ 123
```

`~@d` etc shows a `+` sign is shown on positive numbers.

```
(format #t "~@b" 12) ⊣ +1100
```

If the output is less than the *minwidth* parameter (default no minimum), it's padded on the left with the *padchar* parameter (default space).

```
(format #t "~5,'*d" 12)   ⊣ ***12
(format #t "~5,'0d" 12)   ⊣ 00012
(format #t "~3d"    1234) ⊣ 1234
```

`~:d` adds commas (or the *commachar* parameter) every three digits (or the *commawidth* parameter many).

```
(format #t "~:d" 1234567)         ⊣ 1,234,567
(format #t "~10,'*,'/,2:d" 12345) ⊣ ***1/23/45
```

Hexadecimal `~x` output is in lower case, but the `~(` and `~)` case conversion directives described below can be used to get upper case.

```
(format #t "~x"       65261) ⊣ feed
(format #t "~:@(~x~)" 65261) ⊣ FEED
```

`~r` Integer in words, roman numerals, or a specified radix. Parameters: *radix*, *minwidth*, *padchar*, *commachar*, *commawidth*.

With no parameters output is in words as a cardinal like "ten", or `~:r` prints an ordinal like "tenth".

```
(format #t "~r" 9)  ⊣ nine        ;; cardinal
(format #t "~r" -9) ⊣ minus nine  ;; cardinal
(format #t "~:r" 9) ⊣ ninth       ;; ordinal
```

And also with no parameters, `~@r` gives roman numerals and `~:@r` gives old roman numerals. In old roman numerals there's no "subtraction", so 9 is `VIIII` instead of `IX`. In both cases only positive numbers can be output.

```
(format #t "~@r" 89)  ⊣ LXXXIX     ;; roman
(format #t "~:@r" 89) ⊣ LXXXVIIII  ;; old roman
```

When a parameter is given it means numeric output in the specified *radix*. The modifiers and parameters following the radix are the same as described for `~d` etc above.

```
(format #f "~3r" 27)   ⇒ "1000"    ;; base 3
(format #f "~3,5r" 26) ⇒ "  222"   ;; base 3 width 5
```

`~f` Fixed-point float. Parameters: *width*, *decimals*, *scale*, *overflowchar*, *padchar*.

Output a number or number string in fixed-point format, ie. with a decimal point.

```
(format #t "~f" 5)       ⊣ 5.0
(format #t "~f" "123")   ⊣ 123.0
(format #t "~f" "1e-1")  ⊣ 0.1
```

`~@f` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@f" 0) ⊣ +0.0
```

If the output is less than *width* characters it's padded on the left with *padchar* (space by default). If the output equals or exceeds *width* then there's no padding. The default for *width* is no padding.

```
(format #f "~6f" -1.5)       ⇒ "  -1.5"
(format #f "~6,,,,'*f" 23)   ⇒ "**23.0"
(format #f "~6f" 1234567.0) ⇒ "1234567.0"
```

decimals is how many digits to print after the decimal point, with the value rounded or padded with zeros as necessary. (The default is to output as many decimals as required.)

```
(format #t "~1,2f" 3.125) ⊣ 3.13
(format #t "~1,2f" 1.5)   ⊣ 1.50
```

scale is a power of 10 applied to the value, moving the decimal point that many places. A positive *scale* increases the value shown, a negative decreases it.

```
(format #t "~,,2f" 1234)   ⊣ 123400.0
(format #t "~,,-2f" 1234)  ⊣ 12.34
```

If *overflowchar* and *width* are both given and if the output would exceed *width*, then that many *overflowchar*s are printed instead of the value.

```
(format #t "~5,,,'xf" 12345)  ⊣ 12345
(format #t "~4,,,'xf" 12345)  ⊣ xxxx
```

`~e` Exponential float. Parameters: *width*, *mantdigits*, *expdigits*, *intdigits*, *overflowchar*, *padchar*, *expchar*.

Output a number or number string in exponential notation.

```
(format #t "~e" 5000.25)  ⊣ 5.00025E+3
(format #t "~e" "123.4")  ⊣ 1.234E+2
(format #t "~e" "1e4")    ⊣ 1.0E+4
```

`~@e` prints a `+` sign on positive numbers (including zero). (This is for the mantissa, a `+` or `-` sign is always shown on the exponent.)

```
(format #t "~@e" 5000.0)  ⊣ +5.0E+3
```

If the output is less than *width* characters it's padded on the left with *padchar* (space by default). The default for *width* is to output with no padding.

```
(format #f "~10e" 1234.0)      ⇒ "  1.234E+3"
(format #f "~10,,,,,'*e" 0.5)  ⇒ "****5.0E-1"
```

mantdigits is the number of digits shown in the mantissa after the decimal point. The value is rounded or trailing zeros are added as necessary. The default *mantdigits* is to show as much as needed by the value.

```
(format #f "~,3e" 11111.0)  ⇒ "1.111E+4"
(format #f "~,8e" 123.0)    ⇒ "1.23000000E+2"
```

expdigits is the minimum number of digits shown for the exponent, with leading zeros added if necessary. The default for *expdigits* is to show only as many digits as required. At least 1 digit is always shown.

```
(format #f "~,,1e" 1.0e99)  ⇒ "1.0E+99"
(format #f "~,,6e" 1.0e99)  ⇒ "1.0E+000099"
```

intdigits (default 1) is the number of digits to show before the decimal point in the mantissa. *intdigits* can be zero, in which case the integer part is a single `0`, or it can be negative, in which case leading zeros are shown after the decimal point.

```
(format #t "~,,,3e" 12345.0)   ⊣ 123.45E+2
(format #t "~,,,0e" 12345.0)   ⊣ 0.12345E+5
(format #t "~,,,-3e" 12345.0)  ⊣ 0.00012345E+8
```

If *overflowchar* is given then *width* is a hard limit. If the output would exceed *width* then instead that many *overflowchar*s are printed.

```
(format #f "~6,,,,'xe" 100.0) ⇒ "1.0E+2"
(format #f "~3,,,,'xe" 100.0) ⇒ "xxx"
```

expchar is the exponent marker character (default `E`).

```
(format #t "~,,,,,,'ee" 100.0) ⊣ 1.0e+2
```

`~g` General float. Parameters: *width*, *mantdigits*, *expdigits*, *intdigits*, *overflowchar*, *padchar*, *expchar*.

Output a number or number string in either exponential format the same as `~e`, or fixed-point format like `~f` but aligned where the mantissa would have been and followed by padding where the exponent would have been.

Fixed-point is used when the absolute value is 0.1 or more and it takes no more space than the mantissa in exponential format, ie. basically up to *mantdigits* digits.

```
(format #f "~12,4,2g" 999.0)    ⇒ "   999.0    "
(format #f "~12,4,2g" "100000") ⇒ "  1.0000E+05"
```

The parameters are interpreted as per `~e` above. When fixed-point is used, the *decimals* parameter to `~f` is established from *mantdigits*, so as to give a total $mantdigits + 1$ figures.

`~$` Monetary style fixed-point float. Parameters: *decimals*, *intdigits*, *width*, *padchar*.

Output a number or number string in fixed-point format, ie. with a decimal point. *decimals* is the number of decimal places to show, default 2.

```
(format #t "~$" 5)        ⊣ 5.00
(format #t "~4$" "2.25") ⊣ 2.2500
(format #t "~4$" "1e-2") ⊣ 0.0100
```

`~@$` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@$" 0) ⊣ +0.00
```

intdigits is a minimum number of digits to show in the integer part of the value (default 1).

```
(format #t "~,3$" 9.5)    ⊣ 009.50
(format #t "~,0$" 0.125) ⊣ .13
```

If the output is less than *width* characters (default 0), it's padded on the left with *padchar* (default space). `~:$` puts the padding after the sign.

```
(format #f "~,,8$" -1.5)    ⇒ "   -1.50"
(format #f "~,,8:$" -1.5)   ⇒ "-   1.50"
(format #f "~,,8,'.:@$" 3) ⇒ "+...3.00"
```

Note that floating point for dollar amounts is generally not a good idea, because a cent 0.01 cannot be represented exactly in the binary floating point Guile uses, which leads to slowly accumulating rounding errors. Keeping values as cents (or fractions of a cent) in integers then printing with the scale option in `~f` may be a better approach.

`~i` Complex fixed-point float. Parameters: *width*, *decimals*, *scale*, *overflowchar*, *padchar*.

Output the argument as a complex number, with both real and imaginary part shown (even if one or both are zero).

The parameters and modifiers are the same as for fixed-point `~f` described above. The real and imaginary parts are both output with the same given parameters and modifiers, except that for the imaginary part the `@` modifier is always enabled, so as to print a `+` sign between the real and imaginary parts.

```
(format #t "~i" 1)   ⊣ 1.0+0.0i
```

`~p` Plural. No parameters.

Output nothing if the argument is 1, or '`s`' for any other value.

```
(format #t "enter name~p" 1) ⊣ enter name
(format #t "enter name~p" 2) ⊣ enter names
```

`~@p` prints '`y`' for 1 or '`ies`' otherwise.

```
(format #t "pupp~@p" 1) ⊣ puppy
(format #t "pupp~@p" 2) ⊣ puppies
```

`~:p` re-uses the preceding argument instead of taking a new one, which can be convenient when printing some sort of count.

```
(format #t "~d cat~:p" 9)   ⊣ 9 cats
(format #t "~d pupp~:@p" 5) ⊣ 5 puppies
```

`~p` is designed for English plurals and there's no attempt to support other languages. `~[` conditionals (below) may be able to help. When using `gettext` to translate messages `ngettext` is probably best though (see Section 5.20 [Internationalization], page 345).

`~y` Pretty print. No parameters.

Output an argument with `pretty-print` (see Section 6.7 [Pretty Printing], page 462).

`~?`
`~k` Sub-format. No parameters.

Take a format string argument and a second argument which is a list of arguments for that string, and output the result.

