

Sizzle

Reference Manual
Version 0.0.30

Martin Grabmueller

Copyright © 1999, 2000 Martin Grabmueller

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

1 Introduction

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

Sizzle implements a subset of Scheme as described in the Revised 5 Report on the Algorithmic Language Scheme (referred to as R5RS in this manual). That means that Sizzle employs fully parenthesized prefix notation (as known from Lisp), is statically scoped and handles builtin and user defined procedures as first-class objects. It implements most of Scheme as defined in R5RS, but it does not support complex numbers, arbitrary precision numbers (bignums) or rational numbers. In addition, only outward continuations (a.k.a. escape procedures) are implemented.

That means that Sizzle is a quite powerful language and will (as least I hope so) be useful for a lot of programming tasks.

Most of the text in the reference section of this manual is heavily inspired by R5RS, especially the description of the standard syntactic forms and standard procedures. Some parts, especially documentation on procedures compatible with Guile, have been taken from the Guile documentation strings.

The chapter after this introduction describes Sizzle from the user's point of view, you will learn how to start the interpreter and how to use it interactively as well as how to use it to execute Scheme scripts.

After that, in the third chapter, I will show how to do basic programming in the Scheme programming language and which data types are available and how to make use of them.

The fourth chapter documents the builtin variables which can be used to monitor and manipulate the interpreter's behaviour from Scheme.

The main reference section (chapter 5) documents all procedures as well as the syntactic forms available in the interpreter.

The sixth chapter lists all (known) differences between the Scheme implementation of Sizzle and the standard defined in R5RS.

The manual is closed by a glossary and an index of all functions and variables.

1.1 Sizzle History

I started writing Sizzle simply because I was interested in compiler and interpreter design. A long time ago, I had been porting a lisp interpreter called XLISP, which was written in C to Pascal. Ever since I was interested in parenthesized languages, and when I learned about Scheme, I was immediately fascinated by its clear and beautiful design. So I started writing small Scheme scripts and finally became more interested and decided to build an interpreter. It started out small, as always, but soon I noticed that I was able to implement even a bit more complicated features, so the interpreter grew. Now the distribution is more than half a meg big, and still grows.

1.2 Sizzle Future

I hope that I will be able to make a more accurate and complete documentation someday. Also, there are a few features I would like to see in the interpreter, such as bignum arithmetic, rationals, hygienic macros, etc. Some internal architectural changes seem desirable also, for example for speeding up the reader. Some profiling for creating a faster evaluator will be on the wishlist also, maybe more extensive syntax rewriting may help with this.

I would like to see all (at least all sensible) SRFIs implemented. Currently, only SRFI-1, SRFI-6 and SRFI-8 are implemented. SRFI-13 is implemented partly.

2 Using Sizzle

Sizzle is an embedding language which can be easily integrated into other programs written in C, but it also includes a command line program which can be used to do calculations in a read-eval-print loop as well as running scripts written in the Scheme language.

This chapter will explain how to make use of Sizzle as an interactive program as well as how to execute script files.

2.1 Starting Sizzle

To start Sizzle, simply type ‘sizzle’ at the command line. Sizzle will then initialize and start reading expression from the terminal. It will read and evaluate expressions and print them to the screen until you terminate it by either pressing ‘^D’ at the prompt or entering the expression ‘(exit)’.

The command line interpreter understands the command line options shown in the table below.

- `-s, --script=file`
Execute the Scheme source file ‘file’, and exit.
- `-c, --command=expr`
Evaluate the Scheme expression *expr*, and exit.
- `--`
Stop processing options and treat all following arguments as non-options, even if they begin with a dash (-).
- `-l, --load=file`
Load Scheme source code from ‘file’.
- `-e, --eval=function`
After reading script given with `-s, --script=file`, apply *function* to command line arguments. This option also can be used with `-c, --command=expr`.
- `-h, --help`
Show a help message about program usage and terminate.
- `-v, --version`
Show the version of the program and terminate.
- `-q, --no-init-file`
Do not load the user’s init file ‘~/sizzle’.
- `-n, --no-system-init-file`
Do not load the system init file.
- `-d, --debug`
Increase the debugging level. This option can be given more than once, each occurrence will turn on more debugging messages.

The command line option `-s, --script=file` has to be used whenever a Scheme script is to be executed by Sizzle. The file ‘file’ will be read and all Scheme expression contained in it will be evaluated in order. After evaluation is finished, Sizzle terminates. For more details about Sizzle scripts see Section 2.5 [Sizzle Scripts], page 5.

`-c, --command=expr` is similar, but instead of reading a file of Scheme expressions, the Scheme expression *expr* will be evaluated.

Note that when using `-s` or `-c`, all remaining arguments on the command line are ignored by Sizzle and are passed to the Scheme script or the Scheme expression without interpretation and are available via the primitive `command-line`, please see Section 5.16 [System interface functions], page 55.

The pseudo-option `--` will be discarded from the command line, but all remaining parameters will be treated as normal arguments and not as options, even if they begin with a dash (`-`). This can be used when passing options to a Sizzle script.

All file names given with the option `-l`, `--load=file` (which can be given more than once) are loaded into the interpreter by evaluating all expressions read from the files, but unlike `-s`, the interpreter will not terminate after evaluating the files, but continue as normal. That means, that without `-s` and `-c`, the interactive read-eval-print loop will start, otherwise the appropriate action will be done, as described above.

`-e`, `--eval=function` is intended to be used with `-s`, when you are using Scheme files which can be used both as library files and as executable scripts. You can do this by placing library code in the file and a function which serves as the main function. When you use `-s`, the file will simply be loaded, but the main function will normally not get called. This is exactly what `-e` is for. Simply pass the name of the main function to `-e` and this function will then be applied to the remaining command line arguments after the script file has been loaded.

`-h`, `--help` will print out a short helping message and then quit.

`-v`, `--version` prints out the version of the Sizzle interpreter you have installed, and then terminate. Use this command if you want to complain about a bug in your version of Sizzle, in addition to as much information about your system as possible.

Use the option `-q`, `--no-init-file` if you want to inhibit automatic loading of your user's init file `'~/.sizzle'`.

Similarly the option `-n`, `--no-system-init-file` prevents the system's init file from being loaded. Note that with this option, some common procedures will not be available in the interpreter.

With the option `-d`, `--debug`, Sizzle will print some debugging messages while running. Each occurrence of `-d` increases the debugging level and will cause more information to be printed out. This option will not be very interesting for normal users and is intended for being used by the Sizzle developers.

2.2 Interactive Usage

When you have started the Sizzle interpreter, you can start right away with typing in Scheme expressions and see how they evaluate. I will give some simple examples in this section.

Numbers, characters and strings can be entered in their read syntax and evaluate to themselves.

```
zzz: 1
1
zzz: "Hello World"
"Hello World"
zzz: #\H
#\H
```

Lists and vectors need to be quoted, because they do not evaluate to themselves. Lists are considered to be function calls and therefore evaluate in a special way (see below). The reason why vector constants do not evaluate is not clear to me, but R5RS requires that. Strange...

```
zzz: #(1 2)
exception: misc-error: vectors do not evaluate to themselves
no backtrace available
evaluate '(set! %save-backtrace% #t)' if you want backtraces
zzz: '(1 2)
#(1 2)
zzz: (1 2)
exception: misc-error: not a procedure: 1
zzz: '(1 2)
```

```
(1 2)
```

In the above two examples, error messages have been printed in response to the user's input. In an error message, the type of the error is printed first: Here, it is an exception of type `misc-error`. The following is a description of the error. In addition, we can see that the first error shown to the user includes some more information: The fact that currently backtraces are not available is noted and a tip how to enable procedure backtraces.

Procedures are invoked by writing the function name as the first member in a list. Consider the function `+`, which adds all its arguments.

```
zzz: (+ 1 2)
3
zzz: (+ 1 2 -4)
-4
```

2.3 Automatic Interactive Variable

When running the standalone interpreter, Sizzle defines automatically a special variable to ease interactive usage. The result of an evaluation is automatically assigned to the top-level variable `$$`, which can be used in following expressions. Note that the value of this variable is overwritten after each successful operation, so you have to save it yourself if you want to use it later.

```
zzz: 1
1
zzz: $$
1
zzz: "foobar"
"foobar"
zzz: (string-length $$)
6
```

2.4 Command Line Editing

Note: Readline support has been removed from the Sizzle core, so you have to install the additional package `'sizzleopt'` to make use of it.

When available, Sizzle uses the `'readline'` library to provide command line editing to the command line interpreter.

The `'readline'` library enables command line editing with the cursor keys and some key sequences known from Emacs. Also, a command line history is available, you can recall previously typed in lines with the cursor-up key. Tab-completion, known from the shell is supported, too. When you type the beginning of a word and then hit the `(TAB)` key, the word will be completed; when there are several possible matching words, readline will complete as much as possible and then beep. You can then press the `(TAB)` once more to get a list of the matches.

Sizzle is a little bit context sensitive with tab-completion. When the word you are completing comes directly (or only separated by whitespace) after an opening parentheses, only procedure names are completed, in all other positions, variable names are also completed.

The command line history which is maintained by the `'readline'` can be written to the file `'sizzle_history'` when Sizzle exits and read in when it starts again. By default, this feature is disabled to avoid filling the home directory of users who do not wish their programs to do so. You have to set the boolean variable `%write-history-file%` to `#t` to enable command line history saving. The best place to do so is in the user initialization file `'sizzle'` in the user's home directory.

2.5 Sizzle Scripts

Sizzle is not only suitable for interactive usage, but also for programming Scheme scripts. These scripts can manipulate text, can do numeric calculations or can even be CGI scripts. Sizzle scripts can include code from the Sizzle library using the `load` procedure.

The Sizzle program runs in non-interactive mode if you invoke it with the `-s` option, like `sizzle -s kewlt-scheme-program.scm`.

You can run Sizzle scripts without having to type the name of the Sizzle executable if you include the following two lines at the top of your script and making the script executable using the `chmod` program:

```
#!/usr/local/bin/sizzle -s
!#
```

Another possibility is to make the following lines the first ones in your script. This version has the advantage that it is independent on the location of your Sizzle executable, but it is a little bit less efficient because the operating system has to start a shell, which in turn starts the Sizzle interpreter. Normally, the performance difference will not be noticeable, though.

```
#!/bin/sh
exec sizzle -s $0
!#
```

Scheme scripts can examine the arguments they were given by calling the procedure `command-line`. It returns a list of strings, each being one command line argument. The first member of the list is the name of the interpreter, or (if you have a more sophisticated Unix flavour and are using the `#!` notation for executing the interpreter) the name of the script.

2.6 Startup Sequence

On startup, the file `init.scm` which was installed together with the binaries and libraries is loaded before evaluating expressions from source files or the command line, and before entering the read-eval-print loop. `init.scm` is stored in the directory `$prefix/share/sizzle/$version/`, where `$prefix` is set during configuration time (defaults to `/usr/local`) and `$version` is the version of your installed Sizzle package.

In addition, the system init file loads the file `.sizzle`, if it exists, from the user's home directory. So if you like to customize Sizzle's behaviour, this file is the right one to put Scheme expressions.

Note that these startup files are loaded whenever the interpreter starts up, no matter whether it runs interactively in the read-eval-print loop or was started to run a Scheme script.

Loading of the initialization files can be suppressed by using the command line options `-q`, `--no-init-file` and `-n`, `--no-system-init-file`. The former inhibits the loading of the user's initialization file `~/sizzle`, whereas the latter suppresses loading of the system-wide initialization file.

2.7 Environment Variables

When Sizzle starts up, it examines the contents of the environment variable `SIZZLE_LOAD_PATH`. If it is set, it must contain a colon-separated list of directory names. Each of these path names is then prepended to the global variable `%load-path`, in the order they appear in the environment variable. This environment variable makes it possible to modify the locations where Sizzle searches for files which are loaded via `load` or `primitive-load-path`, or modules which are included with `use-modules`.

2.8 Building Sizzle

Some notes on building the Sizzle library and interpreter.

First, you have to get a Sizzle archive. You should be able to get a recent version from <http://www.pintus.de/mgrabmue/sizzle/sizzle.html>. Just follow the *Download* link and download an archive like 'sizzle-0.0.30.tar.gz'. Replace the version number with the newest available version. You may also want to download the *sizzleopt* package, which includes some useful bindings, such as 'readline' support for the interpreter.

Unpack the distribution tarball to a directory of your choice:

```
$ cd src
$ tar xzvf sizzle-0.0.30.tar.gz
sizzle-0.0.30/
sizzle-0.0.30/Makefile.in
...
```

Change to the newly created directory.

```
$ cd sizzle-0.0.30
```

When compiling for Windoze using Micro\$oft Visual C, you can use the provided Project files. Refer to the 'README-Win32' file for further information.

Now you have configure the package. Most of the times, this just requires you to type

```
./configure
```

but you may want to change the configuration to suit your needs. Invoke 'configure' with the `--help` option to get a list of the supported options.

Currently, the following configuration options besides the standard options are supported:

`--enable-warnings`

Switches on most compiler warnings. This may be useful for the maintainer only or people who want to hack on the package. This is off by default.

`--enable-debug`

Switch on debugging code. This is on by default.

`--enable-posix`

Compile in Posix functions such as `getpid`, `fork` etc. This is on by default.

`--enable-regex`

Compile in support for Posix regular expressions. This is on by default.

`--enable-maximum-functionality`

This option controls inclusion of some code which may not be needed for all users, such as networking support.

`--enable-dl`

Compile in dynamic linking code. This will enable the library to load extension modules without recompilation. This is on by default.

`--enable-uvector`

Enable support for uniform vectors. They are currently not compiled in by default because I do not use them now, and they bloat up the library unnecessarily.

`--with-mingw=DIR`

When compiling for Windoze, you can give the location of your MingW32 installation here.

`--enable-shared`

Enable the compilation of shared libraries. On by default.

--enable-static

Enable the compilation of static libraries. On by default.

--disable-nls

Switch off national language support. You may want to use this if your NLS is broken or you simply don't like localized messages.

Every **--enable** option has a **--disable** pendent to switch off a particular feature.

After a while, the configuration is done and you can compile the package.

```
$ make
```

When compiled, the package is ready for installation. Note that only the super-user may do this if you are installing to the default location `'/usr/local'`.

```
$ su
Password:
# make install
# exit
$
```

When everything worked well, you can now use the Sizzle interpreter.

```
$ sizzle
zzz: (+ 1 2)
3
zzz:
```

3 Programming Scheme

This chapter gives a quick introduction to programming in the Scheme language. It will also mention Sizzle limitations and features where appropriate.

3.1 General

Scheme uses fully parenthesized prefix notation, that means that all procedure calls are enclosed in parentheses. The first element in a procedure call is the procedure which will be invoked and the other elements are the arguments which will be passed to the procedure. Note that the procedure will be evaluated in the same way as the rest parameters.

```
zzz: (+ 1 2)
3
zzz: ((lambda (x) x) 1)
1
```

All arguments in Scheme are passed by value, e.g. evaluated before the procedure is called. Special forms like `if`, `define` etc. do not follow that rule, they evaluate their arguments only when needed. There is no syntactic difference between a procedure call and the application of a special form, so you need to know the names of all special forms in order to tell how some given code will evaluate.

Variables are defined using the special form `define` and can be modified later on by using `set!`, which is another special form.

```
zzz: (define a 1)
zzz: a
1
zzz: (set! a 2)
zzz: a
2
```

Functions are either defined by defining a variable with the value of a lambda expression or using a special form of `define`, which is intended for defining procedures more easily.

```
zzz: (define identity (lambda (x) x))
zzz: (identity 2)
2
zzz: (exit)
$ sizzle
zzz: (define (identity x) x)
zzz: (identity 3)
3
```

When you define a variable more than once in an environment, the former definition will be silently overwritten.

3.2 Storage Model

The Sizzle core contains a garbage collector, so that the programmer is not responsible to free objects which are not needed anymore. Internally, the garbage collector is implemented as a conservative collector which marks all objects as being used which are reachable through global variables (known to the collector) or are reachable by scanning the C stack.

At the moment, the garbage collector is a very simple, recursive mark-and-sweep collector, without generations or incremental behaviour. It has been sufficient for my needs, but maybe I will switch to a more sophisticated algorithm later when a need for it appears.

3.3 Data Types

Sizzle has a lot of useful data types already built in, and provides a lot of procedures for manipulating values of various types. For detailed description of the available data manipulation functions please refer to the Programming Reference chapter in this manual.

3.3.1 Numbers

The Scheme standard defines the so-called *numeric tower*, a hierarchy of numeric data types where every data type includes the values of the data types lower in the hierarchy and which is defined as follows:

number	All numbers belong to the data type <i>number</i> .
complex	<i>Complex</i> numbers consist of a real and an imaginary part.
real	<i>Real</i> numbers are all rational and irrational numbers.
rational	<i>Rational</i> numbers consist of a numerator and a denominator.
integer	<i>Integers</i> are all numbers without a fraction component.

Furthermore, Scheme makes a difference between *exact* and *inexact* numbers. *Exact* numbers are always precise whereas *inexact* numbers are only an approximation. The same differentiation is made for operations on numbers. For example, the addition of two exact numbers always produces an exact result, but on the contrary, the division of exact numbers may be inexact.

The read syntax for numbers is as follows: the number may start with an optional number of exactness and radix, followed by a number of digits which must fit into the radix scheme which may be given with a radix prefix. Then a decimal point and another sequence of numbers may follow, in decimal or scientific notation may follow. A list of the valid exactness and radix prefixes is given in the following table.

#e	Number is exact
#i	Number is inexact
#b	Binary number, radix 2
#o	Octal number, radix 8
#d	Decimal number, radix 10
#x	Hexadecimal number, radix 16

Integers are exact numbers with the same range as the C `long` data type.

Examples of integer constants:

```
42
-23
#xdead
#o755
#b100001010
#d42
```

Floats are floating point values, corresponding to `double` values in C. Real number constants can start with a radix and exactness prefix, but only the radix prefix `#d` is allowed. When using the exactness prefix `#e` (exact number), the number must have a fraction of zero. Read syntax examples:

```
3.1459
-10.1e2
#i1.0
```

`#e1.0`

Complex and rational numbers are not supported by Sizzle. Maybe in the future they will. Also *bignums* (arbitrary precision numbers), as known from other Scheme systems, are not (yet) supported.

3.3.2 Booleans

Booleans are the truth values true and false, in Scheme written like this: `#t` means true and `#f` means false.

In Scheme, `#f` is the only value that is considered false in conditionals, neither 0 nor the empty list `'()` counts as false.

3.3.3 Characters

Characters are denoted in the following form: first, a hash mark (`#`), then a backslash and then the literal character. Examples:

`#\j` The character 'j'
`#\` The space character

Scheme provides another notation for non-printable characters. A space character can also be written like this `#\space` and a newline like `#\newline`. Sizzle understands three additional symbolic names for characters. This table shows all of them:

`#\space` The space character
`#\newline` The newline character
`#\return` The carriage return character
`#\tab` The tab character.
`#\bell` The bell character.

3.3.4 Strings

Strings are sequences of characters. String constants are enclosed in double quotes and can contain all characters from the ASCII set. The double quote character `"` and the backslash `\` have to be quoted with a preceding backslash. Some special non-printable characters can be included in string constants by denoting special backslash-character sequences. The following sequences are defined:

`\n` Newline
`\r` Carriage return
`\t` Tab character
`\v` Vertical tab
`\b` Backspace
`\a` Bell character

Strings can be mutable and immutable. Immutable are those which were literally entered from the input source. They can not be used with destructive procedures like `string-set!` or `string-fill!`. Strings returned by functions like `string-copy` or `string` are mutable and can be used with such strings.

3.3.5 Symbols

Symbols are values with some special properties. They are written by simply writing their name, without any quotes or special markup characters. The main difference between symbols and strings is that symbols which have the same textual representation **are** in fact the same object. Additionally, symbols can have values associated to them, then they act as variables.

When reading symbols, the Sizzle reader considers all characters which do not explicitly end the symbol as part of the symbol. These ending characters are ‘(’, ‘)’, *whitespace*, ‘;’ and ‘”’. It is even possible to include these characters into a symbol name by preceding it with a backslash. Thus, these all are valid symbol names:

```
Hello
i-am-a-symbol
strange\ symbol
```

Of course, it is not too clever to use symbols with enclosed whitespace or parentheses in program source code. Their use can degrade readability of programs considerably.

3.3.6 Keywords

Keywords are similar to symbols in that they are represented externally by a character sequence and that keywords with the same external representation are indeed the same objects internally. The difference between keywords and symbols is that symbols are normally bound to locations containing values and that these values are retrieved when a symbols is evaluated, whereas keywords evaluate to themselves. Thus it is not necessary to quote keywords on input.

Keywords are written like symbols, but are either prefixed with a colon (:) or the sequence hashmark-colon (#:). When comparing, only the keyword name is significant, the prefix is ignored.

```
zzz: #:keyword
#:keyword
zzz: :keyword
#:keyword
zzz: keyword
error: unbound variable: keyword
```

In the above example you can see how keywords evaluate to themselves and that evaluating an unknown symbol signals an error. Note also that keywords are always printed in the #:-notation, regardless how they were entered.

3.3.7 Cons Cells

Cons cells are cells which can hold two other values. The first of the value is called the ‘**car**’ of the cell and the second is called ‘**cdr**’ (for historic reasons). In source files and on the read-eval-print prompt you can create cons cell by using the *dotted pair* syntax.

```
'(foo . bar)
'(1 . 2)
```

Another way to create cons cells is using the constructor function **cons**:

```
zzz: (cons 1 2)
(1 . 2)
```

Lists, the main data structure in Sizzle are also made out of cons cells, but they can be typed in a more convenient way than

```
(1 . (2 . (3 . '())))
```

by simply writing

```
(1 2 3)
```

Both result in the same value.

3.3.8 Vectors

Vectors are similar to arrays in other programming languages. You can store a fixed count of values in a vector and can access them in constant time, as opposed to the time for accessing elements of a list, which is linear to the list length. The difference to ordinary arrays is that the elements of a vector can have different types.

