The Swish Concurrency Engine

Bob Burger, editor

Contents

Chapter 1

Introduction to Swish

1.1 Overview

The Swish Concurrency Engine is a framework used to write fault-tolerant programs with message-passing concurrency. It uses the Chez Scheme [6] programming language and embeds concepts from the Erlang [8] programming language. Swish also provides a web server following the HTTP protocol [14].

Swish uses message-passing concurrency and fault isolation to provide fault-tolerant software [1, 18]. The software is divided into lightweight processes that communicate via asynchronous message passing but are otherwise isolated from each other. Because processes share no mutable state, one process cannot corrupt the state of another process—a problem that plagues software using shared-state concurrency.

Exceptions are raised when the software detects an error and cannot continue normal processing. If an exception is not caught by the process that raised it, the process is terminated. An error logger records process crashes and other software errors.

There are two mechanisms for detecting process termination, *links* and *monitors*. Processes can be linked together so that when one exits abnormally, the others are killed. A process can monitor other processes and receive process-down messages that include the termination reason.

A single event dispatcher receives events from the various processes and sends them to all attached event handlers. Event handlers filter events based on their needs.

Swish is written in Chez Scheme for two main reasons. First, it provides efficient first-class continuations [4, 20] needed to implement lightweight processes with much less memory and CPU overhead than operating system threads. Second, Chez Scheme provides powerful syntactic abstraction capabilities [7] needed to make the code closely reflect the various aspects of the design. For example, the message-passing system uses syntactic abstraction to specify pattern matching succinctly.

I/O operations are performed asynchronously using C code (see Chapter 3), and they complete via Scheme callback functions. Asynchronous I/O is used so that Swish can run in a single thread without blocking for I/O. The results from asynchronous operations are invoked synchronously by the Scheme code, allowing it to control re-entrancy.

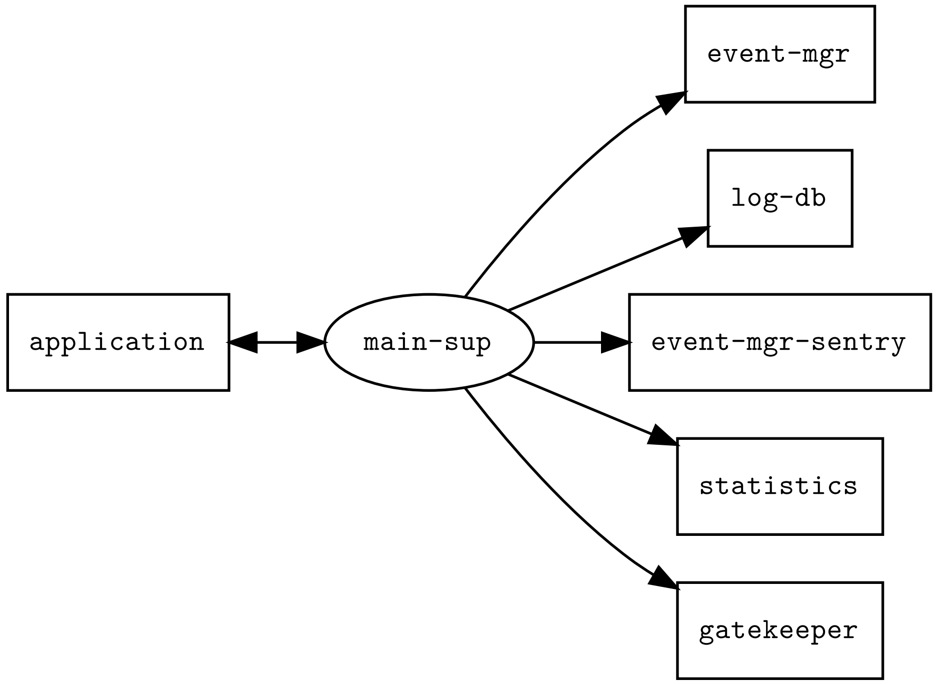

Figure 1.1: Supervision Tree

1.2 Supervision Tree

Calling `app:start` spawns a set of processes organized in a supervision tree. By default, Swish uses the supervision tree illustrated in Figure 1.1. The application (see Chapter 10) is a single gen-server that manages the lifetime of the program. It links the top-level supervisor and shuts down the program when requested or when the linked process dies. The top-level supervisor, main-sup, is configured one-for-all and no restarts so that a failure of any of its children crashes the program. The event-mgr worker is the event manager gen-server (see Chapter 7). The log-db worker is a database gen-server (see Chapter 12) that logs all events to the log database. The event-mgr-sentry worker is used during shutdown to make sure the event manager stops sending events to log-db before log-db shuts down. The statistics worker is a system statistics gen-server (see Chapter 13) that periodically posts a `<statistics>` event. The gatekeeper worker is the gen-server described in Chapter 8.