```
(format #t "~?" "~d ~d" '(1 2))    ⊣ 1 2
```

`~@?` takes arguments for the sub-format directly rather than in a list.

```
(format #t "~@? ~s" "~d ~d" 1 2 "foo") ⊣ 1 2 "foo"
```

`~?` and `~k` are the same, `~k` is provided for T-Scheme compatibility.

`~*` Argument jumping. Parameter: *N*.

Move forward *N* arguments (default 1) in the argument list. `~:*` moves backwards. (*N* cannot be negative.)

```
(format #f "~d ~2*~d" 1 2 3 4) ⇒ "1 4"
(format #f "~d ~:*~d" 6)       ⇒ "6 6"
```

`~@*` moves to argument number *N*. The first argument is number 0 (and that's the default for *N*).

```
(format #f "~d~d again ~@*~d~d" 1 2) ⇒ "12 again 12"
(format #f "~d~d~d ~1@*~d~d" 1 2 3)  ⇒ "123 23"
```

A `#` move to the end followed by a `:` modifier move back can be used for an absolute position relative to the end of the argument list, a reverse of what the `@` modifier does.

```
(format #t "~#*~2:*~a" 'a 'b 'c 'd)   ⊣ c
```

At the end of the format string the current argument postion doesn't matter, any further arguments are ignored.

`~t` Advance to a column position. Parameters: *colnum*, *colinc*, *padchar*.

Output *padchar* (space by default) to move to the given *colnum* column. The start of the line is column 0, the default for *colnum* is 1.

```
(format #f "~tX")  ⇒ " X"
(format #f "~3tX") ⇒ "   X"
```

If the current column is already past *colnum*, then the move is to there plus a multiple of *colinc*, ie. column $colnum + N * colinc$ for the smallest N which makes that value greater than or equal to the current column. The default *colinc* is 1 (which means no further move).

```
(format #f "abcd~2,5,'.tx") ⇒ "abcd...x"
```

`~@t` takes *colnum* as an offset from the current column. *colnum* many pad characters are output, then further padding to make the current column a multiple of *colinc*, if it isn't already so.

```
(format #f "a~3,5'*@tx") ⇒ "a****x"
```

`~t` is implemented using `port-column` (see Section 5.12.2 [Reading], page 272), so it works even there has been other output before `format`.

`~~` Tilde character. Parameter: *n*.

Output a tilde character `~`, or *n* many if a parameter is given. Normally `~` introduces an escape sequence, `~~` is the way to output a literal tilde.

`~%` Newline. Parameter: *n*.

Output a newline character, or *n* many if a parameter is given. A newline (or a few newlines) can of course be output just by including them in the format string.

`~&` Start a new line. Parameter: *n*.

Output a newline if not already at the start of a line. With a parameter, output that many newlines, but with the first only if not already at the start of a line. So for instance 3 would be a newline if not already at the start of a line, and 2 further newlines.

`~_` Space character. Parameter: *n*.

Output a space character, or *n* many if a parameter is given.

With a variable parameter this is one way to insert runtime calculated padding (`~t` or the various field widths can do similar things).

```
(format #f "~v_foo" 4) ⇒ "    foo"
```

`~/` Tab character. Parameter: *n*.

Output a tab character, or *n* many if a parameter is given.

`~|` Formfeed character. Parameter: *n*.

Output a formfeed character, or *n* many if a parameter is given.

`~!` Force output. No parameters.

At the end of output, call `force-output` to flush any buffers on the destination (see Section 5.12.3 [Writing], page 273). `~!` can occur anywhere in the format string, but the force is done at the end of output.

When output is to a string (destination `#f`), `~!` does nothing.

`~newline` (ie. newline character)

Continuation line. No parameters.

Skip this newline and any following whitespace in the format string, ie. don't send it to the output. This can be used to break up a long format string for readability, but not print the extra whitespace.

```
(format #f "abc~
            ~d def~
            ~d" 1 2) ⇒ "abc1 def2"
```

`~:newline` skips the newline but leaves any further whitespace to be printed normally.

`~@newline` prints the newline then skips following whitespace.

`~( ~)` Case conversion. No parameters.

Between `~(` and `~)` the case of all output is changed. The modifiers on `~(` control the conversion.

`~(` — lower case.

`~:@(` — upper case.

For example,

```
(format #t "~(Hello~)")    ⊣ hello
(format #t "~:@(Hello~)") ⊣ HELLO
```

In the future it's intended the modifiers `:` and `@` alone will capitalize the first letters of words, as per Common Lisp `format`, but the current implementation of this is flawed and not recommended for use.

Case conversions do not nest, currently. This might change in the future, but if it does then it will be to Common Lisp style where the outermost conversion has priority, overriding inner ones (making those fairly pointless).

`~{ ~}` Iteration. Parameter: *maxreps* (for `~{`).

The format between `~{` and `~}` is iterated. The modifiers to `~{` determine how arguments are taken. The default is a list argument with each iteration successively consuming elements from it. This is a convenient way to output a whole list.

```
(format #t "~{~d~}"     '(1 2 3))        ⊣ 123
(format #t "~{~s=~d ~}" '("x" 1 "y" 2))  ⊣ "x"=1 "y"=2
```

`~:{` takes a single argument which is a list of lists, each of those contained lists gives the arguments for the iterated format.

```
(format #t "~:{~dx~d ~}" '((1 2) (3 4) (5 6)))
⊣ 1x2 3x4 5x6
```

`~@{` takes arguments directly, with each iteration successively consuming arguments.

```
(format #t "~@{~d~}"     1 2 3)        ⊣ 123
(format #t "~@{~s=~d ~}" "x" 1 "y" 2)  ⊣ "x"=1 "y"=2
```

`~:@{` takes list arguments, one argument for each iteration, using that list for the format.

```
(format #t "~:@{~dx~d ~}" '(1 2) '(3 4) '(5 6))
⊣ 1x2 3x4 5x6
```

Iterating stops when there are no more arguments or when the *maxreps* parameter to `~{` is reached (default no maximum).

```
(format #t "~2{~d~}" '(1 2 3 4)) ⊣ 12
```

If the format between `~{` and `~}` is empty, then a format string argument is taken (before iteration argument(s)) and used instead. This allows a sub-format (like `~?` above) to be iterated.

```
(format #t "~{~}" "~d" '(1 2 3)) ⊣ 123
```

Iterations can be nested, an inner iteration operates in the same way as described, but of course on the arguments the outer iteration provides it. This can be used to work into nested list structures. For example in the following the inner `~{~d~}x` is applied to `(1 2)` then `(3 4 5)` etc.

```
(format #t "~{~{~d~}x~}" '((1 2) (3 4 5))) ⊣ 12x345x
```

See also `~^` below for escaping from iteration.

`~[ ~; ~]` Conditional. Parameter: *selector*.

A conditional block is delimited by `~[` and `~]`, and `~;` separates clauses within the block. `~[` takes an integer argument and that number clause is used. The first clause is number 0.

```
(format #f "~[peach~;banana~;mango~]" 1)  ⇒ "banana"
```

The *selector* parameter can be used for the clause number, instead of taking an argument.

```
(format #f "~2[peach~;banana~;mango~]") ⇒ "mango"
```

If the clause number is out of range then nothing is output. Or the last clause can be `~:;` to use that for a number out of range.

```
(format #f "~[banana~;mango~]"         99) ⇒ ""
(format #f "~[banana~;mango~:;fruit~]" 99) ⇒ "fruit"
```

`~:[` treats the argument as a flag, and expects two clauses. The first is used if the argument is `#f` or the second otherwise.

```
(format #f "~:[false~;not false~]" #f)   ⇒ "false"
(format #f "~:[false~;not false~]" 'abc) ⇒ "not false"

(let ((n 3))
  (format #t "~d gnu~:[s are~; is~] here" n (= 1 n)))
⊣ 3 gnus are here
```

`~@[` also treats the argument as a flag, and expects one clause. If the argument is `#f` then no output is produced and the argument is consumed, otherwise the clause is used and the argument is not consumed, it's left for the clause. This can be used for instance to suppress output if `#f` means something not available.

```
(format #f "~@[temperature=~d~]" 27) ⇒ "temperature=27"
(format #f "~@[temperature=~d~]" #f) ⇒ ""
```

`~^` Escape. Parameters: *val1*, *val2*, *val3*.

Stop formatting if there are no more arguments. This can be used for instance to have a format string adapt to a variable number of arguments.

```
(format #t "~d~^ ~d" 1)   ⊣ 1
(format #t "~d~^ ~d" 1 2) ⊣ 1 2
```

Within a `~{ ~}` iteration, `~^` stops the current iteration step if there are no more arguments to that step, but continuing with possible further steps and the rest of the format. This can be used for instance to avoid a separator on the last iteration, or to adapt to variable length argument lists.

```
(format #f "~{~d~^/~} go"    '(1 2 3))      ⇒ "1/2/3 go"
(format #f "~:{ ~d~^~d~} go" '((1) (2 3))) ⇒ " 1 23 go"
```

Within a `~?` sub-format, `~^` operates just on that sub-format. If it terminates the sub-format then the originating format will still continue.