Vectors are written like lists, but before the opening parentheses you have to put a hash mark.

```
#(1 2 3 4 5 6 7)
#("Hello" #\ "World" \#!)
#(#(1 2) #())
```

The last example shows that vectors may contain vectors and that vectors may be empty.

Vectors can be mutable and immutable. Immutable are those which were literally entered from the input source. They can not be used with destructive procedures like `vector-set!` or `vector-fill`. Vectors returned by functions like `make-vector` or `vector` are mutable and can be used with such strings.

3.3.9 Homogenous numeric vectors

Note: Currently, homogenous numeric vectors are only available if support for them has been explicitly enabled on configuration time for the package. See Section 2.8 [Building Sizzle], page 6 for how to enable the support if your installed version lacks it and you need it.

Normal Scheme vectors are heterogenous datatypes. You can store any value in them, just as you can with lists. Sizzle provides another kind of vectors, so-called homogenous numeric datatypes, as defined in SRFI-4. These vectors are special, because it is only possible to store numbers of a particular data type in there. There are 10 different types of those vectors, each for another numeric data type: signed 8-bit values, unsigned 8-bit values, signed 16-bit values, unsigned 16-bit values, signed 32-bit values, unsigned 32-bit values, signed 64-bit values, unsigned 64-bit values, 32-bit floating point values and 64-bit floating point values. Only values of their type can be stored into these vectors, so they provide additional type safety. They are also more space-efficient, because, for example, vectors of signed 8-bit quantities need only to reserve 8 bits for each elements, whereas normal vectors reserve at least 32 bit for each element, up to 160 bits.

Each of the data types has its own type prefix, which is used in all procedures dealing with that type. The following table defines the prefixes.

s8	signed 8-bit values
u8	unsigned 8-bit values
s16	signed 16-bit values
u16	unsigned 16-bit values
s32	signed 32-bit values
u32	unsigned 32-bit values
s64	signed 64-bit values
u64	unsigned 64-bit values
f32	32-bit floating point values
f64	64-bit floating point values

Homogenous numeric vectors have a read (and print) syntax similar to vectors, but between the hash mark and the opening parentheses the prefix for the vector type must be inserted:

```
#u8(1 2 3 4 5 6 7)
#s16(-42 23 2 0)
#f32(1.2 3.14159265)
```

Similar to normal vectors, literally denoted homogenous vectors are read-only and can not be modified using primitive procedures like `u8vector-set!`. Use procedures like `make-f64vector` to create mutable vectors.

3.3.10 Hash tables

Hash tables (aka associative arrays) are very useful data structures, when a non-trivial programming task is at hand. Hash tables provide a mapping from one object to another and are similar to *association lists*, but they are significantly faster when they grow large.

In Sizzle, hash tables can be written literally with a `#{` prefix, followed by key/value pairs separated by the arrow symbol (`=>`), like this, terminated by a closing brace (`}`):

```
zzz: #{Joe => "cool" Jim => "lame"}
      #{Jim => "lame" Joe => "cool"}
```

This is also the print syntax for hash tables.

Another possibility is to use the procedure `make-hash` which constructs a hash table from its arguments. All arguments must be pairs which will be taken as key/value pairs and inserted in the new hash table.

```
zzz: (make-hash '(Joe . "cool") '(Jim . "lame"))
      #{Jim => "lame" Joe => "cool"}
```

Note that the order of the key/value pairs depends on the size of the data structure internally used to store the values and may differ for the same input, when this size is not the same. The size is either chosen internally, depending on the number of pairs, or can be provided as an optional first argument to `make-hash`:

```
zzz: (make-hash '(Joe . "cool") '(Jim . "lame"))
      #{Jim => "lame" Joe => "cool"}
zzz: (make-hash 32 '(Joe . "cool") '(Jim . "lame"))
      #{Joe => "cool" Jim => "lame"}
```

The best choice for the size argument is a prime number not too near to a power of 2 (so says Knuth).

3.3.11 Procedures

There are three kinds of procedures in Sizzle: Builtin procedures, builtin syntactic forms and user-defined procedures (lambda expressions, or closures).

The difference between builtin functions and syntactic forms is that functions evaluate their arguments before they are called and arguments to syntactic forms are only evaluated when needed.

Builtin functions and syntactic forms do not have any read syntax (that would not make sense) and they print in the following way:

```
zzz: set!
#<macro set!>
zzz: append
#<primitive-procedure append>
```

Closures print as a pair of the lambda expression they stand for and the address of the environment the expression is closed over.

```
zzz: (define (f n) n)
zzz: f
#<closure #<procedure f (n)> 0x401af072>
```

3.3.12 Regular Expressions

Regular expressions are created using the primitive procedure `make-regex`. The resulting regular expression object can then be used in calls to `regex-exec` in order to match the regular expression against strings. The format of the regular expression passed to `make-regex` is the same as for the Posix regular expressions. For more information, type

```
man regex
```

Regular expressions are not part of any Scheme standard, but the procedures implemented in Sizzle are compatible to those used by Guile.

```
zzz: (define rx (make-regex "foo|bar"))
zzz: (regex-exec rx "a-foo-baz")
#("a-foo" (2 . 5))
zzz: (regex-exec rx "bar-o-matic")
#("bar" (0 . 3))
```

Due to limitations in the interface of Posix regular expressions, regular expressions in Sizzle may not contain null bytes. Pattern strings are truncated at the first null byte on compilation.

3.3.13 Multiple Values

In Scheme, it is possible to return more than one value from a procedure. This is done by using the primitive procedure `values` as the last expression in your procedure. Handling those values when returned from a procedure is a bit strange, since the only standard form defined for that purpose is `call-with-values`. The following examples may (hopefully) clear up the whole mess.

```
zzz: (values 1 2 3)
1
2
3
zzz: (call-with-values (lambda () (values 1 2)) (lambda (x y) (+ x y)))
3
```

When multiple values are returned to the read-eval-print loop, they are printed one after the other, every value on a new line. The second example shows the usage of `call-with-values`, where the first lambda expression is evaluated (and returns multiple values) and the values returned from the first procedure are passed to the second procedure as normal arguments.

3.3.14 Special values

Some values which appear in the interpreter are different from all others and cannot be assigned to one of the data types. These values are referred to as *Special Values* in this manual. Examples of these special values are the `end-of-file` object which indicates an end-of-file condition, or the `unspecified` object used inside of the interpreter.

3.3.15 Constants

Sizzle supports a weird data type for constants. They can represent any Scheme object, but have the additional property that references to constants are replaced by their value when they are evaluated in a procedure body. The result is faster execution, because the necessary lookup step for variables is eliminated. Constants are created with the syntactic form `define-constant` and can be used by variable, once they are defined.

Notice that due to the substitution feature, it is not possible to redefine a constant. But would it be a constant then?

3.3.16 Ports

All input and output in Scheme is done using *ports*, which can be regarded as data sources and sinks. Ports are characterized by the fact that they provide procedures for reading and writing data, telling and setting file pointer positions (if they are built on top of files) and various others. Ports are either created explicitly by calls to procedures like `open-input-file`, or implicitly when using `with-output-to-string` and friends. They have a print syntax, but no read syntax.

```
zzz: (current-input-port)
#<rlport:open=1:read=1:write=0>
zzz: (current-output-port)
#<fdport:open=1:read=0:write=1:fd=1>
```

The first port in this example is a readline port, which supports command line editing, parentheses matching etc., the second is a normal file descriptor port connected to the standard output file descriptor.

Note that readline ports are only available if you have installed the ‘`sizzleopt`’ package and used the readline module with (`use-modules (readline readlie)`).

At the moment, Sizzle supports so-called *fdports*, which use an underlying file descriptor, *fports* which use an underlying ‘`stdio`’ file structure, string ports which use an internal character buffer for input and output, and *rlports* (readline ports) which read from the terminal via the ‘`readline`’ library. The latter provide fancy command line editing.

3.3.17 Errors and Exceptions

Errors and Exceptions are distinct data types in Sizzle. Whenever an error occurs (such as when you pass the wrong number of arguments to a procedure), an error object is created and then propagated to the top level, which may be either the read-eval-print loop or a procedure evaluating a file. There the error is displayed, using the information in the error object. In an error object, an error message is stored, and if the error comes from the reader module, a file name and a line and column number is also included. Error objects can not explicitly created, but they are created whenever you use the `error` procedure to signal an error.

Exceptions are a bit different. When raised, they also create objects containing an error message, but they also contain a so-called *exception tag*, which may be any atom. Exceptions can be raised from the Scheme code by calling the procedure `throw`. Using the exception tag for a particular exception or the catch-all exception tag `#t`, the user can catch exceptions with the procedure `catch`.

The rule of thumb is that exceptions are raised when an error situation allows continued evaluation, and errors are signalled on errors which are too serious to continue. An examples for exceptions is the `file-not-found` exception, which is raised when a non-existent file is opened; a quite common situation. Errors are signalled on syntax errors, invalid procedure application or typing errors.

Right now, the difference between exceptions and errors in the Sizzle core is a bit half-baked. Not all continuable error situations throw exceptions, a lot of them unnecessarily signal errors. This will be changed in the future. Currently, support for errors is slowly phased out and all error-signalling places in the code are changed to throw various types of exceptions instead.

3.3.18 Continuations

Sizzle only supports continuations usable as escape procedures. That means that one cannot export a continuation from the defining context by returning it from a procedure or storing it into a global variable, and then try to invoke it.

3.3.19 Boxes

Boxes are a data type which are used to store other Scheme objects. A box simply contains another object and can be used to pass objects to procedures in a style similar to call-by-reference calling semantics.

Boxes are created with a call to the procedure `make-box`, which takes the object to store into the box. The contained object can be retrieved from a box with `box-ref` and set with `box-set!`. The predicate `box?` is used to check whether a given object is actually a box. For more details on these procedures, see Section 5.23 [Box Operations], page 71.

3.3.20 Pointers

Sizzle provides a special data type called *pointer*. Objects of this type wrap so-called bounded pointers, that means that they contain a pointer into the machine's address space together with the size of the object pointed to. These objects are used when interfacing to C library functions which receive character buffers. Pointer objects created with the primitive procedure `make-pointer` allocate memory when created and release the allocated memory when the pointer object gets unreachable. For more information about pointer procedures, refer to Section 5.24 [Pointer Operations], page 71.

4 Predefined Variables

In the Sizzle core, some variables are defined which can be used from within Sizzle scripts to control and modify Sizzle's behaviour.

Variables labelled as *Library variables* are defined in the startup file 'init.scm' and thus are only present if the embedding application has loaded that file on startup.

%load-path Variable

A list of directory names to search when loading Scheme source files. Initially, only the directories where the Sizzle startup file is located and the directory for site-specific Scheme files is in the list. The user can add more directories if needed. Currently, only the `load` library procedure from the system init file uses this variable.

%sizzle-version% Variable

This is a string containing version information of the running interpreter. It has the format `major.minor.patchlevel`, and looks like this: `0.0.30`.

%sizzle-major-version% Library variable

%sizzle-minor-version% Library variable

%sizzle-patchlevel% Library variable

These variables hold the components of `%sizzle-version%`, split up at the dots and converted to integers.

These variables are not defined when the option `-n`, `--no-system-init-file` is passed to the interpreter, or if the system startup file is not found.

%write-history-file% Variable

Boolean variable which controls whether the readline history will be written to the file `~/.sizzle_history` when the interpreter terminates. By default this is false. A good place to set this variable is in your `~/.sizzle` personal initialization file.

(Only available if the optional module (`readline readline`) was used).

gc-message Variable

This is a boolean variable. If set to `#t`, a message will be printed whenever the garbage collection is invoked or the cell heap grows.

%max-evaluate-stack% Variable

Integer variable which holds the maximum evaluation stack size which is allowed. When the current stack size exceeds this value, an error is signalled. This variable can be set by user code to allow deeper recursion, but be careful: when setting this variable to very high values and performing heavy recursive evaluations Sizzle sometimes dumps core, because it runs out of stack space.

%print-read% Variable

Boolean variable which controls whether expressions read in are printed before evaluating. Mostly for debugging purposes.

%print-result% Variable

Boolean variable which controls whether expressions are printed after evaluation even when loading script files. Mostly for debugging purposes.

%print-memoized% Variable

Boolean variable which controls the way memoized expressions are printed. If true, memoized global and local variables are printed in a more verbose way; but then the output of (`backtrace`) is much less readable, so this is false by default.

- %print-func-bodies%** Variable
Boolean variable which controls whether user-define procedures are printed with their procedure bodies or not. If set to **#f**, only the procedure name is printed and the formal parameter list.
- %save-backtrace%** Variable
Boolean variable which controls whether a procedure call history is kept which can be displayed for debugging purposes when errors occur. If you set this to **#t**, the interpreter will consume a lot of memory if your program is heavily recursive.
- %print-backtrace%** Variable
Boolean variable which controls whether procedure call backtraces are printed immediately after an error or an exception occurs. If set to **#t**, backtraces are only printed when the primitive `backtrace` is called.
- %print-backtrace-limit%** Variable
Integer variable which controls how many lines of backtrace are printed when an error occurs. You have to set both `eval-save-backtrace` and `eval-print-backtrace` to **#t** if you want backtraces.
- %eval-time-taken%** Variable
This integer variable holds the time in ticks which the last top-level evaluation took.

5 Builtin Procedures

Sizzle comes with a large number of builtin functions and special forms. The difference between these two is that the arguments to a function are evaluated before the function is called, the arguments to a special form are only evaluated when needed. This makes a difference for forms like `set!`, because the first argument (a symbol) needs not be to be quoted like it would have to if `set!` was a function. Also it is necessary for forms which evaluate their arguments only when a certain condition holds. `if`, for example, only evaluates its third argument if the first evaluates to the value `#t`.

The provided procedures are grouped into two classes: primitive and library. Procedures which are labelled library are only available if the library files, which are installed together with Sizzle, are loaded on initialization. This will be the case if you installed Sizzle properly.

5.1 Equality tests

The following three procedures provide tests for three different degrees of equality. They are always available in the interpreter.

eq? *obj1 obj2* Primitive procedure
Returns `#t` if *obj1* and *obj2* are the same values, that means if both are stored at the same location in memory. Symbols with the same textual representation pass this test, strings don't. For numbers, the result is not predictable, `(eq? 1 1)` **can** return `#t`, but not necessarily. Use `eqv?` or `=` to test numbers for equality. Strings can only compared using the predicate `equal?` or one of the string comparison predicates.

eqv? *obj1 obj2* Primitive procedure
This is the same as `eq?` but differs as far as numbers are concerned. That means that `(eqv? 1.0 1.0)` return `#t` in any case.

equal? *obj1 obj2* Primitive procedure
Returns `#t` if the structure of *obj1* and *obj2* is the same. You can use `equal?` to test lists, vectors and strings for equality, these data structures are tested recursively. If you want to compare strings, use this procedure, or use the procedure `string=?`.

5.2 Numerical operations

The numerical operations documented in this section are available on all platforms without needing additional modules.

number? *obj* Primitive procedure
Returns `#t` if *obj* is a number, `#f` otherwise.

complex? *obj* Primitive procedure
Returns `#t` if *obj* is a complex number, `#f` otherwise.

real? *obj* Primitive procedure
Returns `#t` if *obj* is a inexact number in floating point representation, `#f` otherwise.

rational? *obj* Primitive procedure
Returns `#t` if *obj* is a rational number, `#f` otherwise.

integer? *obj* Primitive procedure
Returns `#t` if *obj* is an exact integer number, `#f` otherwise.

long? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an exact long integer number (a non-immediate integer, not a bignum), #f otherwise.	
exact? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an exact number, #f otherwise.	
inexact? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an inexact number, #f otherwise.	
= <i>z1 z2 z3 ...</i>	Primitive procedure
Returns #t if all arguments are numerically equal, #f otherwise.	
!= <i>z1 z2 z3 ...</i>	Primitive procedure
Returns #t if all arguments after the first are not numerically equal to the first argument, #f otherwise.	
< <i>z1 z2 z3 ...</i>	Primitive procedure
Returns #t if all arguments are strictly ordered in increasing order.	
<= <i>z1 z2 z3 ...</i>	Primitive procedure
Returns #t if all arguments are ordered in increasing order.	
> <i>z1 z2 z3 ...</i>	Primitive procedure
Returns #t if all arguments are strictly ordered in decreasing order.	
>= <i>z1 z2 z3 ...</i>	Primitive procedure
Returns #t if all arguments are ordered in decreasing order.	
zero? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is equal to zero, #f otherwise.	
positive? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is greater than zero, #f otherwise.	
negative? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is less than zero, #f otherwise.	
odd? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an odd number, #f otherwise.	
even? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an even number, #f otherwise.	
max <i>x1 x2 ...</i>	Primitive procedure
Returns the maximum of its arguments.	
min <i>x1 x2 ...</i>	Primitive procedure
Returns the minimum of its arguments.	
+ <i>z1 ...</i>	Primitive procedure
Adds all numbers in the argument list and returns the sum. If called without parameters, 0 is returned.	
* <i>z1 ...</i>	Primitive procedure
Multiplies all argument values and returns the product. Returns 1 if called without parameters.	

- $z1 \dots$ Primitive procedure
Takes the first argument and subtracts all remaining argument values one after the other. At least one parameter is required, if called with exactly one parameter, the negated value of the parameter is returned.
- / $z1 \dots$ Primitive procedure
Divides the first argument by all remaining arguments in turn. If called with one parameter, the reciprocal of the argument is returned. Should any of the remaining arguments be zero, a division-by-zero error is signalled.
- abs** x Primitive procedure
Returns the absolute value of its argument.
- magnitude** z Primitive procedure
Returns the magnitude for x . This is the same as **abs**, since Sizzle does not support complex numbers.
- angle** z Primitive procedure
Returns the angle of x in polar representation. This is $-\pi$ for negative numbers and 0 for positive numbers since Sizzle does not support complex numbers.
- real-part** z Primitive procedure
Returns the real part of the number z . In Sizzle, this returns z itself, because complex numbers are not supported.
- imag-part** z Primitive procedure
Returns the imaginary part of the number z . The Sizzle implementation simply returns 0, because complex numbers are not supported.
- numerator** q Primitive procedure
Returns the numerator of its argument q . Currently, q itself is returned because rational numbers are not supported.
- denominator** q Primitive procedure
Returns the denominator of its argument q . Currently, 1 is returned because rational numbers are not supported.
- quotient** $n1 n2$ Primitive procedure
Performs an integer division operation on the two arguments. An arithmetic overflow error is signalled if the second operand is equal to zero.
- remainder** $n1 n2$ Primitive procedure
Returns the rest of an integer division of its arguments. An arithmetic overflow error is signalled if the second operand is equal to zero.
- modulo** $n1 n2$ Primitive procedure
Returns the rest of an integer division of its arguments. An arithmetic overflow error is signalled if the second operand is equal to zero. This differs from **remainder** as far as negative operands are concerned.
- floor** x Primitive procedure
Returns the largest integer not larger than x .
- ceiling** x Primitive procedure
Returns the smallest integer not smaller than x .
- truncate** x Primitive procedure
Returns the closest integer to x whose absolute value is not larger than the absolute value of x .

round x	Primitive procedure
Returns the closest integer to x rounding it to even when x is halfway between two integers.	
exp z	Primitive procedure
Implements the exponential function.	
log z	Primitive procedure
Returns the natural logarithm of z .	
sin z	Primitive procedure
Returns the sine of z .	
cos z	Primitive procedure
Returns the cosine of z .	
tan z	Primitive procedure
Returns the tangent of z .	
asin z	Primitive procedure
Returns the arcsine of z .	
acos z	Primitive procedure
Returns the arccosine of z .	
atan z	Primitive procedure
Returns the arctangent of z .	
atan y x	Primitive procedure
Returns the arctangent of the two variables y and x . It is similar to computing the arctangent of y / x , except that the signs of both arguments are used to determine the quadrant of the result.	
sqrt x	Primitive procedure
Returns the square root of its argument. Since complex numbers are not supported, x must be positive.	
expt $z1$ $z2$	Primitive procedure
Returns $z1$ raised to the power of $z2$.	
exact->inexact z	Primitive procedure
Returns z , converted to an inexact value.	
inexact->exact z	Primitive procedure
Returns z , converted to an exact value. An error signalled if a conversion is not possible without losing precision.	
number->string z	Primitive procedure
number->string z $radix$	Primitive procedure
Converts its argument to a string. The radix $radix$ is used if given.	
string->number $string$	Primitive procedure
string->number $string$ $radix$	Primitive procedure
Converts $string$ to a number. When given, $radix$ is used as the default radix which can be overridden with an explicit radix prefix in $string$. #f is returned if $string$ is not properly formatted.	
lognot n	Primitive procedure
Returns the bitwise negated value of the argument.	

- logand** *n . . .* Primitive procedure
 Performs a bitwise *and* operation on all arguments and returns the result. Returns a value with all bits set if called without arguments.
- logior** *n . . .* Primitive procedure
 Performs a bitwise *or* operation on all arguments and returns the result. Returns a value with no bits set if called without arguments.
- logxor** *n . . .* Primitive procedure
 Performs a bitwise *exclusive or* operation on all arguments and returns the result. Returns a value with no bits set if called without arguments.
- lsh** *n1 n2* Primitive procedure
 Shifts the integer *n1* left by *n2* bits if *n2* is not less than zero, or shifts *n2* logically right by *-n2* bits if *n2* is less than zero.
- ash** *n1 n2* Primitive procedure
 Shifts the integer *n1* left by *n2* bits if *n2* is not less than zero, or shifts *n2* arithmetically right by *-n2* bits if *n2* is less than zero.
- gcd** [*n1* [*n2*]] Primitive procedure
 Returns the greatest common divisor of its arguments. Without arguments, 0 is returned, if only *n1* is given, *n1* is returned.
- lcm** [*n1* [*n2*]] Primitive procedure
 Returns the least common multiple of its arguments. Without arguments, 1 is returned, if only *n1* is given, *n1* is returned.
- signum** *x* Primitive procedure
 Return -1 if *x* is less than zero, 0 if *x* is equal to zero and 1 if *x* is greater than zero.

5.3 Boolean operations

The boolean operations documented in this section are always available.

- boolean?** *obj* Primitive procedure
 Returns **#t** if *obj* is a boolean value, **#f** otherwise.
- not** *obj* Primitive procedure
 Returns **#t** if *obj* is false, **#f** otherwise.
- boolean->integer** *b* Primitive procedure
 Return 0 if *b* is **#f**, 1 otherwise.
- integer->boolean** *i* Primitive procedure
 Return **#f** if *i* is 0, **#t** otherwise.

5.4 Pairs and lists

Sizzle implements all list and pair operations as defined in R5RS, plus the list operations present in SRFI-1. All procedures without a special remark are available at all times, whereas some procedures need inclusion of the module (`core list-lib`) prior to usage.

5.4.1 List constructors

cons *obj1 obj2* Primitive procedure
 Return a cons cell with a car of *obj1* and a cdr of *obj2*.

list *obj ...* Primitive procedure
 Returns a list whose elements are all arguments of the call to **list**. With no arguments, the empty list `()` is returned.

xcons *obj1 obj2* Primitive procedure
 Equivalent to `(cons obj2 obj1)`, and is of utility only as a value to be conveniently passed to higher-order procedures. (**xcons** stands for *exchanged cons*.)
 This procedure must be imported by using the module `(core list-lib)`.

cons* *elt1 elt2 ...* Primitive procedure
 Like **list**, but the last argument provides the tail of the constructed list.
`(cons* 1 2 3 4) => (1 2 3 . 4)`
 This procedure must be imported by using the module `(core list-lib)`.

make-list *n [fill]* Primitive procedure
 Return an *n*-element list, whose elements are all the value *fill*. If the *fill* argument is not given, the elements of the list are unspecified.
 This procedure must be imported by using the module `(core list-lib)`.

list-tabulate *n init-proc* Primitive procedure
 Returns an *n*-element list. Element *i* of the list, where $0 \leq i < n$, is produced by calling `(init-proc i)`. The dynamic order in which **init-proc** is applied to the indices is not specified.
 This procedure must be imported by using the module `(core list-lib)`.

list-copy *flist* Primitive procedure
 Copies the spine of the argument list.
 This procedure must be imported by using the module `(core list-lib)`.

circular-list *elt1 elt2 ...* Primitive procedure
 Constructs a circular list of the elements.
 This procedure must be imported by using the module `(core list-lib)`.

iota *count [start step]* Primitive procedure
 Returns a list containing the elements
`(start start+step ... start+(count-1)*step)`
 The *start* and *step* parameters default to 0 and 1, respectively. This procedure takes its name from the APL primitive.
 This procedure must be imported by using the module `(core list-lib)`.

5.4.2 List predicates

list? *obj* Primitive procedure
 Return **#t** if *obj* is a list, **#f** otherwise. Only finite lists pass this test, infinite lists return **#f** for this predicate.

- proper-list?** *x* Primitive procedure
 Return **#t** if *x* is a proper list – a finite, nil-terminated list. Return **#f** for any other object.
 This procedure must be imported by using the module (`core list-lib`).
- circular-list?** *x* Primitive procedure
 Return **#t** if *x* is a circular list.
 This procedure must be imported by using the module (`core list-lib`).
- dotted-list?** *x* Primitive procedure
 Return **#t** if *x* is a finite, non-nil-terminated list. This includes non-pair, non-() values (e.g. symbols, numbers), which are considered to be dotted lists of length 0.
 This procedure must be imported by using the module (`core list-lib`).
- pair?** *obj* Primitive procedure
 Returns **#t** if *obj* is a cons cell, **#f** otherwise.
- null?** *obj* Primitive procedure
 Returns **#t** if *obj* is the empty list, **#f** otherwise.
- null-list?** *list* Primitive procedure
list is a proper or circular list. Returns **#t** if the argument is the empty list (), and **#f** otherwise. It is an error to pass this procedure a value which is not a proper or circular list. This procedure is recommended as the termination condition for list-processing procedures that are not defined on dotted lists.
 This procedure must be imported by using the module (`core list-lib`).
- not-pair?** *x* Primitive procedure
 Return **#t** if *x* is not a pair. Provided as a procedure as it can be useful as the termination condition for list-processing procedures that wish to handle all lists, both proper and dotted.
 This procedure must be imported by using the module (`core list-lib`).
- list=** *elt= list1 . . .* Primitive procedure
 Determine list equality, given an element-equality procedure. Proper list *A* equals proper list *B* if they are of the same length and their corresponding elements are equal, as determined by *elt=*.
 It is an error to apply **list=** to anything except proper lists.
 This procedure must be imported by using the module (`core list-lib`).

5.4.3 List selectors

- car** *pair* Primitive procedure
pair must be a cons cell. Return the car of the cons cell *pair*.
- cdr** *pair* Primitive procedure
pair must be a cons cell. Return the cdr of the cons cell *pair*.
- caar** *pair* Primitive procedure
cadr *pair* Primitive procedure
 . . . Primitive procedure
cdddar *pair* Primitive procedure
cddddr *pair* Primitive procedure
 These functions are compositions of **car** and **cdr**, where for example **caddr** could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

There are twenty-eight of these functions defined.

list-ref *list k* Primitive procedure
Returns the *k*th element of *list*. An error is signalled if *list* has fewer than *k* elements.

list-tail *list k* Primitive procedure
Returns the sublist of *list* obtained by omitting the first *k* elements. An error is signalled if *list* has fewer than *k* elements.

first *pair* Primitive procedure
second *pair* Primitive procedure
third *pair* Primitive procedure
fourth *pair* Primitive procedure
fifth *pair* Primitive procedure
sixth *pair* Primitive procedure
seventh *pair* Primitive procedure
eighth *pair* Primitive procedure
ninth *pair* Primitive procedure
tenth *pair* Primitive procedure

Synonyms for `car`, `cadr`, `caddr`, ...

These procedures must be imported by using the module `(core list-lib)`.

car+cdr *pair* Primitive procedure
The fundamental pair deconstructor. Return two values, the first being the *car*, the second being the *cdr* of the pair.

This procedure must be imported by using the module `(core list-lib)`.

take *list k* Primitive procedure
Return a list containing the first *k* elements of *list*. Return *list* if *k* is larger than the length of *list*.

This procedure must be imported by using the module `(core list-lib)`.

drop *list k* Primitive procedure
Return a list containing the elements of *list*, but without the first *k* elements. Return *list* if *k* is less or equal to zero, returns the empty list if *k* is larger than the length of *list*.

This procedure must be imported by using the module `(core list-lib)`.

take-right *flist i* Primitive procedure
drop-right *flist i* Primitive procedure

`take-right` returns the last *i* elements of *flist*.

`drop-right` returns all but the last *i* elements of *flist*.

The returned list may share a common tail with the argument list.

These procedures must be imported by using the module `(core list-lib)`.

take! *x i* Primitive procedure
drop-right! *x i* Primitive procedure

`take!` and `drop-right!` are destructive variants of `take` and `drop-right`. If *x* is circular, `take!` may return a shorter-than-expected list.

These procedures must be imported by using the module `(core list-lib)`.

split-at *x i* Primitive procedure
split-at! *x i* Primitive procedure
split-at splits the list *x* at index *i*, returning two values: a list of the first *i* elements and the remaining tail.
split-at! is the destructive variant of **split-at**.
 These procedures must be imported by using the module (`core list-lib`).

last *pair* Primitive procedure
last-pair *pair* Primitive procedure
last returns the last element of the non-empty, finite list *pair*. **last-pair** returns the last pair of the non-empty, finite list *pair*.

5.4.4 Miscellaneous list procedures

length *list* Primitive procedure
length+ *clist* Primitive procedure
length Returns the length of *list*. It is an error to pass anything but a proper, nil-terminated list to **length**.
length+, on the other hand, will return `#f` when applied to a circular list.
length+ must be imported by using the module (`core list-lib`), whereas **length** is always available.

append *list . . .* Primitive procedure
 Returns a list made out of the elements of all argument lists. The last argument may be an improper list, an improper list will be returned in this case.

append! *list . . .* Primitive procedure
 Destructive version of **append**. The result of the operation is the same as for **append**, but as a side effect, the parameter lists are modified when performing the append operation.

concatenate *list-of-lists* Primitive procedure
concatenate! *list-of-lists* Primitive procedure
 These functions append the elements of their arguments together. **concatenate!** may alter the cons cells of the passed lists.
 This procedure must be imported by using the module (`core list-lib`).

reverse *list* Primitive procedure
 Returns a newly allocated list with all elements of *list* in reversed order.

reverse! *list* Primitive procedure
 Destructive version of **reverse**. The result of the operation is the same as for **reverse**, but as a side effect, the parameter list is modified when performing the reverse operation.

append-reverse *rev-head tail* Procedure procedure
append-reverse! *rev-head tail* Procedure procedure
append-reverse returns (`append (reverse rev-head) tail`). It is provided because it is a common operation.
append-reverse! is just the destructive variant.
 These procedures must be imported by using the module (`core list-lib`).

zip *clist1 clist2 ...* Library procedure

If `zip` is passed n lists, it returns a list as long as the shortest of these lists, each element of which is an n -element lists comprised of the corresponding elements from the parameters lists. At least one of the argument lists must be finite.

This procedure must be imported by using the module (`core list-lib`).

unzip1 *list* Library procedure

unzip2 *list* Library procedure

unzip3 *list* Library procedure

unzip4 *list* Library procedure

unzip5 *list* Library procedure

`unzip1` takes a list of lists, where every list must contain at least one element, and returns a list containing the initial element of each such list. That is, it returns (`map car lists`).

`unzip2` takes a list of lists, where every list must contain at least two elements, and returns two values: a list of the first elements and a list of the second elements, `unzip3` does the same for the first three elements of the lists, and so forth.

```
(unzip2 '((one 1) (two 2) (three 3))) =>
(1 2 3)
(one two three)
```

These procedures must be imported by using the module (`core list-lib`).

count *pred clist1 clist2 ...* Primitive procedure

pred is a procedure taking as many arguments as there are lists and returning a single value. It is applied element-wise to the elements of the *lists*, and a count is tallied of the number of elements that return a true value. This count is returned. `count` is iterative in that it is guaranteed to apply *pred* to the *list* elements in a left-to-right order. The counting stops when the shortest list expires. At least one of the argument lists must be finite.

This procedure must be imported by using the module (`core list-lib`).

5.4.5 Fold, unfold and map

fold *kons knil clist1 clist2 ...* Primitive procedure

The fundamental list iterator.

First, consider the single list-parameter case if `clist1=(e1 e2 ... en)` then this procedure returns

```
(kons en ... (kons e1 (kons e1 knil)))
```

That is, it obeys the (tail) recursion

```
(fold kons knil lis) = (fold kons (kons (car lis) knil) (cdr lis))
(fold kons knil '()) = knil
```

If n argument lists are provided, then the *kons* procedure must take $n+1$ parameters: one element from each list, and the “seed” or fold state, which is initially *knil*. The fold operation terminates when the shortest list runs out of values. At least one of the argument lists must be finite.

This procedure must be imported by using the module (`core list-lib`).

pair-fold *kons knil clist1 clist2 ...* Primitive procedure

Analogous to `fold`, but *kons* is applied to successive sublists of the lists, rather than successive elements – that is, *kons* is applied to the pairs making up the lists.

For finite lists, the *kons* function may reliably apply `set-cdr!` to the pairs it is given without altering the sequence of execution.

This procedure must be imported by using the module (`core list-lib`).

reduce *f ridentity list* Library procedure

`reduce` is a variant of `fold`.

ridentity should be a “right identity” of the procedure *f* – that is, for any value *x* acceptable to *f*,

$$(f\ x\ ridentity) = x$$

Note that *ridentity* is used *only* in the empty-list case. You typically use `reduce` when applying *f* is expensive and you’d like to avoid the extra application incurred when `fold` applies *f* to the head of the *list* and the identity value, redundantly producing the same value passed to *f*.

This procedure must be imported by using the module `(core list-lib)`.

reduce-right *f ridentity list* Library procedure

`reduce-right` is the fold-right variant of `reduce`.

This procedure must be imported by using the module `(core list-lib)`.

unfold *p f g seed [tail-gen]* Library 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 the next seed value.

seed The “state” value for the unfold.

tail-gen Creates the tail of the list; defaults to `(lambda (x) '())`.

In other words, we use *g* to generate a sequence of seed values *seed*, *g*(*seed*), *g*²(*seed*), *g*³(*seed*), ...

These seed values are mapped to list elements by *f*, producing the elements of the result list in a left-to-right order. *p* says when to stop.

This procedure must be imported by using the module `(core list-lib)`.

unfold-right *p f g seed [tail]* Library procedure

`unfold-right` constructs 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 the next seed value.

seed The “state” value for the unfold.

tail List terminator, defaults to `'()`.

This procedure must be imported by using the module `(core list-lib)`.

- map** *proc list1 list2 . . .* Primitive procedure
 The *lists* must be lists and *proc* must be a procedure taking as many arguments as there are *lists* and returning a single value. All *lists* must be of the same length. **map** applies *proc* element-wise to the elements of the *lists* and returns a list of the results, in order. The dynamic order in which *proc* is applied to the elements of the *lists* is unspecified.
- for-each** *proc list1 list2 . . .* Primitive procedure
proc is applied to the elements of the *lists* as with **map**, but in order. **for-each** is for its side effects of *proc*, therefore the return value is unspecified.
- append-map** *f clist1 clist2 . . .* Library procedure
append-map! *f clist1 clist2 . . .* Library procedure
 Map *f* over the elements of the lists, just as 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.
 These procedures must be imported by using the module (`core list-lib`).
- map!** *f list clist1 . . .* Library procedure
 Destructive variant of **map**. **map!** may alter the cons cells of *list1* to construct the result list.
 This procedure must be imported by using the module (`core list-lib`).
- map-in-order** *f clist1 clist2 . . .* Library procedure
 A variant of the **map** procedure that guarantees to apply *f* across the elements of the list arguments in a left-to-right order. This is useful for mapping procedures that both have side effects and return useful values. At least one of the argument lists must be finite.
 This procedure must be imported by using the module (`core list-lib`).
- pair-for-each** *f clist1 clist2 . . .* Library procedure
 Like **for-each**, but *f* is applied to successive sublists of the argument lists. That is, *f* is applied to the cons cells of the lists, rather than the lists' elements. These applications occur in left-to-right order.
 This procedure must be imported by using the module (`core list-lib`).
- filter-map** *f clist1 clist2 . . .* Library procedure
 Like **map**, but only true values are saved. The dynamic order in which the various applications of *f* are made is not specified. At least one of the argument lists must be finite.
 This procedure must be imported by using the module (`core list-lib`).

5.4.6 Filtering and partitioning

- filter** *pred list* Library procedure
filter! *pred list* Library procedure
 Return all the elements of *list* that satisfy predicate *pred*. The list is not disordered. The returned list may share a common tail with the argument list. The dynamic order in which the various applications of *pred* are made is not specified.
filter! is the destructive variant of **filter**.
 These procedures must be imported by using the module (`core list-lib`).

partition *pred list* Library procedure
partition! *pred list* Library procedure
 Partitions the elements of *list* with predicate *pred*, and returns two values: the list of in-elements and the list of out-elements. The list is not disordered. The dynamic order in which the various applications of *pred* are made is not specified. One of the returned lists may share a common tail with the argument list.
partition! is the destructive variant of **partition**.
 These procedures must be imported by using the module (`core list-lib`).

remove *pred list* Library procedure
remove! *pred list* Library procedure
 Returns *list* without the elements that satisfy the predicate *pred*. The list is not disordered. The dynamic order in which the various applications of *pred* are made is not specified.
remove! is the destructive variant of **remove**.
 These procedures must be imported by using the module (`core list-lib`).

5.4.7 List searching

find *pred clist* Library procedure
 Return the first element in *clist* that satisfies predicate *pred*, #f if no element does.
 This procedure must be imported by using the module (`core list-lib`).

find-tail *pred clist* Library procedure
 Return the first pair of *clist* whose car satisfies predicate *pred*, #f if no element does.
 This procedure must be imported by using the module (`core list-lib`).

take-while *pred clist* Library procedure
take-while! *pred clist* Library procedure
 Return the longest initial prefix of *clist* whose elements all satisfy the predicate *pred*.
take-while! is the destructive variant of **take-while**.
 These procedures must be imported by using the module (`core list-lib`).

drop-while *pred clist* Library procedure
 Drops the longest initial prefix of *clist* whose elements all satisfy the predicate *pred*, and returns the rest of the list.
 This procedure must be imported by using the module (`core list-lib`).

span *pred clist* Library procedure
span! *pred clist* Library procedure
break *pred clist* Library procedure
break! *pred clist* Library procedure
span splits the list into the longest initial prefix whose elements all satisfy *pred*, and the remaining tail. **break** inverts the sense of the predicate: the tail commences with the first element of the input list that satisfies the predicate.
span! and **break!** are the destructive variants.
 This procedure must be imported by using the module (`core list-lib`).

any *pred clist1 clist2 . . .* Library procedure
 Applies the predicate across the lists, returning true if the predicate returns true on any application. If there are *n* arguments *clist1 . . . clistn*, then *pred* must be a procedure taking *n* arguments and returning a boolean result.

The iteration stops when a true value is produced or one of the lists runs out of values, in the latter case, **any** returns **#f**.

Note the difference between **find** and **any** – **find** returns the element that satisfied the predicate, **any** returns the true value that the predicate produced.

This procedure must be imported by using the module (`core list-lib`).

every *pred clist1 clist2 . . .* Library procedure

Applies the predicate across the lists, returning true if the predicate returns true on every application. If there are n arguments *clist1 . . . clistn*, then *pred* must be a procedure taking n arguments and returning a boolean result.

Like **any**, **every**'s name does not end with a question mark – this is to indicate that it does return a general value, not only **#t** or **#f**.

This procedure must be imported by using the module (`core list-lib`).

list-index *pred clist1 clist2 . . .* Primitive procedure

Return the index of the leftmost element that satisfies *pred*. If there are n arguments *clist1 . . . clistn*, then *pred* must be a procedure taking n arguments and returning a boolean result.

This procedure must be imported by using the module (`core list-lib`).

memq *obj list* Primitive procedure

memv *obj list* Primitive procedure

member *obj list [=]* Primitive procedure

These functions return the first sublist of *list* whose car is *obj*. **memq** uses **eq?** for comparing the values, **memv** uses **eqv?** and **member** uses **equal?**. If *obj* does not occur in *list*, **#f** is returned.

member is extended from R5RS to allow the client to pass an optional equality procedure = used to compare keys.

5.4.8 Association lists

assq *obj alist* Primitive procedure

assv *obj alist* Primitive procedure

assoc *obj alist [=]* Primitive procedure

alist (for association list) must be a list of pairs. These procedures return the first pair in *alist* whose car field is *obj*. If no pair in *alist* has *obj* as its car, then **#f** is returned. **assq** uses **eq?** for comparing the values, **assv** uses **eqv?** and **assoc** uses **equal?**.

assoc is extended from R5RS to allow the client to pass in an optional equality procedure = to compare keys.

alist-cons *key datum alist* Primitive procedure

Cons a new alist entry mapping from *key* to *datum* onto *alist*.

This procedure must be imported by using the module (`core list-lib`).

alist-copy *alist* Primitive procedure

Make a fresh copy of *alist*. This means copying each pair that forms an association as well as the spine of the list.

This procedure must be imported by using the module (`core list-lib`).

alist-delete *key alist [=]* Primitive procedure

alist-delete! *key alist [=]* Primitive procedure

alist-delete deletes all associations from *alist* with the given *key*, using key-comparison procedure =, which defaults to **equal?**. The dynamic order in which the various applications of = are made is not specified.

`alist-delete!` is the destructive variant of `alist-delete`. It may modify its argument list in order to produce its result.

This procedure must be imported by using the module (`core list-lib`).

5.4.9 Deletion

`delete` *x list* [=] Primitive procedure
`delete!` *x list* [=] Primitive procedure

`delete` uses the comparison procedure `=`, which defaults to `equal?`, to find all elements of *list* that are equal to *x*, and deletes them from *list*. The dynamic order in which the various applications of `=` are made is not specified. The list is not disordered – elements that appear in the result list occur in the same order as they occur in the argument list.

`delete!` is the destructive variant of `delete`. It is allowed to alter the cons cells in its argument list to construct the result.

These procedures must be imported by using the module (`core list-lib`).

`delete-duplicates` *list* [=] Primitive procedure
`delete-duplicates!` *list* [=] Primitive procedure

`delete-duplicates` removes duplicate elements from the list argument. If there are multiple equal elements in the argument list, the result list only contains the first or leftmost of these elements in the result. The order of the surviving elements is the same as in the original list.

`delete-duplicates!` is the destructive variant of `delete-duplicates`. It is allowed to alter the argument list to produce the result.

These procedures must be imported by using the module (`core list-lib`).

5.4.10 Primitive side-effects

`set-car!` *pair obj* Primitive procedure
pair must be a cons cell. Sets the car of *pair* to *obj*. The return value is not specified.

`set-cdr!` *pair obj* Primitive procedure
pair must be a cons cell. Sets the cdr of *pair* to *obj*. The return value is not specified.

5.4.11 Set operations on lists

`lset<=` = *list1* . . . Library procedure
 Returns true iff every *list_i* is a subset of *list_{i+1}*, using `=` for the element-equality procedure.

This procedure must be imported by using the module (`core list-lib`).

`lset=` = *list1* . . . Library procedure
 Returns true iff every *list_i* is a set-equal of *list_{i+1}*, using `=` for the element-equality procedure.

This procedure must be imported by using the module (`core list-lib`).