```
(format #t "~? items" "~d~^ ~d" '(1))   ⊣ 1 items
(format #t "~? items" "~d~^ ~d" '(1 2)) ⊣ 1 2 items
```

The parameters to `~^` (which are numbers) change the condition used to terminate. For a single parameter, termination is when that value is zero (notice this makes plain `~^` equivalent to `~#^`). For two parameters, termination is when those two are equal. For three parameters, termination is when *val1* $\leq$ *val2* and *val2* $\leq$ *val3*.

`~q` Inquiry message. Insert a copyright message into the output.

`~:q` inserts the format implementation version.

It's an error if there are not enough arguments for the escapes in the format string, but any excess arguments are ignored.

Iterations `~{ ~}` and conditionals `~[ ~; ~]` can be nested, but must be properly nested, meaning the inner form must be entirely within the outer form. So it's not possible, for instance, to try to conditionalize the endpoint of an iteration.

```
(format #t "~{ ~[ ... ~] ~}" ...)        ;; good
(format #t "~{ ~[ ... ~} ... ~]" ...)    ;; bad
```

The same applies to case conversions `~( ~)`, they must properly nest with respect to iterations and conditionals (though currently a case conversion cannot nest within another case conversion).

When a sub-format (`~?`) is used, that sub-format string must be self-contained. It cannot for instance give a `~{` to begin an iteration form and have the `~}` up in the originating format, or similar.

Guile contains a `format` procedure even when the module `(ice-9 format)` is not loaded. The default `format` is `simple-format` (see Section 5.12.3 [Writing], page 273), it doesn't support all escape sequences documented in this section, and will signal an error if you try to use one of them. The reason for two versions is that the full `format` is fairly large and requires some time to load. `simple-format` is often adequate too.

6.9 File Tree Walk

The functions in this section traverse a tree of files and directories, in a fashion similar to the C `ftw` and `nftw` routines (see section "Working with Directory Trees" in *GNU C Library Reference Manual*).

```
(use-modules (ice-9 ftw))
```

`ftw` *startname proc* [*'hash-size n*] [Function]

Walk the filesystem tree descending from *startname*, calling *proc* for each file and directory.

Hard links and symbolic links are followed. A file or directory is reported to *proc* only once, and skipped if seen again in another place. One consequence of this is that `ftw` is safe against circularly linked directory structures.

Each *proc* call is `(`*proc* `filename statinfo flag)` and it should return `#t` to continue, or any other value to stop.

filename is the item visited, being *startname* plus a further path and the name of the item. *statinfo* is the return from `stat` (see Section 6.2.3 [File System], page 381) on *filename*. *flag* is one of the following symbols,

`regular` *filename* is a file, this includes special files like devices, named pipes, etc.

`directory`
filename is a directory.

`invalid-stat`
An error occurred when calling `stat`, so nothing is known. *statinfo* is `#f` in this case.

`directory-not-readable`
filename is a directory, but one which cannot be read and hence won't be recursed into.

`symlink` *filename* is a dangling symbolic link. Symbolic links are normally followed and their target reported, the link itself is reported if the target does not exist.

The return value from `ftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from *proc* which caused the stop.

Optional argument symbol `hash-size` and an integer can be given to set the size of the hash table used to track items already visited. (see Section 5.6.12.2 [Hash Table Reference], page 217)

In the current implementation, returning non-`#t` from *proc* is the only valid way to terminate `ftw`. *proc* must not use `throw` or similar to escape.

`nftw` *startname proc* [*'chdir*] [*'depth*] [*'hash-size n*] [*'mount*] [*'physical*] [Function]

Walk the filesystem tree starting at *startname*, calling *proc* for each file and directory. `nftw` has extra features over the basic `ftw` described above.

Like `ftw`, hard links and symbolic links are followed. A file or directory is reported to *proc* only once, and skipped if seen again in another place. One consequence of this is that `nftw` is safe against circular linked directory structures.

Each *proc* call is `(proc filename statinfo flag base level)` and it should return `#t` to continue, or any other value to stop.

filename is the item visited, being *startname* plus a further path and the name of the item. *statinfo* is the return from `stat` on *filename* (see Section 6.2.3 [File System], page 381). *base* is an integer offset into *filename* which is where the basename for this item begins. *level* is an integer giving the directory nesting level, starting from 0 for the contents of *startname* (or that item itself if it's a file). *flag* is one of the following symbols,

`regular` *filename* is a file, including special files like devices, named pipes, etc.

`directory`
filename is a directory.

`directory-processed`
filename is a directory, and its contents have all been visited. This flag is given instead of `directory` when the `depth` option below is used.

`invalid-stat`
An error occurred when applying `stat` to *filename*, so nothing is known about it. *statinfo* is `#f` in this case.

`directory-not-readable`
filename is a directory, but one which cannot be read and hence won't be recursed into.

`stale-symlink`
filename is a dangling symbolic link. Links are normally followed and their target reported, the link itself is reported if its target does not exist.

`symlink` When the `physical` option described below is used, this indicates *filename* is a symbolic link whose target exists (and is not being followed).

The following optional arguments can be given to modify the way `nftw` works. Each is passed as a symbol (and `hash-size` takes a following integer value).

`chdir`
: Change to the directory containing the item before calling *proc*. When `nftw` returns the original current directory is restored.

  Under this option, generally the *base* parameter to each *proc* call should be used to pick out the base part of the *filename*. The *filename* is still a path but with a changed directory it won't be valid (unless the *startname* directory was absolute).

`depth`
: Visit files "depth first", meaning *proc* is called for the contents of each directory before it's called for the directory itself. Normally a directory is reported first, then its contents.

  Under this option, the *flag* to *proc* for a directory is `directory-processed` instead of `directory`.

`hash-size` *n*
: Set the size of the hash table used to track items already visited. (see Section 5.6.12.2 [Hash Table Reference], page 217)

`mount`
: Don't cross a mount point, meaning only visit items on the same filesystem as *startname* (ie. the same `stat:dev`).

`physical`
: Don't follow symbolic links, instead report them to *proc* as `symlink`. Dangling links (those whose target doesn't exist) are still reported as `stale-symlink`.

The return value from `nftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from *proc* which caused the stop.

In the current implementation, returning non-`#t` from *proc* is the only valid way to terminate `ftw`. *proc* must not use `throw` or similar to escape.

6.10 Queues

The functions in this section are provided by

```
(use-modules (ice-9 q))
```

This module implements queues holding arbitrary scheme objects and designed for efficient first-in / first-out operations.

`make-q` creates a queue, and objects are entered and removed with `enq!` and `deq!`. `q-push!` and `q-pop!` can be used too, treating the front of the queue like a stack.

`make-q` [Scheme Procedure]
: Return a new queue.

`q?` *obj* [Scheme Procedure]
: Return `#t` if *obj* is a queue, or `#f` if not.

  Note that queues are not a distinct class of objects but are implemented with cons cells. For that reason certain list structures can get `#t` from `q?`.

`enq!` *q obj* [Scheme Procedure]

Add *obj* to the rear of *q*, and return *q*.

`deq!` *q* [Scheme Procedure]
`q-pop!` *q* [Scheme Procedure]

Remove and return the front element from *q*. If *q* is empty, a `q-empty` exception is thrown.

`deq!` and `q-pop!` are the same operation, the two names just let an application match `enq!` with `deq!`, or `q-push!` with `q-pop!`.

`q-push!` *q obj* [Scheme Procedure]

Add *obj* to the front of *q*, and return *q*.

`q-length` *q* [Scheme Procedure]

Return the number of elements in *q*.

`q-empty?` *q* [Scheme Procedure]

Return true if *q* is empty.

`q-empty-check` *q* [Scheme Procedure]

Throw a `q-empty` exception if *q* is empty.

`q-front` *q* [Scheme Procedure]

Return the first element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-rear` *q* [Scheme Procedure]

Return the last element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-remove!` *q obj* [Scheme Procedure]

Remove all occurences of *obj* from *q*, and return *q*. *obj* is compared to queue elements using `eq?`.

The `q-empty` exceptions described above are thrown just as `(throw 'q-empty)`, there's no message etc like an error throw.

A queue is implemented as a cons cell, the `car` containing a list of queued elements, and the `cdr` being the last cell in that list (for ease of enqueuing).

```
(list . last-cell)
```

If the queue is empty, *list* is the empty list and *last-cell* is `#f`.

An application can directly access the queue list if desired, for instance to search the elements or to insert at a specific point.

`sync-q!` *q* [Scheme Procedure]

Recompute the *last-cell* field in *q*.

All the operations above maintain *last-cell* as described, so normally there's no need for `sync-q!`. But if an application modifies the queue *list* then it must either maintain *last-cell* similarly, or call `sync-q!` to recompute it.

6.11 Streams

A stream represents a sequence of values, each of which is calculated only when required. This allows large or even infinite sequences to be represented and manipulated with familiar operations like "car", "cdr", "map" or "fold". In such manipulations only as much as needed is actually held in memory at any one time. The functions in this section are available from

```
(use-modules (ice-9 streams))
```

Streams are implemented using promises (see Section 5.13.5 [Delayed Evaluation], page 294), which is how the underlying calculation of values is made only when needed, and the values then retained so the calculation is not repeated.

Here is a simple example producing a stream of all odd numbers,

```
(define odds (make-stream (lambda (state)
                            (cons state (+ state 2)))
                          1))
(stream-car odds)              ⇒ 1
(stream-car (stream-cdr odds)) ⇒ 3
```

`stream-map` could be used to derive a stream of odd squares,

```
(define (square n) (* n n))
(define oddsquares (stream-map square odds))
```

These are infinite sequences, so it's not possible to convert them to a list, but they could be printed (infinitely) with for example