`lset-adjoin` = *list elt1* . . . Library procedure
 Adds the *elts* not already in the list parameter to the result list. The result shares a common tail with the list parameter. The `=` parameter is an equality procedure used to determine if an *elt_i* is already a member of *list*.

The list parameter is always a suffix of the result – even if the list parameter contains repeated elements, these are not reduced.

This procedure must be imported by using the module (`core list-lib`).

- lset-union** = *list1* ... Library procedure
Returns the union of the lists, using = for the element-equality procedure.
This procedure must be imported by using the module (`core list-lib`).
- lset-intersection** = *list1* ... Library procedure
Returns the intersection of the lists, using = for the element-equality procedure.
This procedure must be imported by using the module (`core list-lib`).
- lset-difference** = *list1 list2* ... Library procedure
Returns the intersection of the lists, using = for the element-equality procedure.
This procedure must be imported by using the module (`core list-lib`).
- lset-xor** = *list1* ... Library procedure
Returns the exclusive-or of the sets, using = for the element-equality procedure. The result is a list of all elements which appear in an odd number of the parameter *lists*.
This procedure must be imported by using the module (`core list-lib`).
- lset-diff+intersection** = *list1 list2* ... Library procedure
Returns two values – the difference and the intersection of the lists.
This procedure must be imported by using the module (`core list-lib`).
- lset-union!** = *list1* ... Library procedure
lset-intersection! = *list1 list2* ... Library procedure
lset-difference! = *list1 list2* ... Library procedure
lset-xor! = *list1* ... Library procedure
lset-diff+intersection! = *list1 list2* ... Library procedure
These are destructive variants of the pure functions.
These procedures must be imported by using the module (`core list-lib`).

5.5 Hash tables

These procedures are always available.

- hashq** *obj n* Primitive procedure
hashv *obj n* Primitive procedure
hash *obj n* Primitive procedure
Determine a hash value for *obj* that is suitable for lookups in a hash table of size *n*, where `eq?`, `eqv?` or `equal?` (for `hashq`, `hashv`, `hash`, respectively) is used as the equality predicate. The function returns an integer in the range 0 to *n* - 1. Note that these functions may use internal addresses. Thus two calls to the same function where the keys are equal according to the sameness predicate used are not guaranteed to deliver the same value if the key object *obj* gets garbage collected in between.
- make-hash** [*capacity*] *pair* ... Primitive procedure
Creates a hash table, initialized with values taken from the *pairs*. The car of each pair is taken as a hash key and the corresponding cdr as the value. If the first argument is not a pair, it must be an integer and will give the number of buckets to use for the hash table.
- hashq-get-handle** *tab key* Primitive procedure
hashv-get-handle *tab key* Primitive procedure
hash-get-handle *tab key* Primitive procedure
These procedures are similar to their `-ref` cousins, but return *handles* from the hash table rather than the value associated with *key*. By convention, a handle in a hash table is the pair which associates a key with a value. Where `hashq-ref tab key` returns only a value, `hashq-get-handle table key` returns the pair (`key . value`).

hashq-ref <i>tab key</i> [<i>default</i>]	Primitive procedure
hashv-ref <i>tab key</i> [<i>default</i>]	Primitive procedure
hash-ref <i>tab key</i> [<i>default</i>]	Primitive procedure
Look up <i>key</i> in the hash table <i>tab</i> , and return the value (if any) associated with it. If <i>key</i> is not found, return <i>default</i> (or #f if no <i>default</i> argument is supplied). hashq-ref uses eq? for equality testing, the other functions use eqv? and equal? .	
hashq-create-handle! <i>tab key value</i>	Primitive procedure
hashv-create-handle! <i>tab key value</i>	Primitive procedure
hash-create-handle! <i>tab key value</i>	Primitive procedure
These functions look up <i>key</i> in <i>tab</i> and return its handle. If <i>key</i> is not already present, a new handle is created which associates <i>key</i> with <i>init</i> .	
hashq-set! <i>tab key value</i>	Primitive procedure
hashv-set! <i>tab key value</i>	Primitive procedure
hash-set! <i>tab key value</i>	Primitive procedure
Find the entry in <i>tab</i> associated with <i>key</i> , and store <i>value</i> there. hashq-set! uses eq? for equality testing, the other functions use eqv? and equal? . All these procedures return the <i>value</i> .	
hashq-remove! <i>tab key</i>	Primitive procedure
hashv-remove! <i>tab key</i>	Primitive procedure
hash-remove! <i>tab key</i>	Primitive procedure
Remove <i>key</i> (and any value associated with it) from <i>tab</i> . hashq-remove! uses eq? for equality testing, the other functions use eqv? and equal? . The procedures return the handle (that is, the key–value pair) that was removed or #f if the key was not found.	
hash-fold <i>proc seed table</i>	Primitive procedure
Iterate over the elements of the hash table <i>table</i> . <i>proc</i> is applied to every hash table element with the arguments (proc key value init), where on the first call <i>init</i> is the given <i>seed</i> , and on subsequent calls <i>init</i> is the result of the previous call.	

5.6 Symbol operations

The procedures in this section do not need any modules to be loaded.

symbol? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a symbol, #f otherwise.	
symbol->string <i>symbol</i>	Primitive procedure
Converts <i>symbol</i> to a string.	
string->symbol <i>string</i>	Primitive procedure
Converts <i>string</i> to a symbol.	
gensym [<i>prefix</i>]	Primitive procedure
Generate a new symbol, constructed from <i>prefix</i> and the value of a counter which is incremented after each call. <i>prefix</i> defaults to)gen , if not given.	

5.7 Character operations

All of these procedures are available without loading additional modules.

char? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a character value, #f otherwise.	

char= <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> is the same character as <i>char2</i> , #f otherwise.	
char< <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a smaller character value than <i>char2</i> , #f otherwise.	
char<= <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a smaller or equal character value than <i>char2</i> , #f otherwise.	
char> <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a greater character value than <i>char2</i> , #f otherwise.	
char>= <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a greater or equal character value than <i>char2</i> , #f otherwise.	
char-ci= <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> is the same character as <i>char2</i> , #f otherwise. This function ignores case when comparing.	
char-ci< <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a smaller character value than <i>char2</i> , #f otherwise. This function ignores case when comparing.	
char-ci<= <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a smaller or equal character value than <i>char2</i> , #f otherwise. This function ignores case when comparing.	
char-ci> <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a greater character value than <i>char2</i> , #f otherwise. This function ignores case when comparing.	
char-ci>= <i>char1 char2</i>	Primitive procedure
Returns #t if <i>char1</i> has a greater or equal character value than <i>char2</i> , #f otherwise. This function ignores case when comparing.	
char-alphabetic? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an alphabetic character.	
char-numeric? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a numeric character.	
char-whitespace? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is whitespace character. The characters space, tab, line feed, form feed and carriage return are considered to be whitespace.	
char-upper-case? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is an upper case character.	
char-lower-case? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a lower case character.	
char->integer <i>char</i>	Primitive procedure
Convert the character <i>char</i> to its ASCII value.	
integer->char <i>n</i>	Primitive procedure
Returns a character with ASCII value <i>i</i> .	

char-upcase *char* Primitive procedure
Returns the upper case version of character *char* if *char* is a lower case character and *char* otherwise.

char-downcase *char* Primitive procedure
Returns the lower case version of character *char* if *char* is an upper case character and *char* otherwise.

5.8 String operations

For convenient string handling, a lot of string primitives have been included into the Sizzle core. Unless otherwise noted, these procedures do not need the inclusion of additional modules.

5.8.1 String Predicates

string? *obj* Primitive procedure
Return **#t** if *obj* is a string, **#f** otherwise.

string-null? *obj* Primitive procedure
Return **#t** if *obj* is the empty string "", **#f** otherwise.

string-every *pred s [start end]* Primitive procedure
Returns a true value, if the applications of *pred* to all of the characters of *s* yield true values. *start* and *end* may be given to restrict operation to the characters of *s* between these indices.

string-any *pred s [start end]* Primitive procedure
Returns a true value, if the application of *pred* to any of the characters of *s* yields a true value. *start* and *end* may be given to restrict operation to the characters of *s* between these indices.

5.8.2 String Constructors

make-string *k* Primitive procedure
make-string *k char* Primitive procedure

Return a newly allocated string of length *k*. *char* is used to initialize the string, if omitted, the contents of the string is unspecified.

string *char ...* Primitive procedure
Return a newly allocated string composed of its arguments, which all must be characters.

string-append *string ...* Primitive procedure
Return a string made out of the concatenation of all string arguments. An empty string is returned if called without arguments.

string->list *string [start end]* Primitive procedure
Convert the string *string* to a list of all characters of *string*. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

list->string *lst* Primitive procedure
Convert the list *lst*, which must be made up from character values, to a string.

reverse-list->string *lst* Primitive procedure
An efficient implementation of (compose string->list reverse): (reverse-list->string '(#\a #\B #\c)) => "cBa"

- string-copy** *string* [*start end*] Primitive procedure
Returns a newly allocated string with the same contents as *string*. *start* and *end* delimit the string to be copied, they default to 0 and the length of *string*.
- string-tabulate** *proc len* Primitive procedure
Create a string of length *len*, where each character is initialized from the value returned by applying *proc* to the character index.
- string-join** *string-list delimiter grammar* Primitive procedure
string-list must be a list of strings, *delimiter* must be a string and *grammar* must be any of the symbols *infix*, *strict-infix*, *prefix* or *suffix*. The strings from *string-list* are joined with *delimiter* inbetween according to *grammar*.
This procedure will throw an exception if *string-list* is empty and *grammar* is **string-infix**.
- string-reverse** *s* [*start end*] Primitive procedure
string-reverse! *s* [*start end*] Primitive procedure
Reverse the string argument *s*. *start* and *end* are optional start and end indices and default to 0 and the length of *s*, respectively. **string-reverse** returns a reversed version of *s*, whereas **string-reverse!** reverses *s* in place and returns an unspecified value.

5.8.3 String Accessors

- string-length** *string* Primitive procedure
Return the number of characters in *string*.
- string-ref** *string k* Primitive procedure
Return the *k*th character of *string*. The index is zero-based. *k* must be a valid index into *string*, or an error will be signalled.
- string-set!** *string k char* Primitive procedure
Store *char* in element *k* of *string*. An error will be signalled if *k* is not a valid index into *string*. The return value of this function is unspecified.
- substring** *string start end* Primitive procedure
Return a string formed from the characters of *string* starting at *start* (inclusive) and ending with *end* (exclusive). An error is signalled if the following equation is not satisfied: $0 \leq start \leq end \leq (\text{string-length } string)$
- string-fill!** *string char* [*start end*] Primitive procedure
Stores the character *char* in all elements of *string*. The return value is not specified. *start* and *end* delimit the region which is filled and default to 0 and the length of *string*.

5.8.4 String Comparison

- string=** *string1 string2* Primitive procedure
Return **#t** if *string1* is the same string as *string2*, **#f** otherwise.
- string<?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically less than *string2*, **#f** otherwise.
- string<=?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically less or equal to *string2*, **#f** otherwise.

- string>?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically greater than *string2*, **#f** otherwise.
- string>=?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically greater or equal to *string2*, **#f** otherwise.
- string-ci=?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is the same string as *string2*, **#f** otherwise. This function ignores case when comparing.
- string-ci<?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically less than *string2*, **#f** otherwise. This function ignores case when comparing.
- string-ci<=?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically less or equal to *string2*, **#f** otherwise. This function ignores case when comparing.
- string-ci>?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically greater than *string2*, **#f** otherwise. This function ignores case when comparing.
- string-ci>=?** *string1 string2* Primitive procedure
Returns **#t** if *string1* is lexicographically greater or equal to *string2*, **#f** otherwise. This function ignores case when comparing.
- string-prefix-length** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return the length of the longest common prefix of the two strings.
- string-prefix-length-ci** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return the length of the longest common prefix of the two strings. When comparing characters, case is ignored.
- string-suffix-length** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return the length of the longest common suffix of the two strings.
- string-suffix-length-ci** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return the length of the longest common suffix of the two strings. Case is ignored when comparing characters.
- string-prefix?** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return **#t** if *s1* is a prefix of *s2*.
- string-prefix-ci?** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return **#t** if *s1* is a prefix of *s2*. Case is ignored when comparing characters.
- string-suffix?** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return **#t** if *s1* is a suffix of *s2*.
- string-suffix-ci?** *s1 s2 [start1 end1 start2 end2]* Primitive procedure
Return **#t** if *s1* is a suffix of *s2*. Case is ignored when comparing characters.

5.8.5 String Searching

string-index *string ch/pred [start end]* Primitive procedure
 Scans the string *string* for the leftmost appearance of character *ch* and returns the index or **#f** if not found. If the second argument is a procedure, it must be a predicate which will be applied to each character in *string*, and the index of the first character for which the predicate yields true is returned. *start* is the start index and defaults to zero, *end* is the end index and defaults to the length of *string* - 1.

string-index-right *string ch/pred [start end]* Primitive procedure
 Scans the string *string* for the rightmost appearance of character *ch* and returns the index or **#f** if not found. If the second argument is a procedure, it must be a predicate which will be applied to each character in *string* from right to left, and the index of the last character for which the predicate yields true is returned. *start* is the start index and defaults to zero, *end* is the end index and defaults to the length of *string* - 1. This function searches from right to left, whereas **string-index** searches from left to right.

string-skip *string ch/pred [start end]* Primitive procedure
 Scans the string *string* for the leftmost appearance of a character different to *ch* and returns the index or **#f** if not found. If the second argument is a procedure, it must be a predicate which will be applied to each character in *string*, and the index of the first character for which the predicate yields false is returned. *start* is the start index and defaults to zero, *end* is the end index and defaults to the length of *string* - 1.

string-skip-right *string ch/pred [start end]* Primitive procedure
 Scans the string *string* for the rightmost appearance of a character different from *ch* and returns the index or **#f** if not found. If the second argument is a procedure, it must be a predicate which will be applied to each character in *string* from right to left, and the index of the last character for which the predicate yields false is returned. *start* is the start index and defaults to zero, *end* is the end index and defaults to the length of *string* - 1. This function searches from right to left, whereas **string-index** searches from left to right.

string-count *string ch/pred [start end]* Primitive procedure
 Return a count of the number of characters in *string* that satisfy the *ch/pred* argument. If this argument is a procedure, it is applied to all characters as a predicate; if it is a character, it is used in an equality test. *start* is the start index and defaults to zero, *end* is the end index and defaults to the length of *string* - 1.

string-contains *string1 string2 [start1 end1 start2 end2]* Primitive procedure
 Return the index in *string1* where *string2* occurs as a substring, or **#f**. The optional *start/end* indices restrict the operation to the indicated substrings. The returned index is in the range [*start1*, *end1*). A successful match must lie entirely in the [*start1*, *end1*) range of *string1*.

Note: The current implementation uses the naivest possible algorithm and is slow when *string1* is a long string.

string-contains-ci *string1 string2 [start1 end1 start2 end2]* Primitive procedure
 Return the index in *string1* where *string2* occurs as a substring, or **#f**. The optional *start/end* indices restrict the operation to the indicated substrings. The returned index is in the range [*start1*, *end1*). A successful match must lie entirely in the [*start1*, *end1*) range of *string1*. Case is ignored when comparing individual characters.

Note: The current implementation uses the naivest possible algorithm and is slow when *string1* is a long string.

string-hash *s* [*bound start end*] Primitive procedure
 Compute a hash value for the string *s*. *bound* is either the boolean constant `#t` or a positive integer, then the result of the procedure will be in the range $[0, bound)$. *start* and *end* are optional parameters delimiting the characters which are used to calculate the hash value, they default to 0 and the length of *s*.

string-hash-ci *s* [*bound start end*] Primitive procedure
 Compute a hash value for the string *s*. *bound* is either the boolean constant `#t` or a positive integer, then the result of the procedure will be in the range $[0, bound)$. *start* and *end* are optional parameters delimiting the characters which are used to calculate the hash value, they default to 0 and the length of *s*. The case of the characters in *s* is ignored when computing the hash value.

5.8.6 String Decomposition

string-split *ch string* Primitive procedure
 Splits the string *string* into pieces, using *ch* as the delimiting character. Returns a list of strings, where the strings may be empty if two or more delimiting characters are in *string* without characters inbetween.

string-take *s nchars* Primitive procedure
 Return the first *nchars* characters of *s*.

string-take-right *s nchars* Primitive procedure
 Return the last *nchars* characters of *s*.

string-drop *s nchars* Primitive procedure
 Return all but the first *nchars* characters of *s*.

string-drop-right *s nchars* Primitive procedure
 Return all but the last *nchars* characters of *s*.

5.8.7 String Case Conversion

string-upcase! *string* [*start end*] Primitive procedure
 Convert all characters in *string* to upper case, modifying *string* in-place. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

string-upcase *string* [*start end*] Primitive procedure
 Convert all characters in *string* to upper case, returning a newly allocated string. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

string-downcase! *string* [*start end*] Primitive procedure
 Convert all characters in *string* to lower case, modifying *string* in-place. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

string-downcase *string* [*start end*] Primitive procedure
 Convert all characters in *string* to lower case, returning a newly allocated string. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

string-titlecase! *string* [*start end*] Primitive procedure
 Convert all characters in *string* which start words to upper case and all other characters to lower case, modifying *string* in-place. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

string-titlecase *string* [*start end*] Primitive procedure
 Convert all characters in *string* which start words to upper case and all other characters to lower case, returning a newly allocated string. *start* and *end* delimit the region of operation and default to 0 and the length of *string*.

5.8.8 String Trimming

string-trim *string* [*char/pred start end*] Primitive procedure
 Skip all characters matching the character *char* or passing application of *pred* at the left of the string. *start* and *end* are optional parameters to restrict operation to the characters between these indices.

string-trim-right *string* [*char/pred start end*] Primitive procedure
 Skip all characters matching character *char* or passing application of *pred* at the right of the string. *start* and *end* are optional parameters to restrict operation to the characters between these indices.

string-trim-both *string* [*char/pred start end*] Primitive procedure
 Skip all characters matching the character *char* or passing application of *pred* at both ends of the string. *start* and *end* are optional parameters to restrict operation to the characters between these indices.

string-chop *string* Primitive procedure
 Return a newly allocated string with the contents of *string*, but with all trailing newline characters removed.

trim-whitespace *string* Library procedure
 Return *str*, but with all whitespace both at the beginning and at the end removed. The same characters are considered to be whitespace as for the `char-whitespace?` primitive.
 This procedure must be imported by using the module `core string-lib`.
 Note: This procedure has been deprecated. Use `string-trim-both` instead.

5.8.9 String Padding

string-pad *s len* [*char start end*] Primitive procedure
 Build a string of length *len* comprised of *s* padded on the left by as many occurrences of the character *char* as needed. If *s* has more than *len* characters, it is truncated on the left to length *len*. *char* defaults to `#\space`.

string-pad-right *s len* [*char start end*] Primitive procedure
 Build a string of length *len* comprised of *s* padded on the right by as many occurrences of the character *char* as needed. If *s* has more than *len* characters, it is truncated on the right to length *len*. *char* defaults to `#\space`.

5.8.10 String Pathname Operations

basename *name* [*suffix*] Primitive procedure
 Removes any leading directory components from *name*; if *suffix* is specified and is identical to the end of name, it is removed from *name* as well. The resulting string is returned.

dirname *filename* Primitive procedure
 Returns a string of all but the final slash-delimited component of *filename*. If *filename* is a single component, **dirname** returns `.` (meaning the current directory).

cleanup-filename *filename* Primitive procedure
 Clean the given *filename* up and convert it to a canonical form by removing all appearances of parent directory references (`/foo/..`), current directory marks (`./`), multiple slashes (`//`) and trailing slashes. The root directory indicator `/` is not deleted, though it is a trailing slash.

tilde-expand *string* Library procedure
 Expand character sequences like `~` and `~username` in *filename* like bash and Emacs do. *filename* is unchanged if it has the form `~username`, but the given user does not exist.

This procedure must be imported by using the module `core string-lib`.

5.9 Vector operations

Vectors are heterogenous data types, similar to lists. That means that any Scheme data type can be stored into vectors. The main difference between lists and vectors is that vectors are of fixed size and that elements can be accessed in constant time, whereas lists require linear time to access individual elements.

5.9.1 Vector constructors

Vectors can be created by either entering them literally in the normal Scheme vector syntax `#(...)`, or they can be created with one of these functions. See Section 5.9.4 [Vector conversion], page 44 for the procedure `list->vector`, which is also a constructor function.

Vectors created with one of these procedures are mutable and can therefore be passed to procedures like `vector-set!`.

make-vector *k* Primitive procedure
make-vector *k fill* Primitive procedure
 Return a newly allocated vector of length *k*. *fill* is used to initialize the vector, if omitted the contents of the vector is unspecified.

vector *obj ...* Primitive procedure
 Returns a newly allocated vector composed of its arguments.

vector-tabulate *count init-proc* Library procedure
 Create a vector with *count* elements. The vector is initialized by setting the vector element at position *i* to the value returned by applying *init-proc* to *i*. The dynamic order in which *init-proc* is applied is not specified.

This procedure must be imported by using the module `core vector-lib`.

vector-copy *vec* Library procedure
 Create a freshly allocated copy of the vector *vec*.

This procedure must be imported by using the module `core vector-lib`.

5.9.2 Vector predicates

vector? *obj* Primitive procedure
Returns **#t** if *obj* is a vector, **#f** otherwise.

vector= *elt= vec1 vec2 . . .* Library procedure
Returns true if all given vectors are equal. Two vectors are equal if they have the same length and if the elements at the corresponding indices are equal according to the procedure *elt=*.
This procedure must be imported by using the module `core vector-lib`.

5.9.3 Vector selectors

Use one of these procedures to obtain information from a vector and to access individual elements.

vector-length *vector* Primitive procedure
Returns the number of elements of *vector*.

vector-ref *vector k* Primitive procedure
Returns the *k*th element of *vector*. The index is zero-based. *k* must be a valid index into *vector*, or an error will be signalled.

5.9.4 Vector conversion

vector->list *vector* Primitive procedure
Convert the vector *vector* to a list of all elements of *vector*.

list->vector *lst* Primitive procedure
Convert the list *list* to a vector.

5.9.5 Vector modifying

The procedures in this section may only be used with mutable vectors. Vectors literally entered may not be passed to these procedures.

vector-set! *vector k obj* Primitive procedure
Stores *obj* in element *k* of *vector*. An error will be signalled if *k* is not a valid index into *vector*. The return value of this function is unspecified.

vector-fill! *vector obj* Primitive procedure
Stores *obj* in all elements of *vector*. The return value is not specified.

5.10 Homogenous Vector Operations

Homogenous vectors are different to normal Scheme vectors, because only numeric values can be stored into those vectors, and all elements must have the same type.

Please note that homogenous vectors are only available when they were included on configuration time of the Sizzle library. If you need these data types and primitives, please Section 2.8 [Building Sizzle], page 6 and build the library with enabled homogenous vectors.

In the following procedure descriptions, the placeholder *TAG* can be replaced by any of the strings `s8`, `u8`, `s16`, `u16`, `s32`, `u32`, `s64`, `u64`, `f32` and `f64`, depending on the data type the procedure should be used for. Examples are: `s8vector?`, `make-u16vector` or `f32vector-length`.

TAGvector? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> matches the type indicated by <i>TAG</i> , #f otherwise.	
make-TAGvector <i>n</i> [<i>fill</i>]	Primitive procedure
Create a vector for holding values of the type <i>TAG</i> , which can hold exactly <i>n</i> elements. <i>fill</i> is used to initialize the vector and defaults to 0, if not present.	
TAGvector-length <i>vec</i>	Primitive procedure
Returns the length of the homogenous vector <i>vec</i> , which must be of type <i>TAG</i> .	
TAGvector <i>obj1</i> . . .	Primitive procedure
Creates and returns a homogenous vector of type <i>TAG</i> and initializes it with all parameters. The resulting vector has as much elements as there are <i>obj</i> parameters.	
TAGvector-ref <i>vec n</i>	Primitive procedure
Returns the element at position <i>n</i> in the homogenous vector <i>vec</i> .	
TAGvector-set! <i>vec n obj</i>	Primitive procedure
Stores the object <i>obj</i> at position <i>n</i> into the homogenous vector <i>vec</i> . <i>vec</i> must be of the vector type indicated by <i>TAG</i> , and <i>obj</i> must match <i>TAG</i> also.	
TAGvector->list <i>vec</i>	Primitive procedure
Converts the homogenous vector <i>vec</i> , which must match the vector type indicated by <i>TAG</i> , to a list of its elements.	
list->TAGvector <i>list</i>	Primitive procedure
Creates a homogenous vector which holds all elements of the list <i>list</i> . All elements must match the type <i>TAG</i> .	
TAGvector-fill!! <i>vec fill</i>	Primitive procedure
Stores the object <i>fill</i> into all slots of the homogenous vector <i>vec</i> . <i>vec</i> must match the vector type indicated by <i>TAG</i> , and <i>fill</i> must be of a type compatible with <i>TAG</i> .	
string->s8vector <i>str</i>	Primitive procedure
string->u8vector <i>str</i>	Primitive procedure
Convert the string <i>str</i> into a signed or unsigned 8-bit vector.	
s8vector->string <i>vec</i>	Primitive procedure
u8vector->string <i>vec</i>	Primitive procedure
Convert the signed or unsigned 8-bit vector <i>vec</i> to a string.	

5.11 Control flow operations

The following procedures can be used to maintain the flow of control in Scheme programs. The basic procedure `apply`, exception handling using `catch` and `throw` and handling of multiple values with `values`, `call-with-values` and `receive` are documented in this section.

These procedures are always available in the Sizzle library except when otherwise noted.

procedure? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a primitive or user-defined procedure, #f otherwise.	
primitive-procedure? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a primitive procedure, #f otherwise.	
closure? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a user-defined procedure, #f otherwise.	

- form?** *obj* Primitive procedure
Returns `#t` if *obj* is a syntactic form, `#f` otherwise.
- apply** *proc arg1 . . . args* Primitive procedure
proc must be a procedure and *args* must be a list. Calls *proc* with the elements of the list (`append (list arg1 . . .) args`) as the actual arguments.
- force** *promise* Primitive procedure
Force the evaluation of *promise*, which must be created with a prior call to `delay`. The value of the promise is cached so that a second force will return the cached value without evaluating it again.
- values** *obj . . .* Primitive procedure
Delivers all its arguments to its continuation. Used with `call-with-values` (see below).
- call-with-values** *producer consumer* Primitive procedure
Calls its *producer* thunk and calls the *consumer* with whatever values have been returned by *producer*. *consumer* must take as many arguments as *producer* has returned values.
- receive** *formals expression body* Library syntax
This is a more convenient way to handle multiple values. *expression* is evaluated and the results are assigned to variables named in *formals*, then *body* is evaluated. *formals* must be of the same form as allowed for specifying the formal parameters of a lambda expression.
This syntax is not supported when the option `-n, --no-system-init-file` is passed to the interpreter, or if the system startup file is not found.
- dynamic-wind** *before thunk after* Primitive procedure
Calls *thunk* without arguments, returning the result(s) of the call. *before* is called before *thunk* is called, and *after* is called after *thunk*. `dynamic-wind` can be used for cleaning up, since *after* is called even if *thunk* causes an error or an exception.
- signal** *no proc* Primitive procedure
Install a signal handler. After installing a signal handler with this function, *proc* is called whenever the signal *no* is sent to the Sizzle process. The signals defined for your system are available as constants, on a normal GNU/Linux box the following signals are predefined:
- | | |
|---------|--|
| SIGHUP | Hangup signal. |
| SIGINT | Interrupt, normally generated when the user types <code>⏏</code> . |
| SIGQUIT | Quit. |
| SIGILL | Illegal instruction. |
| SIGTRAP | Trap. |
| SIGABRT | Process abort. |
| SIGIOT | Input/output trap. |
| SIGBUS | Bus error. Normally due to misaligned data access. |
| SIGFPE | Floating point exception. |
| SIGKILL | Kill signal. This cannot be caught. |
| SIGUSR1 | First user-definable signal. |

SIGUSR2	Second user-definable signal.
SIGSEGV	Segmentation fault.
SIGPIPE	Broken pipe. Generated when writing to a closed pipe.
SIGALRM	Alarm.
SIGTERM	Process termination.
SIGSTKFLT	Stack fault
SIGCLD	The same as SIGCHLD.
SIGCHLD	Child process terminated.
SIGCONT	Continue signal.
SIGSTOP	Stop signal.
SIGTSTP	Terminal stop.
SIGTTIN	Terminal input signal.
SIGTTOU	Terminal output signal.
SIGURG	Urgency signal.
SIGXCPU	CPU time exceeded.
SIGXFSZ	File limit exceeded.
SIGVTALRM	Virtual alarm.
SIPROF	Profiler signal.
SIGWINCH	Window configuration changed.
SIGPOLL	Poll signal.
SIGIO	Input/output signal.
SIGPWR	Power failure.

Note that this procedure is only available when the Un*x signals work for the underlying platform.

error *obj* . . . Primitive procedure

Signal an error. The current computation is aborted and the list of the arguments which is passed to the top level together with the error is displayed on the current standard error output port.

throw *symbol obj* . . . Primitive procedure

Raise an exception with the exception tag *tag* which will be passed upward in the call chain until either a **catch** clause for the *tag* exception tag catches the exception or the top level is reached, where the exception will be displayed on the current standard error output port.

catch *tag thunk handler* Primitive procedure

catch is used to implement exception handling. *thunk* is evaluated but any exception which may be thrown by an expression evaluated inside of *thunk* and is equal to the exception tag *tag* (in the sense of **eq?**) is caught. Should no exception be raised, the result of the last expression is returned. Otherwise, *handler* is called with the tag which was thrown to and additional arguments which may be passed to throw as arguments and the result of evaluating *handler* is returned.

If *tag* is **#t**, all exceptions are caught.

```

zzz: (catch 'panic
      (lambda () (throw 'panic "Help" "Me" "!") ))
      (lambda (tag . rest) (display rest) (newline) #f))
(Help Me !)
#f
zzz: (catch #t
      (lambda () (throw 'burgs))
      (lambda (tag . rest) #f))
#f

```

- toplevel** Primitive procedure
 Return to the top level read-eval-print loop of the current Sizzle session, or, if running non-interactively, returns to the top level file loading loop.
- exit** Primitive procedure
exit *n* Primitive procedure
 Terminate the running Sizzle process, returning to the shell which executed it. *n* is the exit code of the process, if given, otherwise the exit code is zero.
- call-with-current-continuation** *proc* Primitive procedure
call/cc *proc* Primitive procedure
 Calls *proc* with the current continuation as its argument. *proc* may then call the passed continuation with a parameter to deliver that value to the continuation. The result of **call-with-current-continuation** is either the result of *proc* (if the continuation was not called) or the argument passed to the continuation.
 When calling a continuation with more than one parameter, all parameters are delivered to the continuation as multiple values.
call/cc is synonym to **call-with-current-continuation**.
 Note that Sizzle does not implement a full-featured **call/cc**. The continuation passed to *proc* may only be used as an escape procedure. Storing it into a variable and using it from anywhere not in the dynamic extent of *proc* is an error and will signal an exception.
- primitive-load** *string* Primitive procedure
 Loads and evaluates all Scheme expressions from the file named *string*. Evaluation takes place in the toplevel environment. The return value is unspecified. Throws an **file-not-found** exception if *string* does not refer to an existing file. The return value is unspecified.
- primitive-load-path** *string* Primitive procedure
 Loads and evaluates all Scheme expressions from the file named *string*. Evaluation takes place in the toplevel environment. The return value is unspecified. When *string* is not an absolute file name, **primitive-load-path** tries to locate the file in the current directory and then in all directories in the pathname list *%load-path*. Throws an **file-not-found** exception if *string* does not refer to an existing file. The return value is unspecified.
- load** *string* Primitive procedure
 Synonym to **primitive-load-path**.
- transcript-on** *filename* Primitive procedure
transcript-off Primitive procedure
filename must be a string naming an output file to be created. The effect of **transcript-on** is to open the named file for output, and to cause a transcript of subsequent interaction between the user and the Scheme system to be written to the

file. The transcript is ended by a call to `transcript-off`, which closes the transcript file. Only one transcript may be in progress at any time. The values returned by these procedures are unspecified.

5.12 Eval function

All of the following procedures are always defined in the Sizzle interpreter.

- eval** *expression* Primitive procedure
- eval** *expression environment-specifier* Primitive procedure
- Evaluate *expression* in the environment given by *environment-specifier*. If omitted, the current environment is used. Environment specifiers can only be obtained by using one of the following three procedures.
- scheme-report-environment** *version* Primitive procedure
- Return an environment with all bindings defined in R5RS to be used with `eval`. *version* must be an exact integer specifying the supported Scheme report version, currently this is 5.
- null-environment** *version* Primitive procedure
- Return an empty environment to be used with `eval`. *version* must be an exact integer specifying the supported Scheme report version, currently this is 5.
- interaction-environment** Primitive procedure
- Return the environment of the read-eval-print loop to be used with `eval`.
- current-environment** Primitive procedure
- Return the environment currently used to look up bindings to be used with `eval`.
- safe-environment** *symbol-list* Primitive procedure
- Create and return a *safe* environment, in which only the special forms are defined. Also, the current bindings for all symbols in *symbol-list* are entered into the new environment. This procedure can be used to construct environments in which untrusted code can be safely executed. The executed code then only has access to the builtin special forms and all bindings passed explicitly to `safe-environment`, and is not able to affect any other environments.

5.13 Debugging Support

The following procedures are always supported.

- backtrace** Primitive procedure
- Prints a procedure backtrace for the last error signalled. You have to evaluate (`set! eval-save-backtrace #t`) prior to evaluating the invalid expression or the backtrace will not be available.
- trace** *proc on/off* Primitive procedure
- Note: Tracing is currently disabled because it slows down the interpreter and was not too useful. If you should need it, please contact the author.
- If *on/off* is true, start tracing procedure *proc*, otherwise, switch tracing for *proc* off.
- When a procedure is traced, a message will be printed to the current error port whenever the procedure is called. Given a procedure `foo`, which receives one argument, the format of the message is

```
+ foo <- (1)
```

That means that `foo` is entered (indicated by the `+`) with the parameter list `(1)`.

When tracing primitive procedures, a message when exiting from the procedure is also printed. A call to the traced primitive `modulo` with the parameter list `(3 2)` looks like this:

```
+ modulo <- (3 2)
- modulo -> 1
```

The second line tells that the result of the function is `1`.

Currently, no exit message is printed for non-primitive procedures because of problems with tail calls. With the current scheme, it is not possible to tell easily when a procedure exits. I hope to change that behaviour in the future.

symbol-table Primitive procedure
Returns the global symbol table, in which all globally existing symbols are stored. The returned table is a vector of lists of symbols, which can be examined if you are curious. Be careful when using this procedure and never modify the table or any data stored in it, or the interpreter may heavily crash.

keyword-table Primitive procedure
Returns the global keyword table, in which all globally existing keywords are stored. The returned table is a vector of lists of keywords, which can be examined if you are curious. Be careful when using this procedure and never modify the table or any data stored in it, or the interpreter may heavily crash.

5.14 Input and output

The procedures in this section all deal with input and output of data from and to various types of ports. Unless otherwise noted, all the procedures are always available in the interpreter.

call-with-input-file *string proc* Primitive procedure
Opens the file named *string* for input and calls *proc* with the newly created input port as a single argument.

call-with-output-file *string proc* Primitive procedure
Opens the file named *string* for output and calls *proc* with the newly created output port as a single argument.

input-port? *obj* Primitive procedure
Returns `#t` if *obj* is an input port, `#f` otherwise.

output-port? *obj* Primitive procedure
Returns `#t` if *obj* is an output port, `#f` otherwise.

current-input-port Primitive procedure
Returns the current default input port.

current-output-port Primitive procedure
Returns the current default output port.

current-error-port Primitive procedure
Returns the current default error output port.

set-current-input-port *port* Primitive procedure
Set *port* to be the value which subsequent calls to `current-input-port` will return.

- set-current-output-port** *port* Primitive procedure
Set *port* to be the value which subsequent calls to `current-output-port` will return.
- set-current-error-port** *port* Primitive procedure
Set *port* to be the value which subsequent calls to `current-error-port` will return.
- with-input-from-file** *string thunk* Primitive procedure
Opens the file named *string* for input and makes it the current default input port for the time *proc* is called.
- with-output-to-file** *string thunk* Primitive procedure
Opens the file named *string* for output and makes it the current default output port for the time *proc* is called.
- with-error-to-file** *string thunk* Primitive procedure
Opens the file named *string* for output and makes it the current default error port for the time *proc* is called.
- open-input-file** *filename* Primitive procedure
Returns an input port connected to the file named *filename*. An error is signalled if the file cannot be opened.
- open-output-file** *filename* Primitive procedure
Returns an output port connected to the file named *filename*. An error is signalled if the file cannot be opened. If a file with the same name already exists, it will be overwritten.
- close-port** *port* Primitive procedure
Close the port *port*.
- close-input-port** *port* Primitive procedure
Close the input port *port*. This is synonymous to `close-port`, but is required by R5RS.
- close-output-port** *port* Primitive procedure
Close the output port *port*. This is synonymous to `close-port`, but is required by R5RS.
- force-output** *port* Primitive procedure
Force all buffered output to *port* to be actually written to the underlying file. If *port* is a string port, nothing happens. The return value is unspecified.
- read** Primitive procedure
read *port* Primitive procedure
Read a scheme object from *port*. *port* is omitted, the current input port is used.
- read-char** Primitive procedure
read-char *port* Primitive procedure
Reads a character from *port* and returns it. *port* is omitted, the current input port is used. Returns the end-of-file object on end of file.
- unread-char** *char* Primitive procedure
unread-char *char port* Primitive procedure
Put the character *char* back on the port *port*, so that it can be read again with `read-char`. *port* defaults to the value returned by `current-input-port`, if not given.
- peek-char** Primitive procedure
peek-char *port* Primitive procedure
Reads a character from *port* and return it without removing it from the port. A subsequent call to `read-char` on the same port will return the character again. *port* is omitted, the current input port is used. Returns the end-of-file object on end of file.