```
(stream-for-each (lambda (n sq)
                   (format #t "~a squared is ~a\n" n sq))
                 odds oddsquares)
⊣
1 squared is 1
3 squared is 9
5 squared is 25
7 squared is 49
...
```

`make-stream` *proc initial-state* [Function]

Return a new stream, formed by calling *proc* successively.

Each call is `(proc state)`, it should return a pair, the `car` being the value for the stream, and the `cdr` being the new *state* for the next call. For the first call *state* is the given *initial-state*. At the end of the stream, *proc* should return some non-pair object.

`stream-car` *stream* [Function]

Return the first element from *stream*. *stream* must not be empty.

`stream-cdr` *stream* [Function]

Return a stream which is the second and subsequent elements of *stream*. *stream* must not be empty.

`stream-null?` *stream* [Function]

Return true if *stream* is empty.

`list->stream` *list* [Function]
`vector->stream` *vector* [Function]

Return a stream with the contents of *list* or *vector*.

list or *vector* should not be modified subsequently, since it's unspecified whether changes there will be reflected in the stream returned.

`port->stream` *port readproc* [Function]

Return a stream which is the values obtained by reading from *port* using *readproc*. Each read call is `(`*`readproc port`*`)`, and it should return an EOF object (see Section 5.12.2 [Reading], page 272) at the end of input.

For example a stream of characters from a file,

```
(port->stream (open-input-file "/foo/bar.txt") read-char)
```

`stream->list` *stream* [Function]

Return a list which is the entire contents of *stream*.

`stream->reversed-list` *stream* [Function]

Return a list which is the entire contents of *stream*, but in reverse order.

`stream->list&length` *stream* [Function]

Return two values (see Section 5.11.6 [Multiple Values], page 257), being firstly a list which is the entire contents of *stream*, and secondly the number of elements in that list.

`stream->reversed-list&length` *stream* [Function]

Return two values (see Section 5.11.6 [Multiple Values], page 257) being firstly a list which is the entire contents of *stream*, but in reverse order, and secondly the number of elements in that list.

`stream->vector` *stream* [Function]

Return a vector which is the entire contents of *stream*.

`stream-fold` *proc init stream0 . . . streamN* [Function]

Apply *proc* successively over the elements of the given streams, from first to last until the end of the shortest stream is reached. Return the result from the last *proc* call.

Each call is `(`*`proc`*` elem0 ... elemN prev)`, where each *elem* is from the corresponding *stream*. *prev* is the return from the previous *proc* call, or the given *init* for the first call.

`stream-for-each` *proc stream0 . . . streamN* [Function]

Call *proc* on the elements from the given *stream*s. The return value is unspecified.

Each call is `(`*`proc`*` elem0 ... elemN)`, where each *elem* is from the corresponding *stream*. `stream-for-each` stops when it reaches the end of the shortest *stream*.

`stream-map` *proc stream0 . . . streamN* [Function]

Return a new stream which is the results of applying *proc* to the elements of the given *stream*s.

Each call is `(`*`proc`*` elem0 ... elemN)`, where each *elem* is from the corresponding *stream*. The new stream ends when the end of the shortest given *stream* is reached.

6.12 Buffered Input

The following functions are provided by

```
(use-modules (ice-9 buffered-input))
```

A buffered input port allows a reader function to return chunks of characters which are to be handed out on reading the port. A notion of further input for an application level logical expression is maintained too, and passed through to the reader.

`make-buffered-input-port` *reader* [Function]

Create an input port which returns characters obtained from the given *reader* function. *reader* is called (*reader* cont), and should return a string or an EOF object.

The new port gives precisely the characters returned by *reader*, nothing is added, so if any newline characters or other separators are desired they must come from the reader function.

The *cont* parameter to *reader* is `#f` for initial input, or `#t` when continuing an expression. This is an application level notion, set with `set-buffered-input-continuation?!` below. If the user has entered a partial expression then it allows *reader* for instance to give a different prompt to show more is required.

`make-line-buffered-input-port` *reader* [Function]

Create an input port which returns characters obtained from the specified *reader* function, similar to `make-buffered-input-port` above, but where *reader* is expected to be a line-oriented.

reader is called (*reader* cont), and should return a string or an EOF object as above. Each string is a line of input without a newline character, the port code inserts a newline after each string.

`set-buffered-input-continuation?!` *port cont* [Function]

Set the input continuation flag for a given buffered input *port*.

An application uses this by calling with a *cont* flag of `#f` when beginning to read a new logical expression. For example with the Scheme `read` function (see Section 5.13.2 [Scheme Read], page 290),

```
(define my-port (make-buffered-input-port my-reader))

(set-buffered-input-continuation?! my-port #f)
(let ((obj (read my-port)))
  ...
```

6.13 Expect

The macros in this section are made available with:

```
(use-modules (ice-9 expect))
```

`expect` is a macro for selecting actions based on the output from a port. The name comes from a tool of similar functionality by Don Libes. Actions can be taken when a particular string is matched, when a timeout occurs, or when end-of-file is seen on the port. The `expect` macro is described below; `expect-strings` is a front-end to `expect` based on regexec (see the regular expression documentation).

`expect-strings` *clause* . . . [Macro]

By default, `expect-strings` will read from the current input port. The first term in each clause consists of an expression evaluating to a string pattern (regular expression). As characters are read one-by-one from the port, they are accumulated in a buffer string which is matched against each of the patterns. When a pattern matches, the remaining expression(s) in the clause are evaluated and the value of the last is returned. For example:

```
(with-input-from-file "/etc/passwd"
  (lambda ()
    (expect-strings
      ("^nobody" (display "Got a nobody user.\n")
                 (display "That's no problem.\n"))
      ("^daemon" (display "Got a daemon user.\n")))))
```

The regular expression is compiled with the `REG_NEWLINE` flag, so that the ^ and $ anchors will match at any newline, not just at the start and end of the string.

There are two other ways to write a clause:

The expression(s) to evaluate can be omitted, in which case the result of the regular expression match (converted to strings, as obtained from regexec with match-pick set to "") will be returned if the pattern matches.

The symbol `=>` can be used to indicate that the expression is a procedure which will accept the result of a successful regular expression match. E.g.,

```
("^daemon" => write)
("^d(aemon)" => (lambda args (for-each write args)))
("^da(em)on" => (lambda (all sub)
                  (write all) (newline)
                  (write sub) (newline)))
```

The order of the substrings corresponds to the order in which the opening brackets occur.

A number of variables can be used to control the behaviour of `expect` (and `expect-strings`). Most have default top-level bindings to the value `#f`, which produces the default behaviour. They can be redefined at the top level or locally bound in a form enclosing the expect expression.

`expect-port`
: A port to read characters from, instead of the current input port.

`expect-timeout`
: `expect` will terminate after this number of seconds, returning `#f` or the value returned by expect-timeout-proc.

`expect-timeout-proc`
: A procedure called if timeout occurs. The procedure takes a single argument: the accumulated string.

`expect-eof-proc`
: A procedure called if end-of-file is detected on the input port. The procedure takes a single argument: the accumulated string.

`expect-char-proc`
: A procedure to be called every time a character is read from the port. The procedure takes a single argument: the character which was read.

`expect-strings-compile-flags`
: Flags to be used when compiling a regular expression, which are passed to `make-regexp` See Section 5.5.6.1 [Regexp Functions], page 146. The default value is `regexp/newline`.

`expect-strings-exec-flags`
: Flags to be used when executing a regular expression, which are passed to regexp-exec See Section 5.5.6.1 [Regexp Functions], page 146. The default value is `regexp/noteol`, which prevents `$` from matching the end of the string while it is still accumulating, but still allows it to match after a line break or at the end of file.

Here's an example using all of the variables:

```
(let ((expect-port (open-input-file "/etc/passwd"))
      (expect-timeout 1)
      (expect-timeout-proc
        (lambda (s) (display "Times up!\n")))
      (expect-eof-proc
        (lambda (s) (display "Reached the end of the file!\n")))
      (expect-char-proc display)
      (expect-strings-compile-flags (logior regexp/newline regexp/icase))
      (expect-strings-exec-flags 0))
   (expect-strings
     ("^nobody"  (display "Got a nobody user\n"))))
```

`expect` *clause* ... [Macro]

`expect` is used in the same way as `expect-strings`, but tests are specified not as patterns, but as procedures. The procedures are called in turn after each character is read from the port, with two arguments: the value of the accumulated string and a flag to indicate whether end-of-file has been reached. The flag will usually be `#f`, but if end-of-file is reached, the procedures are called an additional time with the final accumulated string and `#t`.

The test is successful if the procedure returns a non-false value.

If the `=>` syntax is used, then if the test succeeds it must return a list containing the arguments to be provided to the corresponding expression.

In the following example, a string will only be matched at the beginning of the file:

```
(let ((expect-port (open-input-file "/etc/passwd")))
  (expect
     ((lambda (s eof?) (string=? s "fnord!"))
       (display "Got a nobody user!\n"))))
```

The control variables described for `expect-strings` also influence the behaviour of `expect`, with the exception of variables whose names begin with `expect-strings-`.

6.14 The Scheme shell (scsh)

An incomplete port of the Scheme shell (scsh) is available for Guile as a separate package. The current status of guile-scsh can be found at `http://arglist.com/guile/`.

For information about scsh see `http://www.scsh.net/`.

The closest emulation of scsh can be obtained by running:

```
(load-from-path "scsh/init")
```

See the USAGE file supplied with guile-scsh for more details.

Appendix A Data Representation in Guile

by Jim Blandy

[Due to the rather non-orthogonal and performance-oriented nature of the SCM interface, you need to understand SCM internals *before* you can use the SCM API. That's why this chapter comes first.]

[NOTE: this is Jim Blandy's essay almost entirely unmodified. It has to be adapted to fit this manual smoothly.]

In order to make sense of Guile's SCM_ functions, or read libguile's source code, it's essential to have a good grasp of how Guile actually represents Scheme values. Otherwise, a lot of the code, and the conventions it follows, won't make very much sense. This essay is meant to provide the background necessary to read and write C code that manipulates Scheme values in a way that is compatible with libguile.

We assume you know both C and Scheme, but we do not assume you are familiar with Guile's implementation.

A.1 Data Representation in Scheme

Scheme is a latently-typed language; this means that the system cannot, in general, determine the type of a given expression at compile time. Types only become apparent at run time. Variables do not have fixed types; a variable may hold a pair at one point, an integer at the next, and a thousand-element vector later. Instead, values, not variables, have fixed types.

In order to implement standard Scheme functions like `pair?` and `string?` and provide garbage collection, the representation of every value must contain enough information to accurately determine its type at run time. Often, Scheme systems also use this information to determine whether a program has attempted to apply an operation to an inappropriately typed value (such as taking the `car` of a string).

Because variables, pairs, and vectors may hold values of any type, Scheme implementations use a uniform representation for values — a single type large enough to hold either a complete value or a pointer to a complete value, along with the necessary typing information.

The following sections will present a simple typing system, and then make some refinements to correct its major weaknesses. However, this is not a description of the system Guile actually uses. It is only an illustration of the issues Guile's system must address. We provide all the information one needs to work with Guile's data in Section A.2 [How Guile does it], page 489.

A.1.1 A Simple Representation

The simplest way to meet the above requirements in C would be to represent each value as a pointer to a structure containing a type indicator, followed by a union carrying the real value. Assuming that `SCM` is the name of our universal type, we can write:

```
enum type { integer, pair, string, vector, ... };

typedef struct value *SCM;
```

```
struct value {
  enum type type;
  union {
    int integer;
    struct { SCM car, cdr; } pair;
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};
```

with the ellipses replaced with code for the remaining Scheme types.

This representation is sufficient to implement all of Scheme's semantics. If *x* is an `SCM` value:

- To test if *x* is an integer, we can write `x->type == integer`.
- To find its value, we can write `x->value.integer`.
- To test if *x* is a vector, we can write `x->type == vector`.
- If we know *x* is a vector, we can write `x->value.vector.elts[0]` to refer to its first element.
- If we know *x* is a pair, we can write `x->value.pair.car` to extract its car.

A.1.2 Faster Integers

Unfortunately, the above representation has a serious disadvantage. In order to return an integer, an expression must allocate a `struct value`, initialize it to represent that integer, and return a pointer to it. Furthermore, fetching an integer's value requires a memory reference, which is much slower than a register reference on most processors. Since integers are extremely common, this representation is too costly, in both time and space. Integers should be very cheap to create and manipulate.

One possible solution comes from the observation that, on many architectures, structures must be aligned on a four-byte boundary. (Whether or not the machine actually requires it, we can write our own allocator for `struct value` objects that assures this is true.) In this case, the lower two bits of the structure's address are known to be zero.

This gives us the room we need to provide an improved representation for integers. We make the following rules:

- If the lower two bits of an `SCM` value are zero, then the SCM value is a pointer to a `struct value`, and everything proceeds as before.
- Otherwise, the `SCM` value represents an integer, whose value appears in its upper bits.

Here is C code implementing this convention:

```
enum type { pair, string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
```

```
      union {
        struct { SCM car, cdr; } pair;
        struct { int length; char *elts; } string;
        struct { int length; SCM  *elts; } vector;
        ...
      } value;
    };

    #define POINTER_P(x) (((int) (x) & 3) == 0)
    #define INTEGER_P(x) (! POINTER_P (x))

    #define GET_INTEGER(x)  ((int) (x) >> 2)
    #define MAKE_INTEGER(x) ((SCM) (((x) << 2) | 1))
```

Notice that `integer` no longer appears as an element of `enum type`, and the union has lost its `integer` member. Instead, we use the `POINTER_P` and `INTEGER_P` macros to make a coarse classification of values into integers and non-integers, and do further type testing as before.

Here's how we would answer the questions posed above (again, assume *x* is an `SCM` value):

- To test if *x* is an integer, we can write `INTEGER_P (x)`.
- To find its value, we can write `GET_INTEGER (x)`.
- To test if *x* is a vector, we can write:

  ```
  POINTER_P (x) && x->type == vector
  ```

  Given the new representation, we must make sure *x* is truly a pointer before we dereference it to determine its complete type.
- If we know *x* is a vector, we can write `x->value.vector.elts[0]` to refer to its first element, as before.
- If we know *x* is a pair, we can write `x->value.pair.car` to extract its car, just as before.

This representation allows us to operate more efficiently on integers than the first. For example, if *x* and *y* are known to be integers, we can compute their sum as follows:

```
MAKE_INTEGER (GET_INTEGER (x) + GET_INTEGER (y))
```

Now, integer math requires no allocation or memory references. Most real Scheme systems actually use an even more efficient representation, but this essay isn't about bit-twiddling. (Hint: what if pointers had `01` in their least significant bits, and integers had `00`?)

A.1.3 Cheaper Pairs

However, there is yet another issue to confront. Most Scheme heaps contain more pairs than any other type of object; Jonathan Rees says that pairs occupy 45% of the heap in his Scheme implementation, Scheme 48. However, our representation above spends three `SCM`-sized words per pair — one for the type, and two for the CAR and CDR. Is there any way to represent pairs using only two words?

Let us refine the convention we established earlier. Let us assert that:

- If the bottom two bits of an `SCM` value are `#b00`, then it is a pointer, as before.
- If the bottom two bits are `#b01`, then the upper bits are an integer. This is a bit more restrictive than before.
- If the bottom two bits are `#b10`, then the value, with the bottom two bits masked out, is the address of a pair.

Here is the new C code:

```
enum type { string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};

struct pair {
  SCM car, cdr;
};

#define POINTER_P(x) (((int) (x) & 3) == 0)

#define INTEGER_P(x)  (((int) (x) & 3) == 1)
#define GET_INTEGER(x)  ((int) (x) >> 2)
#define MAKE_INTEGER(x) ((SCM) (((x) << 2) | 1))

#define PAIR_P(x) (((int) (x) & 3) == 2)
#define GET_PAIR(x) ((struct pair *) ((int) (x) & ~3))
```

Notice that `enum type` and `struct value` now only contain provisions for vectors and strings; both integers and pairs have become special cases. The code above also assumes that an `int` is large enough to hold a pointer, which isn't generally true.

Our list of examples is now as follows:

- To test if *x* is an integer, we can write `INTEGER_P (`*x*`)`; this is as before.
- To find its value, we can write `GET_INTEGER (`*x*`)`, as before.
- To test if *x* is a vector, we can write:

  ```
  POINTER_P (x) && x->type == vector
  ```

  We must still make sure that *x* is a pointer to a `struct value` before dereferencing it to find its type.
- If we know *x* is a vector, we can write *x*`->value.vector.elts[0]` to refer to its first element, as before.

- We can write `PAIR_P (x)` to determine if *x* is a pair, and then write `GET_PAIR (x)->car` to refer to its car.

This change in representation reduces our heap size by 15%. It also makes it cheaper to decide if a value is a pair, because no memory references are necessary; it suffices to check the bottom two bits of the `SCM` value. This may be significant when traversing lists, a common activity in a Scheme system.

Again, most real Scheme systems use a slightly different implementation; for example, if GET_PAIR subtracts off the low bits of `x`, instead of masking them off, the optimizer will often be able to combine that subtraction with the addition of the offset of the structure member we are referencing, making a modified pointer as fast to use as an unmodified pointer.

A.1.4 Guile Is Hairier

We originally started with a very simple typing system — each object has a field that indicates its type. Then, for the sake of efficiency in both time and space, we moved some of the typing information directly into the `SCM` value, and left the rest in the `struct value`. Guile itself employs a more complex hierarchy, storing finer and finer gradations of type information in different places, depending on the object's coarser type.

In the author's opinion, Guile could be simplified greatly without significant loss of efficiency, but the simplified system would still be more complex than what we've presented above.

A.2 How Guile does it

Here we present the specifics of how Guile represents its data. We don't go into complete detail; an exhaustive description of Guile's system would be boring, and we do not wish to encourage people to write code which depends on its details anyway. We do, however, present everything one need know to use Guile's data.

This section is in limbo. It used to document the 'low-level' C API of Guile that was used both by clients of libguile and by libguile itself.

In the future, clients should only need to look into the sections Chapter 4 [Programming in C], page 55 and Chapter 5 [API Reference], page 93. This section will in the end only contain stuff about the internals of Guile.

A.2.1 General Rules

Any code which operates on Guile datatypes must `#include` the header file `<libguile.h>`. This file contains a definition for the `SCM` typedef (Guile's universal type, as in the examples above), and definitions and declarations for a host of macros and functions that operate on `SCM` values.

All identifiers declared by `<libguile.h>` begin with `scm_` or `SCM_`.

The functions described here generally check the types of their `SCM` arguments, and signal an error if their arguments are of an inappropriate type. Macros generally do not, unless that is their specified purpose. You must verify their argument types beforehand, as necessary.

Macros and functions that return a boolean value have names ending in `P` or `_p` (for "predicate"). Those that return a negated boolean value have names starting with `SCM_N`. For example, `SCM_IMP (x)` is a predicate which returns non-zero iff *x* is an immediate value (an `IM`). `SCM_NCONSP (x)` is a predicate which returns non-zero iff *x* is *not* a pair object (a `CONS`).

A.2.2 Conservative Garbage Collection

Aside from the latent typing, the major source of constraints on a Scheme implementation's data representation is the garbage collector. The collector must be able to traverse every live object in the heap, to determine which objects are not live.

There are many ways to implement this, but Guile uses an algorithm called *mark and sweep*. The collector scans the system's global variables and the local variables on the stack to determine which objects are immediately accessible by the C code. It then scans those objects to find the objects they point to, *et cetera*. The collector sets a *mark bit* on each object it finds, so each object is traversed only once. This process is called *tracing*.

When the collector can find no unmarked objects pointed to by marked objects, it assumes that any objects that are still unmarked will never be used by the program (since there is no path of dereferences from any global or local variable that reaches them) and deallocates them.

In the above paragraphs, we did not specify how the garbage collector finds the global and local variables; as usual, there are many different approaches. Frequently, the programmer must maintain a list of pointers to all global variables that refer to the heap, and another list (adjusted upon entry to and exit from each function) of local variables, for the collector's benefit.

The list of global variables is usually not too difficult to maintain, since global variables are relatively rare. However, an explicitly maintained list of local variables (in the author's personal experience) is a nightmare to maintain. Thus, Guile uses a technique called *conservative garbage collection*, to make the local variable list unnecessary.

The trick to conservative collection is to treat the stack as an ordinary range of memory, and assume that *every* word on the stack is a pointer into the heap. Thus, the collector marks all objects whose addresses appear anywhere in the stack, without knowing for sure how that word is meant to be interpreted.

Obviously, such a system will occasionally retain objects that are actually garbage, and should be freed. In practice, this is not a problem. The alternative, an explicitly maintained list of local variable addresses, is effectively much less reliable, due to programmer error.

To accommodate this technique, data must be represented so that the collector can accurately determine whether a given stack word is a pointer or not. Guile does this as follows:

- Every heap object has a two-word header, called a *cell*. Some objects, like pairs, fit entirely in a cell's two words; others may store pointers to additional memory in either of the words. For example, strings and vectors store their length in the first word, and a pointer to their elements in the second.
- Guile allocates whole arrays of cells at a time, called *heap segments*. These segments are always allocated so that the cells they contain fall on eight-byte boundaries, or

whatever is appropriate for the machine's word size. Guile keeps all cells in a heap segment initialized, whether or not they are currently in use.

- Guile maintains a sorted table of heap segments.

Thus, given any random word *w* fetched from the stack, Guile's garbage collector can consult the table to see if *w* falls within a known heap segment, and check *w*'s alignment. If both tests pass, the collector knows that *w* is a valid pointer to a cell, intentional or not, and proceeds to trace the cell.

Note that heap segments do not contain all the data Guile uses; cells for objects like vectors and strings contain pointers to other memory areas. However, since those pointers are internal, and not shared among many pieces of code, it is enough for the collector to find the cell, and then use the cell's type to find more pointers to trace.

A.2.3 Immediates vs Non-immediates

Guile classifies Scheme objects into two kinds: those that fit entirely within an `SCM`, and those that require heap storage.

The former class are called *immediates*. The class of immediates includes small integers, characters, boolean values, the empty list, the mysterious end-of-file object, and some others.

The remaining types are called, not surprisingly, *non-immediates*. They include pairs, procedures, strings, vectors, and all other data types in Guile.

`int SCM_IMP` (*SCM x*) [Macro]

Return non-zero iff *x* is an immediate object.

`int SCM_NIMP` (*SCM x*) [Macro]

Return non-zero iff *x* is a non-immediate object. This is the exact complement of `SCM_IMP`, above.

Note that for versions of Guile prior to 1.4 it was necessary to use the `SCM_NIMP` macro before calling a finer-grained predicate to determine *x*'s type, such as `SCM_CONSP` or `SCM_VECTORP`. This is no longer required: the definitions of all Guile type predicates now include a call to `SCM_NIMP` where necessary.

A.2.4 Immediate Datatypes

The following datatypes are immediate values; that is, they fit entirely within an `SCM` value. The `SCM_IMP` and `SCM_NIMP` macros will distinguish these from non-immediates; see Section A.2.3 [Immediates vs Non-immediates], page 491 for an explanation of the distinction.

Note that the type predicates for immediate values work correctly on any `SCM` value; you do not need to call `SCM_IMP` first, to establish that a value is immediate.

A.2.4.1 Integers

Here are functions for operating on small integers, that fit within an `SCM`. Such integers are called *immediate numbers*, or *INUMs*. In general, INUMs occupy all but two bits of an `SCM`.

Bignums and floating-point numbers are non-immediate objects, and have their own, separate accessors. The functions here will not work on them. This is not as much of a problem as you might think, however, because the system never constructs bignums that could fit in an INUM, and never uses floating point values for exact integers.

`int SCM_INUMP` (*SCM* `x`) [Macro]

Return non-zero iff *x* is a small integer value.

`int SCM_NINUMP` (*SCM* `x`) [Macro]

The complement of SCM_INUMP.

`int SCM_INUM` (*SCM* `x`) [Macro]

Return the value of *x* as an ordinary, C integer. If *x* is not an INUM, the result is undefined.

`SCM SCM_MAKINUM` (*int* `i`) [Macro]

Given a C integer *i*, return its representation as an `SCM`. This function does not check for overflow.

A.2.4.2 Characters

Here are functions for operating on characters.

`int SCM_CHARP` (*SCM* `x`) [Macro]

Return non-zero iff *x* is a character value.

`unsigned int SCM_CHAR` (*SCM* `x`) [Macro]

Return the value of `x` as a C character. If *x* is not a Scheme character, the result is undefined.

`SCM SCM_MAKE_CHAR` (*int* `c`) [Macro]

Given a C character *c*, return its representation as a Scheme character value.

A.2.4.3 Booleans

Booleans are represented as two specific immediate SCM values, `SCM_BOOL_T` and `SCM_BOOL_F`. See Section 5.5.1 [Booleans], page 99, for more information.

A.2.4.4 Unique Values

The immediate values that are neither small integers, characters, nor booleans are all unique values — that is, datatypes with only one instance.

`SCM SCM_EOL` [Macro]

The Scheme empty list object, or "End Of List" object, usually written in Scheme as `'()`.

`SCM SCM_EOF_VAL` [Macro]

The Scheme end-of-file value. It has no standard written representation, for obvious reasons.

`SCM SCM_UNSPECIFIED` [Macro]

The value returned by expressions which the Scheme standard says return an "unspecified" value.

This is sort of a weirdly literal way to take things, but the standard read-eval-print loop prints nothing when the expression returns this value, so it's not a bad idea to return this when you can't think of anything else helpful.

`SCM SCM_UNDEFINED` [Macro]