- char-ready?** Primitive procedure
char-ready? *port* Primitive procedure
 Tests whether a character is available to be read from *port* and returns **#t** if it is, **#f** otherwise. If *port* is not specified, the current input port is used.
- read-line** [*port* [*handle-delim*]] Library procedure
read-line reads a newline-terminated string from *port*. The return value depends on the value of *handle-delim*, which may be one of the symbols **trim**, **concat**, **peek** and **split**. If it is **trim** (the default), the trailing newline is removed and the string is returned. If **concat**, the string is returned with the trailing newline intact. If **peek**, the newline is left in the input port buffer and the string is returned. If **split**, the newline is split from the string and **read-line** returns a pair consisting of the truncated string and the newline.
 This procedure is not supported when the option **-n**, **--no-system-init-file** is passed to the interpreter, or if the system startup file is not found.
- %read-delimited!** *delims buf gobble?* [*port start end*] Primitive procedure
 Read characters from *port* into *buf* until one of the characters in the *delims* string is encountered. If *gobble?* is true, store the delimiter character in *buf* as well; otherwise, discard it. If *port* is not specified, use the value returned by **current-input-port**. If *start* or *end* are specified, store data only into the substring of *buf* bounded by *start* and *end* (which default to the beginning and end of the buffer, 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 buffer was filled without encountering a delimiter, this value is *#f*.
- write-line** *obj* [*port*] Primitive procedure
 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])
```
- seek** *port offset whence* Primitive procedure
 Change the position of the file pointer for *port*. *offset* is used to calculate the new file offset, depending on *whence*. Three variable may be used for values to give as *whence*.
SEEK_SET *offset* is used as the new file offset.
SEEK_CUR *offset* is added to the current offset to yield the new file offset.
SEEK_END *offset* is subtracted from the file length to yield the new file offset.
port may also be a string port, which means that the read/write pointer for the string port will be modified.
 The return value is the new offset of the file pointer.
- ftell** *port* Primitive procedure
 Return the current offset of the file pointer of *port*, which may be a file or string port.
- eof-object?** *obj* Primitive procedure
 Returns **#t** if *obj* is the end-of-file object, **#f** otherwise.
- port-closed?** *port* Primitive procedure
 Returns **#t** if *port* is a closed port, **#f** otherwise.

- display** *obj* Primitive procedure
display *obj port* Primitive procedure
 Print the textual representation of *obj* to *port*. *obj* is not quoted when printing. **display**'s return value is unspecified. If *port* is omitted, the current output port is used.
- write** *obj* Primitive procedure
write *obj port* Primitive procedure
 Print the textual representation of *obj* to *port*. *obj* is quoted when printing. **write**'s return value is unspecified. If *port* is omitted, the current output port is used.
- write-char** *char* Primitive procedure
write-char *char port* Primitive procedure
 Write *char* to *port*. The actual character code of *char* is written, not its read syntax. **write-char**'s return value is unspecified. If *port* is omitted, the current output port is used.
- newline** Primitive procedure
newline *port* Primitive procedure
 Simply prints a newline character to *port*, thus ending the current output line. The same can be achieved by printing a newline character using the function **display**. **newline**'s return value is unspecified. If *port* is omitted, the current output port is used.
- call-with-input-string** *string proc* Primitive procedure
 Create an input string port which will return the contents of *string* and call *proc*, passing the newly created string port as the argument.
- call-with-output-string** *proc* Primitive procedure
 Call *proc* with a string port as the argument and return all data which was written to the string port while *proc* was executing as a string.
- with-input-from-string** *string thunk* Primitive procedure
 Call *thunk* with the current input port connected to a string port which delivers the contents of *string*.
- with-output-to-string** *thunk* Primitive procedure
 Call *thunk* and return a string consisting of all data which was written to the current output port during execution of *thunk*.
- with-error-to-string** *thunk* Primitive procedure
 Call *thunk* and return a string consisting of all data which was written to the current error port during execution of *thunk*.
- open-input-string** *string* Primitive procedure
 Takes *string* and returns 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** Primitive procedure
 Returns an output port that will accumulate characters for retrieval by **get-output-string**. The port can be closed by **close-output-port**, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.
- get-output-string** *output-port* Primitive procedure
 Given an output port created with **open-output-string**, returns a string consisting of the characters that have been output to the port so far.

with-input-from-port <i>port thunk</i>	Primitive procedure
Call <i>thunk</i> , but while <i>thunk</i> is evaluating, redirect all input from the input port <i>port</i> .	
with-output-to-port <i>port thunk</i>	Primitive procedure
Call <i>thunk</i> and redirect all output sent to the current output port to <i>port</i> while <i>thunk</i> is evaluating.	
with-error-to-port <i>port thunk</i>	Primitive procedure
Call <i>thunk</i> and redirect all output sent to the current error port to <i>port</i> while <i>thunk</i> is evaluating.	
port-line	Primitive procedure
port-line <i>port</i>	Primitive procedure
Return the current line number of the port <i>port</i> . If <i>port</i> is not specified, the current input port is used.	
set-port-line! <i>port n</i>	Primitive procedure
Set the current line number of the port <i>port</i> to the value of <i>n</i> , which must be an integer value. The return value is not specified.	
port-column	Primitive procedure
port-column <i>port</i>	Primitive procedure
Return the current line column number of the port <i>port</i> . If <i>port</i> is not specified, the current input port is used.	
set-port-column! <i>port n</i>	Primitive procedure
Set the current column number of the port <i>port</i> to the value of <i>n</i> , which must be an integer value. The return value is not specified.	
port-filename	Primitive procedure
port-filename <i>port</i>	Primitive procedure
Return the file name of the port <i>port</i> . If <i>port</i> is not specified, the current input port is used.	
set-port-filename! <i>port s</i>	Primitive procedure
Set the file name of the port <i>port</i> to the value of <i>s</i> , which must be a string value. The return value is not specified.	
isatty? [<i>port</i>]	Primitive procedure
Return #t , if <i>port</i> is a tty, #f otherwise. <i>port</i> defaults to the current input port.	
object->string <i>obj</i>	Primitive procedure
Return a string with the characters <code>display</code> would produce when printing <i>obj</i> .	

5.15 Regular expression functions

Sizzle supports Posix regular expressions. This section documents all functions provided to match text using regular expressions.

Note that Posix regular expressions only support matching of strings which do not include null bytes.

Regular expression operations are not available if the Sizzle library was configured with the `--disable-regexp` option or if no usable regular expression matching library was found on configuration time.

regexp? <i>obj</i>	Primitive procedure
Returns #t if <i>obj</i> is a compiled regular expression object, #f otherwise.	

make-regexp *string flags . . .* Primitive procedure

Create a compiled regular expression object which represents the string *string*. The following additional arguments can be passed to control regular expression compilation:

regexp/ignorecase

The matching is performed case-insensitively.

regexp/extended

Support extended Posix regular expressions (this is the default).

regexp/basic

Do not support extended Posix regular expressions.

regexp/newline

Treat a newline in the matched string as beginning/end of the string as far as the beginning-of-line and end-of-line operators are concerned.

regexp-exec *rx string [start [flags]]* Primitive procedure

Match the regular expression *rx* against *string*. If *start* is given, start matching at that position. Flags may be a mask of the following constants:

regexp/notbol

The match-beginning-of-line operator always fails to match.

regexp/noteol

The end-beginning-of-line operator always fails to match.

5.16 System interface functions

The Posix standard and many traditional Unix versions define a lot of functions which are useful for systems programming. Some of the most useful functions have been included into Sizzle, so it can be used for basic system programming tasks as well as all other uses.

Not all functionality documented in this section is available on all platforms. **fork**, for example cannot be implemented efficiently under Windows. Primitives, which do not function properly on the system Sizzle was compiled on, will throw a **not-available** exception when called.

Except when otherwise noted, procedures in this section are not available when Sizzle was configured with the **--disable-posix** option. You also need to perform a **(use-modules (core posix))** call to make these procedures available.

command-line Primitive procedure

Returns a list containing the command line arguments passed to the interpreter. Note that if you invoke a Sizzle script with the **-s SCRIPT** command line option, only the script name (as the first element in the command line) and the arguments after **SCRIPT** are contained in the command line list.

This procedure is always available.

system [*command*] Primitive procedure

Execute the command *command* using the command line shell and return the exit code of the given command. If no *command* is given, return **#t** if a command processor is available and **#f** otherwise. This function may raise a **system-error** exception if an error occurs. The return value is the exit code of the child process.

getenv *string* Primitive procedure

Returns the value of the environment variable called *string* if defined, and **#f** otherwise.

This procedure is always available.

- putenv** *string* Primitive procedure
Insert the string *string* into the process environment. *string* must be of the form *name=value*. The return value is not specified.
This procedure is only available if supported by the underlying C library.
- getcwd** Primitive procedure
Returns a string which denotes the current working directory.
This procedure is only available if supported by the underlying C library.
- chdir** *string* Primitive procedure
Change the current working directory to *string*. A **system-error** exception will be raised if the directory *string* cannot be changed to. The return value is not specified.
- getpid** Primitive procedure
Returns the process identifier (pid) of the running Sizzle process.
- getppid** Primitive procedure
Returns the parent process identifier of the running Sizzle process.
This procedure is only available if supported by the underlying C library.
- getuid** Primitive procedure
Returns the user identifier (uid) of the user running the current Sizzle process.
This procedure is only available if supported by the underlying C library.
- setuid** *uid* Primitive procedure
Set the user identifier of the current Sizzle process to *uid*. Only the superuser can do that. This function may raise a **system-error** exception if an error occurs. The return value is not specified.
This procedure is only available if supported by the underlying C library.
- geteuid** Primitive procedure
Returns the effective user identifier of the running Sizzle process. A **system-error** exception may be raised if an error occurs.
This procedure is only available if supported by the underlying C library.
- seteuid** *euid* Primitive procedure
Set the effective user identifier of the current Sizzle process to *euid*. This function may raise a **system-error** exception if an error occurs. The return value is not specified.
This procedure is only available if supported by the underlying C library.
- getgid** Primitive procedure
Returns the group identifier of the running Sizzle process. A **system-error** exception may be raised if an error occurs.
This procedure is only available if supported by the underlying C library.
- setgid** *gid* Primitive procedure
Set the group identifier of the running Sizzle process to *gid*. A **system-error** exception may be raised if an error occurs. The return value is not specified.
This procedure is only available if supported by the underlying C library.
- getegid** Primitive procedure
Returns the effective group identifier of the running Sizzle process. A **system-error** exception may be raised if an error occurs.
This procedure is only available if supported by the underlying C library.

setegid *egid* Primitive procedure
 Set the effective group identifier of the running Sizzle process to *egid*. A **system-error** exception may be raised if an error occurs. The return value is not specified.

This procedure is only available if supported by the underlying C library.

getpgid *pid* Primitive procedure
 Returns the process group identifier of the process specified by *pid*. If *pid* is zero, the process identifier of the current process is used. This function may raise a **system-error** exception if an error occurs.

This procedure is only available if supported by the underlying C library.

getpgrp Primitive procedure
 This is equivalent to (getpgid 0).

This procedure is only available if supported by the underlying C library.

getpwnam *name* Primitive procedure
 Returns a vector containing the broken out fields of a line from `‘/etc/passwd’` for the entry that matches the user name *name*. This function may raise a **system-error** exception if an error occurs. The returned vector contains the fields:

username	Login name
password	Crypted password or x if using shadow passwords
uid	User identifier
gid	group identifier
gecos	Full name, maybe additional information
dir	Login directory
shell	Login shell

This procedure is only available if supported by the underlying C library.

getpwuid *uid* Primitive procedure
 Returns a vector containing the broken out fields of a line from `‘/etc/passwd’` for the entry that matches the user identifier *uid*. This function may raise a **system-error** exception if an error occurs. The returned vector has the same format as for **getpwnam**.

This procedure is only available if supported by the underlying C library.

pw:name <i>vec</i>	Library procedure
pw:passwd <i>vec</i>	Library procedure
pw:uid <i>vec</i>	Library procedure
pw:gid <i>vec</i>	Library procedure
pw:gecos <i>vec</i>	Library procedure
pw:dir <i>vec</i>	Library procedure
pw:shell <i>vec</i>	Library procedure

Accessor procedures for the fields of the vector returned by **getpwnam** and **getpwuid**.

getgroups Primitive procedure
 Returns a vector containing all groups the current user belongs to. This function may raise a **system-error** exception if an error occurs.

This procedure is only available if supported by the underlying C library.

stat *filename* Primitive procedure

Returns information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file. This function may raise a **system-error** exception if an error occurs. The returned vector has the following fields:

dev	Device the file is located on.
ino	Inode number of the file.
mode	The file mode and permissions.
nlink	Number of hard links to the file.
uid	User identifier of the file owner.
gid	Group identifier of the file owner
rdev	Device type (if inode device)
size	Total size in bytes
atime	Time of last access
mtime	Time of last modification
ctime	Time of last change
blksize	Block size for the filesystem the file is on
blocks	Number of blocks the file allocates
type	The type of the file as a symbol, which may be any of regular , directory , symlink , block-special , char-special , fifo , socket or unknown .
perms	Permissions of the file. This is the file mode without the file type bits.

The file mode is a bitset which can be tested by logically combining it with one or more of the following constants:

S_ISUID	Set user identifier
S_ISGID	Set group identifier
S_IRWXU	User (file owner) has read, write and execute permission
S_IRUSR	User has read permission
S_IWUSR	User has write permission
S_IXUSR	User has execute permission
S_IRWXG	Group has read, write and execute permission
S_IRGRP	Group has read permission
S_IWGRP	Group has write permission
S_IXGRP	Group has execute permission
S_IRWXO	Others have read, write and execute permission
S_IROTH	Others have read permission
S_IWOTH	Others have write permission
S_IXOTH	Others have execute permission

This procedure is only available if supported by the underlying C library.

lstat *filename* Primitive procedure

Like **stat**, but does not follow symbolic links.

This procedure is only available if supported by the underlying C library.

stat:dev <i>vec</i>	Library procedure
stat:ino <i>vec</i>	Library procedure
stat:mode <i>vec</i>	Library procedure
stat:nlink <i>vec</i>	Library procedure
stat:uid <i>vec</i>	Library procedure
stat:gid <i>vec</i>	Library procedure
stat:rdev <i>vec</i>	Library procedure
stat:size <i>vec</i>	Library procedure
stat:atime <i>vec</i>	Library procedure
stat:mtime <i>vec</i>	Library procedure
stat:ctime <i>vec</i>	Library procedure
stat:blksize <i>vec</i>	Library procedure
stat:blocks <i>vec</i>	Library procedure
stat:type <i>vec</i>	Library procedure
stat:perms <i>vec</i>	Library procedure

These are accessor functions for extracting fields from the vectors returned by **stat** and **lstat**.

access? *filename mode* Primitive procedure

Checks whether the process would be allowed to read, write or test for existence of the file (or other file system object) whose name is *pathname*. If *pathname* is a symbolic link permissions of the file referred to by this symbolic link are tested. Returns **#t** if access is permitted, **#f** otherwise. This function may raise a **system-error** exception if an error occurs. *mode* is a mask of one or more of the following constants:

R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission
F_OK	Test for file existence

access? is only successful if all permissions are granted.

sys:pipe Primitive procedure

Creates a pair of file descriptors, pointing to a pipe inode. The car of the returned pair is a file descriptor for reading, the cdr is a file descriptor for writing. This function may raise a **system-error** exception if an error occurs.

This procedure is only available if supported by the underlying C library and under Windows.

open-fdes *filename flags [mode]* Primitive procedure

Attempts to open the file called *filename* and returns a file descriptor. *flags* is a mask of one or more of the following:

O_RDONLY	Open the file read only.
O_WRONLY	Open the file write only.
O_RDWR	Open the file for reading and writing.
O_CREAT	If the file does not exist it will be created.
O_EXCL	When used with O_CREAT, if the file already exists it is an error and the open will fail.

- O_NOCTTY** If *pathname* refers to a terminal device – see `tty(4)` – it will not become the process's controlling terminal even if the process does not have one.
- O_TRUNC** If the file already exists it will be truncated.
- O_APPEND** The file is opened in append mode. Initially, and before each write, the file pointer is positioned at the end of the file, as if with `lseek`.
- O_NONBLOCK**
Open the file in non-blocking mode, which means that operations on the file do not wait for completion.
- O_NDELAY** The same as **O_NONBLOCK**.
- O_SYNC** Writes to the file will be immediately written to disk.

If **O_CREAT** is given and the file will be created, the additional argument *mode* must be given, which denotes the file permission the new file will have. *mode* is a mask of the permission constants listed in the description of `stat`.

This function may raise a **system-error** exception if an error occurs.

sys:creat *mode* Primitive procedure
This is equivalent to `open` with flags equal to `(logior O_CREAT O_WRONLY O_TRUNC)`.

close *fd* Primitive procedure
Closes a file descriptor, if *fd* is an integer, so that it no longer refers to any file and may be reused, or simply closes *fd* if it is a port. This function may raise a **system-error** exception if an error occurs.

sys:write *fd u8vec count* Primitive procedure
Write *count* bytes from the homogenous unsigned 8-bit vector *u8vec* to the file referenced by the file descriptor *fd*. Returns the number of bytes actually written. This function may raise a **system-error** exception if an error occurs.
This procedure is only available if homogenous vectors were enable on configuration time.

sys:read *fd count* Primitive procedure
Read *count* bytes from the file referenced by the file descriptor *fd* and returns a homogenous unsigned 8-bit vector containing those bytes. This function may raise a **system-error** exception if an error occurs.
This procedure is only available if homogenous vectors were enable on configuration time.

sys:read! *fd u8vec count* Primitive procedure
Read *count* bytes from the file referenced by the file descriptor *fd* and stores them into the homogenous unsigned 8-bit vector *u8vec*. The return value is not specified. This function may raise a **system-error** exception if an error occurs.
This procedure is only available if homogenous vectors were enable on configuration time.

unlink *filename* Primitive procedure
Deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse. The return value is not specified. This function may raise a **system-error** exception if an error occurs.

- link** *oldpath newpath* Primitive procedure
 Creates a new link (also known as a hard link) to an existing file. If *newpath* exists it will not be overwritten. The return value is not specified. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- symlink** *oldpath newpath* Primitive procedure
 Creates a new symbolic link to an existing file. If *newpath* exists it will not be overwritten. The return value is not specified. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- rename** *oldpath newpath* Primitive procedure
 Renames a file, moving it between directories if required. *oldpath* and *newpath* must refer to the same file system. The return value is not specified. This function may raise a **system-error** exception if an error occurs.
- readlink** *filename* Primitive procedure
 Returns the name of the file the link *filename* is referring to. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- chmod** *path mode* Primitive procedure
 The mode of the file given by *path* is changed to *mode*. *mode* is a mask made from the **S_I...** constants listed in the description of **stat**. This function may raise a **system-error** exception if an error occurs.
- chown** *path uid gid* Primitive procedure
 Set the user identifier and group identifier of the file *path* to *uid* and *gid*. If any of *uid* or *gid* is equal to -1, that component of *path*'s ownership is not changed. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- primitive-exit** [*status*] Primitive procedure
 Terminate the current process without performing any cleanup. The exit status is *status*, which defaults to 0 if not specified. This procedure has no return value, because it never returns.
- fork** Primitive procedure
 Creates a child process that differs from the parent process only in its process identifier and parent process identifier, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- kill** *pid sig* Primitive procedure
 Sends any signal to any process group or process. If *pid* is positive, then signal *sig* is sent to *pid*.
 If *pid* equals 0, then *sig* is sent to every process in the process group of the current process. If *pid* equals -1, then *sig* is sent to every process except for the first one, from higher numbers in the process table to lower. If *pid* is less than -1, then *sig* is sent to every process in the process group *-pid*. If *sig* is 0, then no signal is sent, but error checking is still performed.
 This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.

- wait** Primitive procedure
 Waits for any child process to terminate and returns a pair whose car is the process id of the child and whose cdr is the exit code. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- waitpid** *pid* [*options*] Primitive procedure
 Waits until child process *pid* has terminated. For usage of the *options* argument, refer to the `waitpid` man page. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library and under Windows.
- sleep** *n* Primitive procedure
 Suspend the running process for *n* seconds, or until a signal wakes up the process. This function may raise a **system-error** exception if an error occurs.
- execl** *filename args* Primitive procedure
 Execute the file *filename*, passing the list *args* as command line options. This function may raise a **system-error** exception if an error occurs.
- execlp** *filename args* Primitive procedure
 Like `execl`, but search the path for *filename*.
- execle** *filename args env* Primitive procedure
 Like `execl`, but pass the environment list *env* together with the command line arguments *args* to the process.
- nice** *inc* Primitive procedure
 Change the current process priority by amount *inc*. You must be the system administrator to pass a negative *inc*. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- sync** Primitive procedure
 Force all dirty block from the cache of the operating system to be written to the disks. This function may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library.
- uname** Primitive procedure
 Return a vector containing information about the system. This function may raise a **system-error** exception if an error occurs. The returned vector contains the following fields:
- `sysname` The name of the operating system.
 - `nodename` The name of the host.
 - `release` The operating system release.
 - `version` The operating system version.
 - `machine` The machine architecture.
- This procedure is only available if supported by the underlying C library and under Windows.

utime *filename* [*timelist*] Primitive procedure

If *timelist* is omitted, set the access and modification time for the file *filename* to the current time. Otherwise, *timelist* must be a list of two integer values, which will be used to set the access and modification time of *filename*. This function may raise a `system-error` exception if an error occurs.

tmpnam Primitive procedure

The `tmpnam` procedure generates a unique temporary filename in the standard directory for temporary files (normally `'/tmp'`).

This procedure is only available if supported by the underlying C library.

tmpfile Primitive procedure

The `tmpfile` procedure generates a unique temporary filename in the standard directory for temporary files (normally `'/tmp'`). The temporary file is then opened in binary read/write (`w+b`) mode. The file will be automatically deleted when it is closed or the program terminates. The return value is a port open for input and output. This function may raise a `system-error` exception if an error occurs.

This procedure is only available if supported by the underlying C library.

umask [*mask*] Primitive procedure

`umask` sets the umask to (`logand mask #o777`). The umask is used by `open(2)` to set initial file permissions on a newly-created file. Specifically, permissions in the umask are turned off from the mode argument to `open(2)` (so, for example, the common umask default value of `022` results in new files being created with permissions `0666 & ~022 = 0644 = rw-r--r--` in the usual case where the mode is specified as `0666`). If *mask* is not given, the current umask value is returned, without changing the umask for the process.

strftime *format* *vec* Primitive procedure

Format the time-representing vector *vec* according to the format specifier *format* to a string and return that string. For details of the format specified, have a look at `man strftime`.

This procedure is only available if supported by the underlying C library.

strptime *format* *string* Primitive procedure

Parse the string *string* according to the format specifier *format*. Check out `man strptime` for the format specification. Returns a pair consisting of a time vector and the number of characters of *string* which have been used up when parsing. This procedure is essentially the counterpart of `strftime`.

This procedure is only available if supported by the underlying C library.

gmtime *x* Primitive procedure

Split up the time object *x*, which represents the current time in standard Unix convention in seconds since January 1, 1970. Returns a vector with the split up fields of the time. The times in the vector is in Coordinated Universal Time. The result vector contains the following fields:

seconds

minutes

hours

day of month

month

year

day of the week
 day in the year
 daylight saving time
 time zone offset
 time zone name or **#f** if not specified

localtime *x* Primitive procedure
 Returns *x* converted to local time in a vector of the format defined in the description of **gmtime**.

Bug: Right now time zone offset and time zone names are not correctly set by **localtime**.

tm:sec <i>timevec</i>	Library procedure
tm:min <i>timevec</i>	Library procedure
tm:hour <i>timevec</i>	Library procedure
tm:mday <i>timevec</i>	Library procedure
tm:month <i>timevec</i>	Library procedure
tm:year <i>timevec</i>	Library procedure
tm:wday <i>timevec</i>	Library procedure
tm:yday <i>timevec</i>	Library procedure
tm:isdst <i>timevec</i>	Library procedure
tm:zoff <i>timevec</i>	Library procedure
tm:zname <i>timevec</i>	Library procedure

These procedure return the fields their names stand for from the time vector *timevec*, which must have the format documented in the description for **gmtime**.

current-time Primitive procedure
 Return the current time in an integer, represented in the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). The value returned by this procedure can be used in calls to **gmtime** or **localtime**, which split up the time value into its various parts.

directory-files *dirname* Primitive procedure
 Return a list containing all files in the directory *dirname*. This procedure may throw a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library and under Windows.

fcntl *fdesc cmd [arg]* Primitive procedure
fcntl performs one of various operations on the file descriptor *fd*. The following commands are possible:

F_DUPFD	Makes <i>arg</i> be a copy of <i>fd</i> , closing <i>fdesc</i> first if necessary.
F_GETFD	Read the close-on-exec flag. If the low-order bit is 0, the file will remain open across exec, otherwise it will be closed.
F_SETFD	Set the close-on-exec flag to the value specified by <i>arg</i> (only the least significant bit is used).
F_GETFL	Read the descriptor's flags (all flags (as set by open(2)) are returned).
F_SETFL	Set the descriptor's flags to the value specified by <i>arg</i> . Only O_APPEND and O_NONBLOCK may be set. The flags are shared between copies (made with dup etc.) of the same file descriptor.

- F_GETLK, F_SETLK, F_SETLKW** Manage discretionary file locks. This command is not properly implemented in Sizzle right now.
- F_GETLK** This command is not properly implemented in Sizzle right now.
- F_SETLK** This command is not properly implemented in Sizzle right now.
- F_SETLKW** This command is not properly implemented in Sizzle right now.
- F_GETOWN** Get the process ID or process group currently receiving **SIGIO** and **SIGURG** signals for events on file descriptor *fd*. Process groups are returned as negative values.
- F_SETOWN** Set the process ID or process group that will receive **SIGIO** and **SIGURG** signals for events on file descriptor *fd*. Process groups are specified using negative values.

This procedure is only available if supported by the underlying C library and (limited) under Windows.

- fsync** *fdesc* Primitive procedure
fdesc must be either a valid file descriptor or a file port. All pending output is written if *fdesc* is a file port. The data buffered for the underlying file descriptor is then forced to be written to disk using the system call **fsync**.
 This procedure is only available if the **fsync()** system call is available and under Windows.
- copy-file** *oldfile newfile* Primitive procedure
 Copy the file specified by *oldfile* to *newfile*. The return value is unspecified.
- directory-stream?** *obj* Primitive procedure
 Return **#t** if *obj* is a directory stream, **#f** otherwise.
 This procedure is only available if supported by the underlying C library and under Windows.
- opendir** *str* Primitive procedure
 Open the directory called *str* and return a directory stream for that directory, which can be used in calls to **readdir**, **closedir** and **rewinddir**. This procedure may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library and under Windows.
- closedir** *stream* Primitive procedure
 Close the directory stream *stream*. The return value is not specified.
 This procedure is only available if supported by the underlying C library and under Windows.
- readdir** *stream* Primitive procedure
 Return the next filename from the directory stream *stream*, or the end-of-file object if all files have been read. This procedure may raise a **system-error** exception if an error occurs.
 This procedure is only available if supported by the underlying C library and under Windows.
- rewinddir** *stream* Primitive procedure
 Rewind the directory stream to the beginning, so that on the next call to **readdir**, the first file from the stream will be read. The return value is not specified.
 This procedure is only available if supported by the underlying C library and under Windows.

5.17 Network Procedures

The procedures in this section are useful for network programming. You have to use the module (`core net`) in order to make them visible.

network-error Exception
 This exception is thrown when an error occurs while using any of the network functions. The exception contains an error message obtained with `strerror(errno)` or `herror()`, whatever is appropriate for the specific error.

socket *domain type protocol* Primitive procedure
 Creates a socket port using *domain*, *type* and *protocol*. This is a direct wrapper to the C library function, so refer to the libc documentation for details. Normally you specify the *domain* as the constant `AF_INET` and the *type* `SOCK_STREAM` for normal internet sockets. This procedure may raise a `system-error` exception if an error occurs. The following socket types are also defined:

`SOCK_STREAM`
`SOCK_DGRAM`
`SOCK_RAW`
`SOCK_SEQPACKET`
`SOCK_RDM`

For the defined address family constants, refer to the documentation for `bind`.

shutdown *sock how* Primitive procedure
 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.

listen *sock backlog* Primitive procedure
 This procedure enables *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.

gethostbyname *name* Primitive procedure
 Returns the IPv4 address of the host named *name*. *name* may be either a string, which must represent a valid hostname, or an integer value. If it is an integer value, it is returned unchanged and not validated, if it is a string, the IP address is looked up and returned. This procedure may raise a `network-error` exception if an error occurs.

accept *sock* Primitive procedure
 Accepts a connection on a bound, listening socket *sock*. If there are no pending connections in the queue, it waits until one is available unless the non-blocking option has been set on the socket.

The return value is a pair in which the *car* is a new socket port for the connection and the *cdr* is an object with address information about the client which initiated the connection.

If the address is not available then the *cdr* will be an empty vector. *sock* does not become part of the connection and will continue to accept new requests.

- getsockname** *sock* Primitive procedure
Returns the address of *sock*, in the same form as the object returned by **accept**.
- getpeername** *sock* Primitive procedure
Returns the address of the socket *sock* is connected to, in the same form as the object returned by **accept**.
- ntohl** *n* Primitive procedure
Converts the 32 bit value *n* from network to host byte order.
- htonl** *n* Primitive procedure
Converts the 32 bit value *n* from host to network byte order.
- ntohs** *n* Primitive procedure
Converts the 16 bit value *n* from network to host byte order.
- htons** *n* Primitive procedure
Converts the 16 bit value *n* from host to network byte order.
- fileno** *port* Primitive procedure
Returns the file descriptor used for operations on *port*. *port* must be a file descriptor port, or an error will be signalled.
- connect** *hostname port* Primitive procedure
Establish a connection to the host identified by *hostname* on port *port*. *hostname* must be either a string naming the host or an integer value which represents an IP address. Currently, only IPv4 addresses are supported. Returns a socket port which can be written to and read from to communicate with the peer. This procedure may raise **network-error** and **system-error** exceptions.
- server-socket** *port* Primitive procedure
Returns a socket, which listens on the local IP address on port *port*. A port returned by this procedure may be used in calls to **accept**.
- server-loop** *port proc* Primitive procedure
This procedure can be used to implement simple network applications. *port* must be a port created by **server-socket** or any equivalent calling sequence to **socket**, **bind** and **listen**. **server-loop** will wait for incoming connection using **accept** and call *proc* when a connection is available. *proc* must expect two parameters, the first one being an input port connected to the accepted socket, the second an output port connected to the socket.
- bind** *scheme-port family address port* Primitive procedure
Bind the port *scheme-port*, created with a call to **socket** to the address specified by *family*, *address*, and the port number *port*. This procedure returns an unspecified value.
- The following address families are defined:
- AF_UNIX** Unix sockets family.
- AF_INET** Internet socket family.

If your system supports them, the following families are also defined:

AF_ISO

AF_NS

AF_IMPLINK

- select** *rdfd wrfd exfd sec usec* Primitive procedure
 Perform the **select** system call on the file descriptor sets *rdfd*, *wrfd* and *exfd*. The timeout value is calculated from the seconds *sec* and the microseconds *usec*. The number of the highest file descriptor in any of the sets is returned as the result.
- gethostname** Primitive procedure
 Get the host name of the local host. The returned value is a string.
- sethostname** *name* Primitive procedure
 Set the host name of the local host to *name*. The return value is not specified.
- getdomainname** Primitive procedure
 Get the domain name of the local host. The returned value is a string.
- setdomainname** Primitive procedure
 Set the domain name of the local host to *name*. The return value is not specified.
- inet-aton** *addr* Primitive procedure
 Convert the internet address *addr* from dotted decimal notation to binary data. The returned value is an integer.
- inet-ntoa** *addr* Primitive procedure
 Convert the internet address *addr* from binary representation to a string. The returned value is a string.

5.18 Module System

Scheme is suited for programming large programs, but it is very inconvenient to implement modules with restricted exports using standard techniques. Therefore, a module system similar to Guile's has been implemented.

In Sizzle, a module is a data structure with a private environment in which all definitions local to the module are stored and an export list which defines which of the defined symbols are to be exported to other modules. When a module is imported using the syntactic form **use-modules**, the symbol bindings from the export list are transferred to the environment of the importing module, thus making visible the exports. This works for procedures as well as for variables and constants.

The syntax and procedures for modules are always available.

- use-modules** *name0 name1 ...* Syntax
 Import all modules listed as *names*. If any module is not yet loaded, it will be loaded using the load path. Modules imported by any of the imported modules will be loaded recursively.
- define-module** *name options ...* Syntax
 Define a module called *name* and switch to the context of the new module, that means that all definitions made until the end of the file will be made in the private environment of the module. Definitions can be made public using the syntactic form **export**, documented below.
- Options may be one or more pair of a keyword and a value. The following keywords are presently defined:

#:use-module	The value must be a module name (a list of symbols) which specifies a module to be loaded for defining the new module.	
#:export	The value must be a list of symbols listing the defines to be exported. This is an alternative notation for the export syntactic form.	
export <i>symbol0 symbol1 ...</i>	Add all <i>symbols</i> to the export list of the currently active module.	Syntax
module-list	Return a list of currently loaded modules.	Primitive procedure
current-module	Return the module in which currently definitions take place.	Primitive procedure

5.19 Dynamic Loading

Sizzle support dynamic loading of shared libraries under Unix using the dl interface. The following procedures are not available if no usable dynamic loading support was found for the underlying platform or when disabled with the `--disable-dl` configuration option.

dl-error	This exception is thrown when an error occurs while using the 'dl' library. The exception contains an error message obtained with <code>dlerror()</code> .	Exception
load-library <i>name</i> [<i>filename</i>]	Load and initialize the shared object <i>name</i> . If <i>filename</i> is specified, the file named <i>filename</i> is <code>dlopen()</code> 'ed, otherwise the file name <code>libname.so</code> is opened via <code>dlopen()</code> . When the loaded object exports a symbol called <code>zzz_init_name()</code> , this function is called without arguments and should initialize the library. This function may throw a dl-error exception if the file could not be loaded or if the initialization function returned a value other than <code>RESULT_SUCCESS</code> .	Primitive procedure

Besides the high-level procedure `load-library`, some low-level procedures are provided which implement a wrapper around the 'dl' library.

%dlopen <i>filename</i> [<i>flags</i>]	<code>dlopen()</code> the file <i>filename</i> and return a dynamic library object for that file. <i>flags</i> may be the bitwise or of the following constants:	Primitive procedure
RTLD_GLOBAL	Make the symbols in the loaded file available to other processes.	
RTLD_LAZY	Relocate symbols when they are used.	
RTLD_NOW	Relocate all symbols when the library is loaded.	
%dlclose <i>lib</i>	Take a dynamic library object <i>lib</i> and close it. After that, calls to <code>%dlsym</code> with the library <i>lib</i> will fail.	Primitive procedure
%dlsym <i>lib string</i>	Return the address of the symbol <i>string</i> in the library specified by the dynamic library object <i>lib</i> . Throw a dl-error exception if the dymbol could not be resolved.	Primitive procedure
%dlcall <i>address</i>	Call the function specified by <i>address</i> (which must have been found with a call to <code>%dlsym</code>) without arguments. A dl-error exception is thrown if the function returns a value other than <code>RESULT_SUCCESS</code> .	Primitive procedure

5.20 Macro functions

Sizzle can handle macros defined using the primitive `define-macro`. The macro interface should be compatible to Guile's, but I have not yet verified that completely. The macro syntax and procedures are always available.

define-macro (*name arg ...*) *body* Syntax

Define a macro *name* which as many arguments as *args* are given. When the macro is expanded, the symbols in *args* are bound to the actual parameters of the macro call and *body* is evaluated. The result of evaluating the macro body expression is then substituted for the macro call. The return value is unspecified.

This is an example of a macro `writeln`, which prints all its arguments using `display` and then advances to the next line using `newline`.