The “undefined” value. Its most important property is that is not equal to any valid Scheme value. This is put to various internal uses by C code interacting with Guile.

For example, when you write a C function that is callable from Scheme and which takes optional arguments, the interpreter passes `SCM_UNDEFINED` for any arguments you did not receive.

We also use this to mark unbound variables.

`int SCM_UNBNDP (`*SCM* `x)` [Macro]

Return true if *x* is `SCM_UNDEFINED`. Apply this to a symbol's value to see if it has a binding as a global variable.

A.2.5 Non-immediate Datatypes

A non-immediate datatype is one which lives in the heap, either because it cannot fit entirely within a `SCM` word, or because it denotes a specific storage location (in the nomenclature of the Revised^5 Report on Scheme).

The `SCM_IMP` and `SCM_NIMP` macros will distinguish these from immediates; see Section A.2.3 [Immediates vs Non-immediates], page 491.

Given a cell, Guile distinguishes between pairs and other non-immediate types by storing special *tag* values in a non-pair cell's car, that cannot appear in normal pairs. A cell with a non-tag value in its car is an ordinary pair. The type of a cell with a tag in its car depends on the tag; the non-immediate type predicates test this value. If a tag value appears elsewhere (in a vector, for example), the heap may become corrupted.

Note how the type information for a non-immediate object is split between the `SCM` word and the cell that the `SCM` word points to. The `SCM` word itself only indicates that the object is non-immediate — in other words stored in a heap cell. The tag stored in the first word of the heap cell indicates more precisely the type of that object.

The type predicates for non-immediate values work correctly on any `SCM` value; you do not need to call `SCM_NIMP` first, to establish that a value is non-immediate.

A.2.5.1 Pairs

Pairs are the essential building block of list structure in Scheme. A pair object has two fields, called the *car* and the *cdr*.

It is conventional for a pair's CAR to contain an element of a list, and the CDR to point to the next pair in the list, or to contain `SCM_EOL`, indicating the end of the list. Thus, a set of pairs chained through their CDRs constitutes a singly-linked list. Scheme and libguile define many functions which operate on lists constructed in this fashion, so although lists chained through the CARs of pairs will work fine too, they may be less convenient to manipulate, and receive less support from the community.

Guile implements pairs by mapping the CAR and CDR of a pair directly into the two words of the cell.

`int SCM_CONSP (`*SCM* `x)` [Macro]

Return non-zero iff *x* is a Scheme pair object.

`int SCM_NCONSP (`*SCM* `x)` [Macro]
: The complement of SCM_CONSP.

`SCM scm_cons (`*SCM* `car`, *SCM* `cdr)` [Function]
: Allocate ("CONStruct") a new pair, with *car* and *cdr* as its contents.

The macros below perform no type checking. The results are undefined if *cell* is an immediate. However, since all non-immediate Guile objects are constructed from cells, and these macros simply return the first element of a cell, they actually can be useful on datatypes other than pairs. (Of course, it is not very modular to use them outside of the code which implements that datatype.)

`SCM SCM_CAR (`*SCM* `cell)` [Macro]
: Return the CAR, or first field, of *cell*.

`SCM SCM_CDR (`*SCM* `cell)` [Macro]
: Return the CDR, or second field, of *cell*.

`void SCM_SETCAR (`*SCM* `cell`, *SCM* `x)` [Macro]
: Set the CAR of *cell* to *x*.

`void SCM_SETCDR (`*SCM* `cell`, *SCM* `x)` [Macro]
: Set the CDR of *cell* to *x*.

`SCM SCM_CAAR (`*SCM* `cell)` [Macro]
`SCM SCM_CADR (`*SCM* `cell)` [Macro]
`SCM SCM_CDAR (`*SCM* `cell) ...` [Macro]
`SCM SCM_CDDDDR (`*SCM* `cell)` [Macro]
: Return the CAR of the CAR of *cell*, the CAR of the CDR of *cell*, *et cetera*.

A.2.5.2 Vectors, Strings, and Symbols

Vectors, strings, and symbols have some properties in common. They all have a length, and they all have an array of elements. In the case of a vector, the elements are `SCM` values; in the case of a string or symbol, the elements are characters.

All these types store their length (along with some tagging bits) in the CAR of their header cell, and store a pointer to the elements in their CDR. Thus, the `SCM_CAR` and `SCM_CDR` macros are (somewhat) meaningful when applied to these datatypes.

`int SCM_VECTORP (`*SCM* `x)` [Macro]
: Return non-zero iff *x* is a vector.

`int SCM_STRINGP (`*SCM* `x)` [Macro]
: Return non-zero iff *x* is a string.

`int SCM_SYMBOLP (`*SCM* `x)` [Macro]
: Return non-zero iff *x* is a symbol.

`int SCM_VECTOR_LENGTH (`*SCM* `x)` [Macro]
`int SCM_STRING_LENGTH (`*SCM* `x)` [Macro]
`int SCM_SYMBOL_LENGTH (`*SCM* `x)` [Macro]
: Return the length of the object *x*. The result is undefined if *x* is not a vector, string, or symbol, respectively.

`SCM * SCM_VECTOR_BASE` (*SCM* `x`) [Macro]

Return a pointer to the array of elements of the vector *x*. The result is undefined if *x* is not a vector.

`char * SCM_STRING_CHARS` (*SCM* `x`) [Macro]
`char * SCM_SYMBOL_CHARS` (*SCM* `x`) [Macro]

Return a pointer to the characters of *x*. The result is undefined if *x* is not a symbol or string, respectively.

There are also a few magic values stuffed into memory before a symbol's characters, but you don't want to know about those. What cruft!

Note that `SCM_VECTOR_BASE`, `SCM_STRING_CHARS` and `SCM_SYMBOL_CHARS` return pointers to data within the respective object. Care must be taken that the object is not garbage collected while that data is still being accessed. This is the same as for a smob, See Section 4.4.6 [Remembering During Operations], page 75.

A.2.5.3 Procedures

Guile provides two kinds of procedures: *closures*, which are the result of evaluating a `lambda` expression, and *subrs*, which are C functions packaged up as Scheme objects, to make them available to Scheme programmers.

(There are actually other sorts of procedures: compiled closures, and continuations; see the source code for details about them.)

`SCM scm_procedure_p` (*SCM* `x`) [Function]

Return `SCM_BOOL_T` iff *x* is a Scheme procedure object, of any sort. Otherwise, return `SCM_BOOL_F`.

A.2.5.4 Closures

[FIXME: this needs to be further subbed, but texinfo has no subsubsub]

A closure is a procedure object, generated as the value of a `lambda` expression in Scheme. The representation of a closure is straightforward — it contains a pointer to the code of the lambda expression from which it was created, and a pointer to the environment it closes over.

In Guile, each closure also has a property list, allowing the system to store information about the closure. I'm not sure what this is used for at the moment — the debugger, maybe?

`int SCM_CLOSUREP` (*SCM* `x`) [Macro]

Return non-zero iff *x* is a closure.

`SCM SCM_PROCPROPS` (*SCM* `x`) [Macro]

Return the property list of the closure *x*. The results are undefined if *x* is not a closure.

`void SCM_SETPROCPROPS` (*SCM* `x`, *SCM* `p`) [Macro]

Set the property list of the closure *x* to *p*. The results are undefined if *x* is not a closure.

`SCM SCM_CODE (SCM x)` [Macro]

Return the code of the closure *x*. The result is undefined if *x* is not a closure.

This function should probably only be used internally by the interpreter, since the representation of the code is intimately connected with the interpreter's implementation.

`SCM SCM_ENV (SCM x)` [Macro]

Return the environment enclosed by *x*. The result is undefined if *x* is not a closure.

This function should probably only be used internally by the interpreter, since the representation of the environment is intimately connected with the interpreter's implementation.

A.2.5.5 Subrs

[FIXME: this needs to be further subbed, but texinfo has no subsubsub]

A subr is a pointer to a C function, packaged up as a Scheme object to make it callable by Scheme code. In addition to the function pointer, the subr also contains a pointer to the name of the function, and information about the number of arguments accepted by the C function, for the sake of error checking.

There is no single type predicate macro that recognizes subrs, as distinct from other kinds of procedures. The closest thing is `scm_procedure_p`; see Section A.2.5.3 [Procedures], page 495.

`char * SCM_SNAME (x)` [Macro]

Return the name of the subr *x*. The result is undefined if *x* is not a subr.

`SCM scm_c_define_gsubr (char *name, int req, int opt, int rest, SCM (*function)())` [Function]

Create a new subr object named *name*, based on the C function *function*, make it visible to Scheme the value of as a global variable named *name*, and return the subr object.

The subr object accepts *req* required arguments, *opt* optional arguments, and a *rest* argument iff *rest* is non-zero. The C function *function* should accept `req + opt` arguments, or `req + opt + 1` arguments if `rest` is non-zero.

When a subr object is applied, it must be applied to at least *req* arguments, or else Guile signals an error. *function* receives the subr's first *req* arguments as its first *req* arguments. If there are fewer than *opt* arguments remaining, then *function* receives the value `SCM_UNDEFINED` for any missing optional arguments.

If *rst* is non-zero, then any arguments after the first `req + opt` are packaged up as a list and passed as *function*'s last argument. *function* must not modify that list. (Because when subr is called through `apply` the list is directly from the `apply` argument, which the caller will expect to be unchanged.)

Note that subrs can actually only accept a predefined set of combinations of required, optional, and rest arguments. For example, a subr can take one required argument, or one required and one optional argument, but a subr can't take one required and two optional arguments. It's bizarre, but that's the way the interpreter was written. If the arguments to `scm_c_define_gsubr` do not fit one of the predefined patterns, then `scm_c_define_gsubr` will return a compiled closure object instead of a subr object.

A.2.5.6 Ports

Haven't written this yet, 'cos I don't understand ports yet.

A.2.6 Signalling Type Errors

Every function visible at the Scheme level should aggressively check the types of its arguments, to avoid misinterpreting a value, and perhaps causing a segmentation fault. Guile provides some macros to make this easier.

`void SCM_ASSERT` (*int* `test`, *SCM* `obj`, *unsigned int* `position`, *const char* `*subr`) [Macro]

If *test* is zero, signal a "wrong type argument" error, attributed to the subroutine named *subr*, operating on the value *obj*, which is the *position*'th argument of *subr*.

`int SCM_ARG1` [Macro]
`int SCM_ARG2` [Macro]
`int SCM_ARG3` [Macro]
`int SCM_ARG4` [Macro]
`int SCM_ARG5` [Macro]
`int SCM_ARG6` [Macro]
`int SCM_ARG7` [Macro]

One of the above values can be used for *position* to indicate the number of the argument of *subr* which is being checked. Alternatively, a positive integer number can be used, which allows to check arguments after the seventh. However, for parameter numbers up to seven it is preferable to use `SCM_ARGN` instead of the corresponding raw number, since it will make the code easier to understand.

`int SCM_ARGn` [Macro]

Passing a value of zero or `SCM_ARGn` for *position* allows to leave it unspecified which argument's type is incorrect. Again, `SCM_ARGn` should be preferred over a raw zero constant.

A.2.7 Unpacking the SCM Type

The previous sections have explained how `SCM` values can refer to immediate and non-immediate Scheme objects. For immediate objects, the complete object value is stored in the `SCM` word itself, while for non-immediates, the `SCM` word contains a pointer to a heap cell, and further information about the object in question is stored in that cell. This section describes how the `SCM` type is actually represented and used at the C level.

In fact, there are two basic C data types to represent objects in Guile: `SCM` and `scm_t_bits`.

A.2.7.1 Relationship between `SCM` and `scm_t_bits`

A variable of type `SCM` is guaranteed to hold a valid Scheme object. A variable of type `scm_t_bits`, on the other hand, may hold a representation of a `SCM` value as a C integral type, but may also hold any C value, even if it does not correspond to a valid Scheme object.

For a variable *x* of type `SCM`, the Scheme object's type information is stored in a form that is not directly usable. To be able to work on the type encoding of the scheme value, the `SCM` variable has to be transformed into the corresponding representation as a `scm_t_bits` variable *y* by using the `SCM_UNPACK` macro. Once this has been done, the type of the scheme object *x* can be derived from the content of the bits of the `scm_t_bits` value *y*, in the way illustrated by the example earlier in this chapter (see Section A.1.3 [Cheaper Pairs], page 487). Conversely, a valid bit encoding of a Scheme value as a `scm_t_bits` variable can be transformed into the corresponding `SCM` value using the `SCM_PACK` macro.

A.2.7.2 Immediate objects

A Scheme object may either be an immediate, i.e. carrying all necessary information by itself, or it may contain a reference to a *cell* with additional information on the heap. Although in general it should be irrelevant for user code whether an object is an immediate or not, within Guile's own code the distinction is sometimes of importance. Thus, the following low level macro is provided:

`int SCM_IMP` (*SCM* `x`) [Macro]

A Scheme object is an immediate if it fulfills the `SCM_IMP` predicate, otherwise it holds an encoded reference to a heap cell. The result of the predicate is delivered as a C style boolean value. User code and code that extends Guile should normally not be required to use this macro.

Summary:

- Given a Scheme object *x* of unknown type, check first with `SCM_IMP (x)` if it is an immediate object.
- If so, all of the type and value information can be determined from the `scm_t_bits` value that is delivered by `SCM_UNPACK (x)`.

A.2.7.3 Non-immediate objects

A Scheme object of type `SCM` that does not fulfill the `SCM_IMP` predicate holds an encoded reference to a heap cell. This reference can be decoded to a C pointer to a heap cell using the `SCM2PTR` macro. The encoding of a pointer to a heap cell into a `SCM` value is done using the `PTR2SCM` macro.

`(scm_t_cell *)` *SCM2PTR* (*SCM* `x`) [Macro]

Extract and return the heap cell pointer from a non-immediate `SCM` object *x*.

`SCM PTR2SCM` (*scm_t_cell* * `x`) [Macro]

Return a `SCM` value that encodes a reference to the heap cell pointer *x*.

Note that it is also possible to transform a non-immediate `SCM` value by using `SCM_UNPACK` into a `scm_t_bits` variable. However, the result of `SCM_UNPACK` may not be used as a pointer to a `scm_t_cell`: only `SCM2PTR` is guaranteed to transform a `SCM` object into a valid pointer to a heap cell. Also, it is not allowed to apply `PTR2SCM` to anything that is not a valid pointer to a heap cell.

Summary:

- Only use `SCM2PTR` on `SCM` values for which `SCM_IMP` is false!

- Don't use `(scm_t_cell *) SCM_UNPACK (x)`! Use `SCM2PTR (x)` instead!
- Don't use `PTR2SCM` for anything but a cell pointer!

A.2.7.4 Allocating Cells

Guile provides both ordinary cells with two slots, and double cells with four slots. The following two function are the most primitive way to allocate such cells.

If the caller intends to use it as a header for some other type, she must pass an appropriate magic value in *word_0*, to mark it as a member of that type, and pass whatever value as *word_1*, etc that the type expects. You should generally not need these functions, unless you are implementing a new datatype, and thoroughly understand the code in `<libguile/tags.h>`.

If you just want to allocate pairs, use `scm_cons`.

`SCM scm_cell` (*scm_t_bits word_0, scm_t_bits word_1*) [Function]

Allocate a new cell, initialize the two slots with *word_0* and *word_1*, and return it.

Note that *word_0* and *word_1* are of type `scm_t_bits`. If you want to pass a `SCM` object, you need to use `SCM_UNPACK`.

`SCM scm_double_cell` (*scm_t_bits word_0, scm_t_bits word_1, scm_t_bits word_2, scm_t_bits word_3*) [Function]

Like `scm_cell`, but allocates a double cell with four slots.

A.2.7.5 Heap Cell Type Information

Heap cells contain a number of entries, each of which is either a scheme object of type `SCM` or a raw C value of type `scm_t_bits`. Which of the cell entries contain Scheme objects and which contain raw C values is determined by the first entry of the cell, which holds the cell type information.

`scm_t_bits SCM_CELL_TYPE` (*SCM* `x`) [Macro]

For a non-immediate Scheme object *x*, deliver the content of the first entry of the heap cell referenced by *x*. This value holds the information about the cell type.

`void SCM_SET_CELL_TYPE` (*SCM* `x`, *scm_t_bits* `t`) [Macro]

For a non-immediate Scheme object *x*, write the value *t* into the first entry of the heap cell referenced by *x*. The value *t* must hold a valid cell type.

A.2.7.6 Accessing Cell Entries

For a non-immediate Scheme object *x*, the object type can be determined by reading the cell type entry using the `SCM_CELL_TYPE` macro. For each different type of cell it is known which cell entries hold Scheme objects and which cell entries hold raw C data. To access the different cell entries appropriately, the following macros are provided.

`scm_t_bits SCM_CELL_WORD` (*SCM* `x`, *unsigned int* `n`) [Macro]

Deliver the cell entry *n* of the heap cell referenced by the non-immediate Scheme object *x* as raw data. It is illegal, to access cell entries that hold Scheme objects by using these macros. For convenience, the following macros are also provided.

- SCM_CELL_WORD_0 (*x*) ⇒ SCM_CELL_WORD (*x*, 0)

- SCM_CELL_WORD_1 (*x*) ⇒ SCM_CELL_WORD (*x*, 1)
- ...
- SCM_CELL_WORD_*n* (*x*) ⇒ SCM_CELL_WORD (*x*, *n*)

`SCM SCM_CELL_OBJECT` (*SCM* `x`, *unsigned int* `n`) [Macro]

Deliver the cell entry *n* of the heap cell referenced by the non-immediate Scheme object *x* as a Scheme object. It is illegal, to access cell entries that do not hold Scheme objects by using these macros. For convenience, the following macros are also provided.

- SCM_CELL_OBJECT_0 (*x*) ⇒ SCM_CELL_OBJECT (*x*, 0)
- SCM_CELL_OBJECT_1 (*x*) ⇒ SCM_CELL_OBJECT (*x*, 1)
- ...
- SCM_CELL_OBJECT_*n* (*x*) ⇒ SCM_CELL_OBJECT (*x*, *n*)

`void SCM_SET_CELL_WORD` (*SCM* `x`, *unsigned int* `n`, *scm_t_bits* `w`) [Macro]

Write the raw C value *w* into entry number *n* of the heap cell referenced by the non-immediate Scheme value *x*. Values that are written into cells this way may only be read from the cells using the `SCM_CELL_WORD` macros or, in case cell entry 0 is written, using the `SCM_CELL_TYPE` macro. For the special case of cell entry 0 it has to be made sure that *w* contains a cell type information which does not describe a Scheme object. For convenience, the following macros are also provided.

- SCM_SET_CELL_WORD_0 (*x*, *w*) ⇒ SCM_SET_CELL_WORD (*x*, 0, *w*)
- SCM_SET_CELL_WORD_1 (*x*, *w*) ⇒ SCM_SET_CELL_WORD (*x*, 1, *w*)
- ...
- SCM_SET_CELL_WORD_*n* (*x*, *w*) ⇒ SCM_SET_CELL_WORD (*x*, *n*, *w*)

`void SCM_SET_CELL_OBJECT` (*SCM* `x`, *unsigned int* `n`, *SCM* `o`) [Macro]

Write the Scheme object *o* into entry number *n* of the heap cell referenced by the non-immediate Scheme value *x*. Values that are written into cells this way may only be read from the cells using the `SCM_CELL_OBJECT` macros or, in case cell entry 0 is written, using the `SCM_CELL_TYPE` macro. For the special case of cell entry 0 the writing of a Scheme object into this cell is only allowed if the cell forms a Scheme pair. For convenience, the following macros are also provided.

- SCM_SET_CELL_OBJECT_0 (*x*, *o*) ⇒ SCM_SET_CELL_OBJECT (*x*, 0, *o*)
- SCM_SET_CELL_OBJECT_1 (*x*, *o*) ⇒ SCM_SET_CELL_OBJECT (*x*, 1, *o*)
- ...
- SCM_SET_CELL_OBJECT_*n* (*x*, *o*) ⇒ SCM_SET_CELL_OBJECT (*x*, *n*, *o*)

Summary:

- For a non-immediate Scheme object *x* of unknown type, get the type information by using `SCM_CELL_TYPE (x)`.
- As soon as the cell type information is available, only use the appropriate access methods to read and write data to the different cell entries.

A.2.7.7 Basic Rules for Accessing Cell Entries

For each cell type it is generally up to the implementation of that type which of the corresponding cell entries hold Scheme objects and which hold raw C values. However, there is one basic rule that has to be followed: Scheme pairs consist of exactly two cell entries, which both contain Scheme objects. Further, a cell which contains a Scheme object in it first entry has to be a Scheme pair. In other words, it is not allowed to store a Scheme object in the first cell entry and a non Scheme object in the second cell entry.

`int SCM_CONSP` (*SCM* `x`) [Macro]

> Determine, whether the Scheme object *x* is a Scheme pair, i.e. whether *x* references a heap cell consisting of exactly two entries, where both entries contain a Scheme object. In this case, both entries will have to be accessed using the `SCM_CELL_OBJECT` macros. On the contrary, if the `SCM_CONSP` predicate is not fulfilled, the first entry of the Scheme cell is guaranteed not to be a Scheme value and thus the first cell entry must be accessed using the `SCM_CELL_WORD_0` macro.

Appendix B GNU Free Documentation License

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

B.0.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Indices

Concept Index

This index contains concepts, keywords and non-Schemey names for several features, to make it easier to locate the desired sections.

Procedure Index

This is an alphabetical list of all the procedures and macros in Guile.

When looking for a particular procedure, please look under its Scheme name as well as under its C name. The C name can be constructed from the Scheme names by a simple transformation described in the section See Section 5.1 [API Overview], page 94.

Variable Index

This is an alphabetical list of all the important variables and constants in Guile.

When looking for a particular variable or constant, please look under its Scheme name as well as under its C name. The C name can be constructed from the Scheme names by a simple transformation described in the section See Section 5.1 [API Overview], page 94.

Type Index

This is an alphabetical list of all the important data types defined in the Guile Programmers Manual.

R5RS Index