```
(define-macro (writeln . --x)
  '(begin (for-each display (list ,@--x)) (newline)))
```

You have to be careful about the formal parameter names given with *args*, because they may shadow variables visible when the macro is called.

expand-macro *list* Primitive procedure

Expand the macro call represented by *list* and return the expanded expression. The car of the list must be a macro object.

```
zzz: (define-macro (unless cond . body)
      '(if (not ,cond) (begin ,@body)))
zzz: (expand-macro (list unless '(> x 0) '(display x)))
      (if (not (> x 0)) (begin (display x)))
```

apply-macro *macro list* Primitive procedure

Apply the macro *macro*, which must have been created with `define-macro` to the argument list *list*. This will cause the transformer procedure for *macro* to be applied to the arguments in *list*. The value returned by the macro transformer is returned.

5.21 Runtime system

It is often necessary to access the runtime system for writing efficient programs, but also for debugging purposes. The procedures in this section are provided for that task. They are always available in the interpreter.

garbage-collect Primitive procedure

Triggers immediate garbage collection. The return value is unspecified.

5.22 Self-Documentation

Sizzle provides a mechanism to make using it more comfortable. This is some kind of self-documentation as known from programs like Emacs. The principle of this feature is that every procedure object, both primitive and user-defined can have a documentation object attached. Normally, these objects are simply strings which tell how to use the procedure, which parameters are expected and what value(s) is/are returned, as well as a short description of the functionality.

The following procedures help to make use of the self-documentation system. They are always available in the interpreter.

documentation *obj* Primitive procedure

Returns the documentation object attached to *obj*. Returns `#f` if no such object is available. For procedure objects, `documentation` tries to load a documentation string from the file `'docstrings.txt'`, which was installed with Sizzle. Not all primitives are documented in this file yet, so `#f` may be returned for them, too.

- set-documentation!** *obj doc* Primitive procedure
 Attach the documentation object *doc* to the Scheme object *obj*. *doc* will be returned when applying **documentation** to *obj* after calling this procedure. The return value is unspecified.
- describe** *obj* Primitive procedure
 Print a description of *obj* to the current output port. This description includes the type of *obj* and the documentation object or documentation string attached to *obj*. The return value is unspecified.
- apropos** *string* Primitive procedure
 Print a one-line description of all procedures with *string* their names. This can be used to get a list of procedure names, so that **describe** can be used on the procedures to get more information about them. The return value of this procedure is not specified.

5.23 Box Operations

This section documents the procedures provided for handling boxes. For more information about boxes, refer to Section 3.3.19 [Boxes], page 16. All box primitives are always available in the interpreter.

- box?** *obj* Primitive procedure
 Return **#t** if *obj* is a box, **#f** otherwise.
- make-box** *obj* Primitive procedure
 Return a newly allocated box object which contains the object *obj*.
- box-ref** *b* Primitive procedure
 Return the object contained in the box object *b*.
- box-set!** *b obj* Primitive procedure
 Store the object *obj* into the box object *b*. Return an unspecified value.

5.24 Pointer Operations

Sizzle provides pointer objects which are useful when interfacing to C library procedures. For more information, have a look at Section 3.3.20 [Pointers], page 16. These procedures are always available.

- pointer?** *obj* Primitive procedure
 Return **#t** if *obj* is a pointer object, **#f** otherwise.
- make-pointer** *n* Primitive procedure
 Create a new pointer object. *n* must be a non-negative integer, and a memory block big enough to hold *n* bytes will be allocated and the pointer will point to it. If *n* is 0, then a null-pointer will be returned.
- invalidate-pointer!** *p* Primitive procedure
 Invalidate the pointer *p* by setting its *invalid* flag.

5.25 Misc functions

This section is a catch-all for all primitive procedures which do not fit into any of the other categories. They are always available in the interpreter except when otherwise noted.

gc-stats

Primitive procedure

Return an association list containing various information about the state of the garbage collector. The following elements are contained in the list:

used-cons-cells

Number of cons cells which are currently in use.

used-tagged-cells

Number of tagged cells which are currently in use.

allocated-cons-cells

Number of cons cells in the cons heap, both free cells and cells which are in use.

allocated-tagged-cells

Number of tagged cells in the tagged heap, both free cells and cells which are in use.

cons-heap-usage

Number of bytes currently allocated for the cons heap.

tagged-heap-usage

Number of bytes currently allocated for the tagged heap.

other-usage

Number of bytes currently allocated for other purposes, such as for vector and string contents etc.

gc-count Number of garbage collections since startup.

gc-time Time spent in the garbage collector since the interpreter started.

gc-percent

Amount of the total running time which was spent in the garbage collector since the interpreter started up.

profile-stats

Primitive procedure

Return an association list containing various information about internal profiling statistics. The values in the returned association list are only valid if profiling was enabled during compilation. The following elements are contained in the list:

global-accesses

Variable accesses which required a complete variable lookup.

lloc-accesses

Variable access which went through an `lloc` object.

gloc-accesses

Variable access which went through a `gloc` object.

tail-calls

Procedure calls which were executed tail-recursively.

evaluate-calls

Calls to the function `zzz_evaluate()`.

closure-calls

Calls to closure objects (lambda applications).

primitive-calls

Calls to primitive procedures.

syntactic-calls

Executions of non-memoized syntactic forms.

immediate-calls

Executions of memoized syntactic forms.

random *n* Primitive procedure
 Return a random number (an integer value) which is in the range $[0..n)$. **random** returns always the same sequence of random number, unless you have set the random seed using **srand**.

srand *n* Primitive procedure
 Set the random seed for the procedure **random** to *n*. The return value is not specified.

1+ *x* Library procedure

1- *x* Library procedure

add1 *x* Library procedure

sub1 *x* Library procedure

Handy helper functions. **1+** and **add1** return their argument plus 1, **1-** and **sub1** return their argument minus 1. These functions are often used in textbooks on Scheme, so I put them in.

These procedures are not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

pretty-print *obj* [*port*] Library procedure
 Pretty-print the Scheme object *obj* to *port*, or to the standard output port if *port* is omitted. Pretty-printing means that list and vector structures will be printed in a human-readable form and with proper indentation.

You have to use the module (**core pprint**) to have access to this procedure.

5.26 Syntactic Forms

There are two types of syntactic forms. The forms labelled with *Syntax* are builtin syntax, the forms labelled *Defined syntax* or *Macro* are defined in the startup file `'init.scm'` and may be not present if the embedding application does not load this init file.

Except when otherwise stated, all of these syntactic forms and procedures are always available in the interpreter.

quote *obj* Syntax
 Returns its argument *obj* unevaluated. This is normally used to type in symbols and is used so often that a shorthand form exists: Instead of `'(quote foo)`' you can write `'foo`'.

quasiquote *obj* Syntax
 Works similar to **quote**, with the difference that expressions with **unquote** or **unquote-splicing** are evaluated in *obj*. Also has a shorthand form: `'foo`' has the same meaning `'(quasiquote foo)`'.

unquote *obj* Syntax
 When used inside of a **quasiquote** expression, evaluates its argument. Also has a shorthand form: `',foo`' has the same meaning `'(unquote foo)`'. **unquote** may only be used inside of **quasiquote** expression, an error is signalled if used in another context.

unquote-splicing *obj* Syntax

When used inside of a `quasiquote` expression, evaluates its argument. The difference between `unquote` and `unquote-splicing` is that the latter requires its argument to evaluate to a list which is then inserted at the current position with one level of nesting removed. Also has a shorthand form: `’,@foo’` has the same meaning `’(unquote-splicing foo)’`. `unquote-splicing` may only be used inside of `quasiquote` expression, an error is signalled if used in another context.

set! *variable expression* Syntax

Set the value of *variable* to the value of *expression*. The return value is not specified. You can only use `set!` on variables which have been already defined by either `define` or one of the `let`-constructs.

define *variable expression* Syntax

Define *variable* as a variable and set its value to the value of *expression*. The return value is not specified.

define (*variable arg1 ...*) *commands ...* Syntax

define (*variable arg1 argn*) *commands ...* Syntax

define (*variable . arg*) *commands ...* Syntax

Special form of `define` which is equivalent to

```
(define variable (lambda (arg1 ...) commands ...
```

```
(define variable (lambda (arg1 ... . argn) commands ...
```

```
(define variable (lambda arg commands ...
```

undefine *symbol* Syntax

Make the symbol *symbol* undefined. After calling this special form, any references to *symbol* will cause an `unbound-variable` error. The return value is not specified.

defined? *symbol* Primitive procedure

This is not a special form, but it fits best into this section together with `define` and `undefine`. `defined?` returns `#t` if *symbol* is a defined variable and `#f` otherwise.

define-constant *variable expression* Syntax

Define *variable* as a variable and set its value to the value of *expression*. The return value is not specified. The difference between `define` and `define-constant` is that variables created using `define-constant` are read-only and cannot be modified with `set!`. This enables runtime optimizations because references to constant variables can be replaced by the variable’s value, eliminating symbol lookups.

if *condition then [else]* Syntax

Evaluates *condition* and then checks its return value. If the value is `#t`, *then* is evaluated and the result of that evaluation is returned. Otherwise, the behaviour depends on the presence of the *else* expression. If *else* is not provided, the return value is unspecified, otherwise *else* is evaluated and the result is returned.

cond *clause1 clause2 ...* Syntax

Each *clause* is of the form

```
(test expression1 ...)
```

Alternatively, a *clause* may be of the form

```
(test => expression)
```

and the last *clause* may be an *else clause*, which has the form

```
(else expression1 expression2 ...)
```

A `cond` expression is evaluated by evaluating the *test* expressions of successive *clauses* in order until one of them evaluates to a true values. The first *clause* whose *test* evaluates to true is chosen and the *expressions* in the clause are evaluated in order and the value of the last expression is returned as the value of the `cond` expression. If the selected *clause* has no *expressions*, the value of the *test* is returned. An *else clause* is chosen and the following *expressions* are evaluated if no other *test* evaluates to true.

If the selected *clause* has the `=>` form, the *expression* following the arrow must evaluate to a procedure to which the result of the *test* is applied to yield the result of the `cond` expression.

case *key clause1 clause2 ...* Syntax
key may be any expression. Each *clause* should have the form

```
((datum1 ...) expression1 expression2 ...)
```

where each datum is an external representation of some object. The last *clause* may be of the form

```
(else expression1 expression2 ...)
```

A `case` expression is evaluated by first evaluating the *key*. The resulting value is compared against each of the *datums* using the equality predicate `eqv?`, and if a match is found, the *expression* following the matching *datum* is evaluated and the result of the last *expression* is returned as the result of the `case` expression.

Should no *clause* match and an `else` clause is present, that clause is chosen and the *expressions* following the `else` are evaluated in order to compute the result of the `case` expression.

do ((*variable1 init1 step1*) ...) (*test expression ...*) *command ...* Syntax

`do` is an iteration construct. First, all *variables* are bound to the values of their corresponding *init* values. Then the *test* expression is evaluated and if the value is false, the *commands* are evaluated in order. After each iteration, the *variables* are bound to the result of evaluating the corresponding *step* expressions and the *test* is evaluated again. The loop terminates as soon as the *test* evaluates to a true value. Then the *expressions* after the *test* are evaluated in order and the value of the last *expression* is returned as the value of the `do` expression.

begin *command ...* Syntax

`begin` accepts one or more expressions as its arguments. All expressions are evaluated in the order they appear in the argument list and the result of the last expression is returned. The result is unspecified if no *command* is given.

let ((*variable expression*) ...) *command ...* Syntax

`let` creates a block in which all variables named in the first argument list are redefined to the corresponding values. The expressions in the body are executed and afterwards all variables are rebound to their original values. The result of the last evaluated expression is returned. If no body expressions are given, the result is unspecified. When using `let`, the *variables* are not visible to *expressions*, because the variables are bound after all the expressions are evaluated.

let *variable* ((*variable1 expression*) ...) *command ...* Syntax

The named-let construct is used to implement looping constructs. *variable* is bound to a lambda expression which takes the *variables* as its formal parameters and then calls the expression with actual parameters set to the *expressions*. The body of the `let` can then refer to *variable* by calling it recursively as a function.

let* ((*variable expression*) ...) *command* ... Syntax
 Works like **let**, but the *expressions* are evaluated in left-to-right order where each *variable* is visible to the expression on its right.

letrec ((*variable expression*) ...) *command* ... Syntax
 Works like **let**, but the *variables* are visible to all *expressions*, thus making possible mutual recursive procedures.

lambda (*variable1* ...) *command* ... Syntax

lambda (*variable1* *variablen*) *command* ... Syntax

lambda *variable* *command* ... Syntax

Lambda is used to define procedures. The *variables* are the formal parameters to which actual parameters will be bound. When the procedure is called, the actual parameters are bound and the *commands* are evaluated in order. The three syntactic forms above differ in the way parameters are bound. The first lambda takes exactly as many actual arguments as *variables* are given, the second takes at least as many actual arguments as *arguments* before the dot are given and binds *variablen* to a list of the rest of the arguments and the third form takes any number of arguments which will then be bound to *variable*.

and *expression* ... Syntax
and evaluates all *expressions* in order, but stops evaluation as soon as one of the *expressions* returns a false value. Returns **#f** if any of the expressions returns **#f** and returns the result of the last expression otherwise. When called without arguments, **#t** is returned.

or *expression* ... Syntax
or evaluates all *expressions* in order, but stops evaluation as soon as one of the *expressions* returns a non-false value. Returns **#f** if none of the expressions returns **#t** and returns the result of the first expression returning a non-false value otherwise. When called without arguments, **#f** is returned.

delay *expression* Syntax
 Delays the evaluation of *expression*. Returns a promise, which when later forced by a call to **force** will evaluate *expression* and return the result. Evaluation is performed in the same environment in which the promise was created.

when *test body* ... Macro
 Evaluates all expressions in *body*, but only if *test* evaluates to a true value. If called without arguments, **#f** is returned, if only called with a *test* argument, the value of the *test* is returned, otherwise the value of the last expression in *body* is returned.
 This macro is not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

unless *test body* ... Macro
 Evaluates all expressions in *body*, but only if *test* evaluates to **#f**. If called without arguments, **#f** is returned, if only called with a *test* argument, the value of the *test* is returned, otherwise the value of the last expression in *body* is returned.
 This macro is not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

false-if-exception *expr* Macro
 Evaluates *expr* and returns the result. If an exception occurs while evaluating *expr*, it is caught and **#f** is returned.
 This macro is not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

while *test body . . .* Macro

Evaluates the *body* expressions as long as *test* evaluates to a true value. *test* is always evaluated before the *body* expressions. Returns **#f** if no call to **break** occurs inside of *body*, otherwise the value given to **break** is returned.

This macro is not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

break *val* Library procedure

This is not a syntactic form, but is closely related to **while**, so it was put in this section. **break** is only defined inside of a **while** body. When called, it aborts the innermost **while** loop and lets it return *val*.

This macro is not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

continue Library procedure

Like **break**, this is not a syntactic form, but is closely related to **while**, so it was put in this section. **continue** is only defined inside of a **while** body. When called, it restarts the innermost **while** loop, starting with the test of the **while** condition.

This macro is not supported when the option **-n, --no-system-init-file** is passed to the interpreter, or if the system startup file is not found.

Sizzle vs. R5RS

Sizzle is not a complete implementation of Scheme as defined in the Revised 5 Report on the Algorithmic Language Scheme (R5RS), but most of the standard procedures from that report and all data types except complex and rational numbers are supported. This chapter lists all differences between Sizzle and standard Scheme.

First of all, Sizzle does only support non-hygienic macros at all, and the high-level syntax definitions from R5RS are implemented a little bit shaky. `call-with-current-continuation` only works in upward direction properly, general continuations can be compiled in, but are buggy and will segfault in the current state. Besides of that, Sizzle is a quite useful language for embedding and higher-order functional programming purposes.

Standard Data Types

The following data types are implementd in Sizzle, including most standard procedures for manipulating objects of these types.

Whole numbers

Real numbers

Lists

Vectors

Strings

Characters

Boolean values

The numerical tower is not fully implemented, not available are the number types:

Rational numbers

Complex numbers

Standard Procedures

These are the standard procedures defined in Sizzle.

Supported Syntax

quote, quasiquote, unquote, unquote-splicing, lambda, if, cond, case, set!, and, or, do, let, let*, letrec, begin, define

Supported Procedures

eqv?, eq?, equal?, number?, complex?, real?, rational?, integer?, exact?, inexact?, zero?, positive?, negative?, odd?, even?, max, min, +, *, -, /, abs, quotient, remainder, modulo, floor, ceiling, truncate, round, exp, log, sin, cos, tan, asin, acos, atan, sqrt, expt, exact->inexact, inexact->exact, number->string, string->number, gcd, lcm, not, boolean?, pair?, cons, car, cdr, set-car!, set-cdr!, caar, cadr, . . ., caddr, caddr, null?, list?, list, length, append, reverse, list-tail, list-ref, memq, memv, member, assq, assv, assoc, symbol?, symbol->string, string->symbol, char?, char=?, char<?, char>?, char<=?, char>=?, char-ci=?, char-ci<?, char-ci>?, char-ci<=?, char-ci>=?, char-alphabetic?, char-numeric?, char-whitespace?, char-upper-case?, char-lower-case?, char->integer, integer->char, char-upcase char, char-downcase, string?, make-string, string-length, string-ref, string-set!, string=?, string<?, string>?, string<=?, string>=?, string-ci=?, string-ci<?, string-ci>?, string-ci<=?, string-ci>=?, substring,

string-append, string->list, list->string, string-copy, string->fill!, vector?, make-vector, make-vector, vector, vector-length, vector-ref, vector-set!, vector->list, list->vector, vector-fill!, procedure?, apply, map, for-each, eval, interaction-environment, call-with-input-file, call-with-output-file, with-input-from-file, with-output-to-file, input-port?, output-port?, current-input-port, current-output-port, open-input-file, open-output-file, close-input-port, close-output-port, read, read-char, peek-char, write-char, eof-object?, write, display, newline, delay, force, values, call-with-values, dynamic-wind, transcript-on, transcript-off, null-environment, scheme-report-environment, char-ready?

Partly implemented

load Implemented in the startup file 'init.scm', but Sizzle has a builtin primitive procedure `primitive-load`, which works like `load` as defined in R5RS. The Scheme procedure `load` in Sizzle searches all directories listed in the variable `%load-path` as well as the current directory if the given filename is not absolute.

real-part imag-part magnitude angle

These work for integers and reals only, because complex numbers are not supported.

denominator numerator

These work for integers only, because rational numbers are not supported.

call-with-current-continuation

Only for escape procedures.

Not supported

let-syntax letrec-syntax syntax-rules define-syntax, rationalize, make-rectangular, make-polar

Glossary

application

When a function gets called with actual arguments, we say that the function is applied to the arguments, or that it is an application of the function to the arguments.

boolean Data type. Boolean values can be either `#t` (true) or `#f` (false).

box Data type. Can store another Scheme object and can be used to implement call-by-reference procedures.

catch Exceptions can be handled by catching them using the procedure `catch`.

character Data type. Characters are used to construct strings and are written like this: `#\t`.

cons cell Heap cell with one arbitrary value in both `car` and `cdr`.

dotted pair

A dotted pair is a pair of values, stored in the `car` and `cdr` fields of a cons cell. The read syntax for dotted pairs looks like this: `(a . b)`.

environment

An environment is a mapping from symbols to values. A symbol that is evaluated in a particular environment has its value retrieved from the environment.

error An error is signalled when an evaluation cannot be performed correctly.

exception An exception is thrown when an evaluation cannot be performed correctly. The difference between errors and exceptions is that exceptions can be handled using the `catch` form.

fixnum An integer value which can be stored in a tagged pointer. 29-bit two-complement value.

heap Memory area where all non-immediate values are allocated.

heap cell Cell on the heap of 8 bytes size which holds either two values (cons cell) or a type tag and additional data (tagged cell).

integer Data type. Numeric type which can hold exact integer values.

list Data type. A list is made out of chained cons cells.

long int Integer that does not fit into a fixnum. Value range is the same as a C `int` variable.

multiple values

Scheme supports returning multiple values from procedures. Multiple values are produced using the syntactic form *values*.

pointer Data type. Used to wrap machine pointers.

port Object from which data can be read or to which data can be written. File ports or string ports for example.

real number

A number with (possibly) infinite decimal places like `pi` or `e`.

regexp Short for regular expression.

string Data type. A string is a sequence of characters.

symbol Data type. Symbols are values which can have a value attached.

syntactic form

Element of the syntax which looks like a function call but acts differently. The `if` or `lambda` forms are examples of syntactic forms.

tagged cell

16 byte heap cell with a type tag in its first word and a three data pointers to additional data in the other words.

thunk

A thunk is a procedure without arguments.

vector

Data type. Used to hold a fixed number of other values with constant access time.

Index

!		<	
!=	20	<	20
		<=	20
%		1	
%dcall	69	1-	73
%dclose	69	1+	73
%dlopen	69		
%dlsym	69	A	
%eval-time-taken%	18	abs	21
%load-path	17	accept	66
%max-evaluate-stack%	17	access?	59
%print-backtrace%	18	acos	22
%print-backtrace-limit%	18	add1	73
%print-func-bodies%	18	alist-cons	32
%print-memoized%	17	alist-copy	32
%print-read%	17	alist-delete	32
%print-result%	17	alist-delete!	32
%read-delimited!	52	and	76
%save-backtrace%	18	angle	21
%sizzle-major-version%	17	any	31
%sizzle-minor-version%	17	append	27
%sizzle-patchlevel%	17	append!	27
%sizzle-version%	17	append-map	30
%write-history-file%	17	append-map!	30
		append-reverse	27
*		append-reverse!	27
*	20	apply	46
		apply-macro	70
-		apropos	71
-	21	ash	23
		asin	22
.		assoc	32
.	25	assq	32
		assv	32
/		atan	22
/	21	B	
		backtrace	49
=		basename	42
=	20	begin	75
		bind	67
+		boolean->integer	23
+	20	boolean?	23
		box-ref	71
>		box-set!	71
>	20	box?	71
>=	20	break	31, 77
		break!	31

C

caar.....	25
cadr.....	25
call-with-current-continuation.....	48
call-with-input-file.....	50
call-with-input-string.....	53
call-with-output-file.....	50
call-with-output-string.....	53
call-with-values.....	46
call/cc.....	48
car.....	25
car+cdr.....	26
case.....	75
catch.....	47
cdddar.....	25
cddddr.....	25
cdr.....	25
ceiling.....	21
char->integer.....	36
char-alphabetic?.....	36
char-ci=.....	36
char-ci>.....	36
char-ci>=.....	36
char-ci<.....	36
char-ci<=.....	36
char-downcase.....	37
char-lower-case?.....	36
char-numeric?.....	36
char-ready?.....	52
char-upcase.....	37
char-upper-case?.....	36
char-whitespace?.....	36
char=.....	36
char?.....	35
char>.....	36
char>=.....	36
char<.....	36
char<=.....	36
chdir.....	56
chmod.....	61
chown.....	61
circular-list.....	24
circular-list?.....	25
cleanup-filename.....	43
close.....	60
close-input-port.....	51
close-output-port.....	51
close-port.....	51
closedir.....	65
closure?.....	45
command-line.....	55
complex?.....	19
concatenate.....	27

concatenate!.....	27
cond.....	74
connect.....	67
cons.....	24
cons*.....	24
continue.....	77
copy-file.....	65
cos.....	22
count.....	28
current-environment.....	49
current-error-port.....	50
current-input-port.....	50
current-module.....	69
current-output-port.....	50
current-time.....	64

D

define.....	74
define-constant.....	74
define-macro.....	70
define-module.....	68
defined?.....	74
delay.....	76
delete.....	33
delete!.....	33
delete-duplicates.....	33
delete-duplicates!.....	33
denominator.....	21
describe.....	71
directory-files.....	64
directory-stream?.....	65
dirname.....	43
display.....	53
dl-error.....	69
do.....	75
documentation.....	70
dotted-list?.....	25
drop.....	26
drop-right.....	26
drop-right!.....	26
drop-while.....	31
dynamic-wind.....	46

E

eighth.....	26
eof-object?.....	52
eq?.....	19
equal?.....	19
equiv?.....	19
error.....	47
eval.....	49
even?.....	20
every.....	32

exact->inexact	22
exact?	20
execl	62
execle	62
execlp	62
exit	48
exp	22
expand-macro	70
export	69
expt	22

F

false-if-exception	76
fcntl	64
fifth	26
fileno	67
filter	30
filter!	30
filter-map	30
find	31
find-tail	31
first	26
floor	21
fold	28
for-each	30
force	46
force-output	51
fork	61
form?	46
fourth	26
fsync	65
ftell	52

G

garbage-collect	70
gc-message	17
gc-stats	72
gcd	23
gensym	35
get-output-string	53
getcwd	56
getdomainname	68
getegid	56
getenv	55
geteuid	56
getgid	56
getgroups	57
gethostbyname	66
gethostname	68
getpeername	67
getpgid	57
getpgrp	57
getpid	56

getppid	56
getpwnam	57
getpwuid	57
getsockname	67
getuid	56
gmtime	63

H

hash	34
hash-create-handle!	35
hash-fold	35
hash-get-handle	34
hash-ref	35
hash-remove!	35
hash-set!	35
hashq	34
hashq-create-handle!	35
hashq-get-handle	34
hashq-ref	35
hashq-remove!	35
hashq-set!	35
hashv	34
hashv-create-handle!	35
hashv-get-handle	34
hashv-ref	35
hashv-remove!	35
hashv-set!	35
htonl	67
htons	67

I

if	74
imag-part	21
inet-aton	68
inet-ntoa	68
inexact->exact	22
inexact?	20
input-port?	50
integer->boolean	23
integer->char	36
integer?	19
interaction-environment	49
invalidate-pointer!	71
iota	24
isatty?	54

K

keyword-table	50
kill	61

L

lambda	76
last	27

last-pair	27
lcm	23
length	27
length+	27
let	75
let*	76
letrec	76
link	61
list	24
list->string	37
list->TAGvector	45
list->vector	44
list-copy	24
list-index	32
list-ref	26
list-tabulate	24
list-tail	26
list=	25
list?	24
listen	66
load	48
load-library	69
localtime	64
log	22
logand	23
logior	23
lognot	22
logxor	23
long?	20
lset-adjoin	33
lset-diff+intersection	34
lset-diff+intersection!	34
lset-difference	34
lset-difference!	34
lset-intersection	34
lset-intersection!	34
lset-union	34
lset-union!	34
lset-xor	34
lset-xor!	34
lset=	33
lset<=	33
lsh	23
lstat	59

M

magnitude	21
make-box	71
make-hash	34
make-list	24
make-pointer	71
make-regexp	55
make-string	37

make-TAGvector	45
make-vector	43
map	30
map!	30
map-in-order	30
max	20
member	32
memq	32
memv	32
min	20
module-list	69
modulo	21

N

negative?	20
network-error	66
newline	53
nice	62
ninth	26
not	23
not-pair?	25
ntohl	67
ntohs	67
null-environment	49
null-list?	25
null?	25
number->string	22
number?	19
numerator	21

O

object->string	54
odd?	20
open-fdes	59
open-input-file	51
open-input-string	53
open-output-file	51
open-output-string	53
opendir	65
or	76
output-port?	50

P

pair-fold	28
pair-for-each	30
pair?	25
partition	31
partition!	31
peek-char	51
pointer?	71
port-closed?	52
port-column	54
port-filename	54

<code>string->number</code>	22	<code>string-upcase</code>	41
<code>string->s8vector</code>	45	<code>string-upcase!</code>	41
<code>string->symbol</code>	35	<code>string=</code>	38
<code>string->u8vector</code>	45	<code>string?</code>	37
<code>string-any</code>	37	<code>string>=?</code>	39
<code>string-append</code>	37	<code>string>?</code>	39
<code>string-chop</code>	42	<code>string<=?</code>	38
<code>string-ci=?</code>	39	<code>string<?</code>	38
<code>string-ci>=?</code>	39	<code>strptime</code>	63
<code>string-ci>?</code>	39	<code>sub1</code>	73
<code>string-ci<=?</code>	39	<code>substring</code>	38
<code>string-ci<?</code>	39	<code>symbol->string</code>	35
<code>string-contains</code>	40	<code>symbol-table</code>	50
<code>string-contains-ci</code>	40	<code>symbol?</code>	35
<code>string-copy</code>	38	<code>symlink</code>	61
<code>string-count</code>	40	<code>sync</code>	62
<code>string-downcase</code>	41	<code>sys:creat</code>	60
<code>string-downcase!</code>	41	<code>sys:pipe</code>	59
<code>string-drop</code>	41	<code>sys:read</code>	60
<code>string-drop-right</code>	41	<code>sys:read!</code>	60
<code>string-every</code>	37	<code>sys:write</code>	60
<code>string-fill!</code>	38	<code>system</code>	55
<code>string-hash</code>	41		
<code>string-hash-ci</code>	41	T	
<code>string-index</code>	40	<code>TAGvector</code>	45
<code>string-index-right</code>	40	<code>TAGvector->list</code>	45
<code>string-join</code>	38	<code>TAGvector-fill!!</code>	45
<code>string-length</code>	38	<code>TAGvector-length</code>	45
<code>string-null?</code>	37	<code>TAGvector-ref</code>	45
<code>string-pad</code>	42	<code>TAGvector-set!</code>	45
<code>string-pad-right</code>	42	<code>TAGvector?</code>	45
<code>string-prefix-ci?</code>	39	<code>take</code>	26
<code>string-prefix-length</code>	39	<code>take!</code>	26
<code>string-prefix-length-ci</code>	39	<code>take-right</code>	26
<code>string-prefix?</code>	39	<code>take-while</code>	31
<code>string-ref</code>	38	<code>take-while!</code>	31
<code>string-reverse</code>	38	<code>tan</code>	22
<code>string-reverse!</code>	38	<code>tenth</code>	26
<code>string-set!</code>	38	<code>third</code>	26
<code>string-skip</code>	40	<code>throw</code>	47
<code>string-skip-right</code>	40	<code>tilde-expand</code>	43
<code>string-split</code>	41	<code>tm:hour</code>	64
<code>string-suffix-ci?</code>	39	<code>tm:isdst</code>	64
<code>string-suffix-length</code>	39	<code>tm:mday</code>	64
<code>string-suffix-length-ci</code>	39	<code>tm:min</code>	64
<code>string-suffix?</code>	39	<code>tm:month</code>	64
<code>string-tabulate</code>	38	<code>tm:sec</code>	64
<code>string-take</code>	41	<code>tm:wday</code>	64
<code>string-take-right</code>	41	<code>tm:yday</code>	64
<code>string-titlecase</code>	42	<code>tm:year</code>	64
<code>string-titlecase!</code>	41	<code>tm:zname</code>	64
<code>string-trim</code>	42	<code>tm:zoff</code>	64
<code>string-trim-both</code>	42	<code>tmpfile</code>	63
<code>string-trim-right</code>	42		

tmpnam 63
 toplevel 48
 trace 49
 transcript-off 48
 transcript-on 48
 trim-whitespace 42
 truncate 21

U

u8vector->string 45
 umask 63
 uname 62
 undefine 74
 unfold 29
 unfold-right 29
 unless 76
 unlink 60
 unquote 73
 unquote-splicing 74
 unread-char 51
 unzip1 28
 unzip2 28
 unzip3 28
 unzip4 28
 unzip5 28
 use-modules 68
 utime 63

V

values 46
 vector 43
 vector->list 44
 vector-copy 43

vector-fill! 44
 vector-length 44
 vector-ref 44
 vector-set! 44
 vector-tabulate 43
 vector= 44
 vector? 44

W

wait 62
 waitpid 62
 when 76
 while 77
 with-error-to-file 51
 with-error-to-port 54
 with-error-to-string 53
 with-input-from-file 51
 with-input-from-port 54
 with-input-from-string 53
 with-output-to-file 51
 with-output-to-port 54
 with-output-to-string 53
 write 53
 write-char 53
 write-line 52

X

xcons 24

Z

zero? 20
 zip 28

Table of Contents

1	Introduction	1
1.1	Sizzle History	1
1.2	Sizzle Future	1
2	Using Sizzle	2
2.1	Starting Sizzle	2
2.2	Interactive Usage	3
2.3	Automatic Interactive Variable	4
2.4	Command Line Editing	4
2.5	Sizzle Scripts	5
2.6	Startup Sequence	5
2.7	Environment Variables	5
2.8	Building Sizzle	6
3	Programming Scheme	8
3.1	General	8
3.2	Storage Model	8
3.3	Data Types	9
3.3.1	Numbers	9
3.3.2	Booleans	10
3.3.3	Characters	10
3.3.4	Strings	10
3.3.5	Symbols	11
3.3.6	Keywords	11
3.3.7	Cons Cells	11
3.3.8	Vectors	12
3.3.9	Homogenous numeric vectors	12
3.3.10	Hash tables	13
3.3.11	Procedures	13
3.3.12	Regular Expressions	14
3.3.13	Multiple Values	14
3.3.14	Special values	14
3.3.15	Constants	14
3.3.16	Ports	15
3.3.17	Errors and Exceptions	15
3.3.18	Continuations	15
3.3.19	Boxes	16
3.3.20	Pointers	16
4	Predefined Variables	17
5	Builtin Procedures	19
5.1	Equality tests	19
5.2	Numerical operations	19
5.3	Boolean operations	23
5.4	Pairs and lists	23
5.4.1	List constructors	24
5.4.2	List predicates	24

5.4.3	List selectors	25
5.4.4	Miscellaneous list procedures	27
5.4.5	Fold, unfold and map	28
5.4.6	Filtering and partitioning	30
5.4.7	List searching	31
5.4.8	Association lists	32
5.4.9	Deletion	33
5.4.10	Primitive side-effects	33
5.4.11	Set operations on lists	33
5.5	Hash tables	34
5.6	Symbol operations	35
5.7	Character operations	35
5.8	String operations	37
5.8.1	String Predicates	37
5.8.2	String Constructors	37
5.8.3	String Accessors	38
5.8.4	String Comparison	38
5.8.5	String Searching	40
5.8.6	String Decomposition	41
5.8.7	String Case Conversion	41
5.8.8	String Trimming	42
5.8.9	String Padding	42
5.8.10	String Pathname Operations	42
5.9	Vector operations	43
5.9.1	Vector constructors	43
5.9.2	Vector predicates	44
5.9.3	Vector selectors	44
5.9.4	Vector conversion	44
5.9.5	Vector modifying	44
5.10	Homogenous Vector Operations	44
5.11	Control flow operations	45
5.12	Eval function	49
5.13	Debugging Support	49
5.14	Input and output	50
5.15	Regular expression functions	54
5.16	System interface functions	55
5.17	Network Procedures	66
5.18	Module System	68
5.19	Dynamic Loading	69
5.20	Macro functions	70
5.21	Runtime system	70
5.22	Self-Documentation	70
5.23	Box Operations	71
5.24	Pointer Operations	71
5.25	Misc functions	72
5.26	Syntactic Forms	73
Sizzle vs. R5RS		78
	Standard Data Types	78
	Standard Procedures	78
	Supported Syntax	78
	Supported Procedures	78
	Partly implemented	79
	Not supported	79

Glossary 80

Index 82