A web server can be added to the supervision tree by calling `http:add-file-server` or `http:add-server` (see Chapter 14) before calling `app:start`.

Chapter 2

Developing Software with Swish

2.1 Introduction

Swish can be used to build, test, and deploy programs ranging from small scripts to large standalone applications. This chapter describes some of the tools and mechanisms that Swish provides for these purposes.

2.2 Deployment Types

For interactive development, Swish provides a REPL that reads, evaluates, and prints the values of programs entered at the prompt. At the REPL, the `load` procedure can be used to evaluate the contents of a file containing source or object code. This is convenient when developing larger programs.

Swish provides several options for deploying programs. This section describes these options and their trade-offs.

2.2.1 Scripts

A simple deployment option is to place source code in a file that begins with a `#!` line specifying the absolute path to an executable that can evaluate the script. This could be the absolute path to the Swish executable. More often `/usr/bin/env` is used to locate the Swish executable via the program search path. For example, we might have:

```
$ cat hello
#!/usr/bin/env swish
(printf "Hello, World!\n")
$ chmod +x hello
$ ./hello
Hello, World!
```

In the preceding example, running `./hello` invokes the Swish executable with `./hello` as its sole command-line argument. At boot time, Swish calls `swish-start` to process its command-line arguments. Since the first argument is not an option (`-h`, `--help`, etc.), `swish-start` runs the file named by that argument with the remaining arguments, if any, as its command-line arguments. For example, the following `Echo` script processes the arguments that are passed to the script.

```
$ cat Echo
#!/usr/bin/env swish
(printf "~{~:(~a~)~}\n" (command-line-arguments))
$ chmod +x Echo
$ ./Echo some camel case identifiers are hard to read
SomeCamelCaseIdentifiersAreHardToRead
```

To provide arguments to the Swish executable before the script filename, add the `-S` option to `env` and add the desired arguments after `swish`. Here the `--` option tells Swish to start a REPL after loading the script.

```
$ cat howdy
#!/usr/bin/env -S swish --
(printf "Howdy, Folks!\n")
(pretty-print (command-line-arguments))
$ chmod +x howdy
$ ./howdy

Swish Version 0.0.0
Howdy, Folks!
("--" "./howdy")
> (+ 2 3)
5
> (exit)
```

Limitations

There are several constraints to consider when deploying Swish scripts.

- Naturally, Swish must be installed. To use `/usr/bin/env` as in the preceding examples, the `PATH` environment variable must contain the directory where the Swish executable is installed. This is preferable to hard-coding the absolute path to the Swish executable in your scripts.
- Chez Scheme must also be installed. In particular, the version of Chez Scheme that was used to build Swish must be installed. Swish must be able to locate the Chez Scheme boot files `petite.boot` and `scheme.boot`.

  If Chez Scheme is installed in a non-standard location, it may be necessary to set the `SCHEMEHEAPDIRS` environment variable to help Swish locate the boot files. To see where Swish looks for boot files, run `swish --verbose`.
- The `#!` scripts shown in this section do not run under Windows. In MinGW/MSYS, these scripts may work unless Posix path conversion is disabled by setting the `MSYS_NO_PATHCONV`

environment variable to 1. In Cygwin, these scripts may work if the directory containing the script is mounted as its Windows twin, e.g., `C:/Users` and `/Users`. For either of these options to work, Swish must be able to locate the appropriate Chez Scheme DLL via the standard search order. Scripts will fail if the current drive is not the drive containing the script.

2.2.2 Linked Programs

A linked program is simply a Scheme object file that begins with a "`#!/usr/bin/env swish`" line. Swish runs these programs the same way it runs scripts, except that `swish-start` skips the compilation step when it runs the file.

We can use `swish-build` to build a linked program `foo` from a source file `foo.ss`, as follows:

```
$ swish-build -o foo foo.ss
```

Limitations

All of the limitations in Section 2.2.1 apply. Since linked programs use `/usr/bin/env`, the Swish executable must be in a directory in the `PATH` on the target machine.

In addition, the Swish executable (and supporting code) that is used to run a linked program must be identical to the one that was used to build it.

2.2.3 Stand-alone Programs

A stand-alone program consists of an executable and a boot file that must be installed in the same directory. The name of the boot file is the same as the executable's name with any extension replaced by `.boot`. The executable is simply a copy of the Swish executable. The boot file includes the necessary Chez Scheme boot files along with the compiled application code.

We can use `swish-build` to build a linked program `foo` from a source file `foo.ss` by specifying a base boot file via the `-b` option:

```
$ swish-build -o foo foo.ss -b petite
```

Limitations

On Windows, the Swish DLLs `osi.dll`, `uv.dll`, and `sqlite3.dll` must be installed in the same directory as the program executable. These DLLs can be found in the directory containing the Swish executable. In addition, the program executable must be able to locate Microsoft's C Runtime Library `vcruntime140.dll` and the appropriate Chez Scheme DLL via the standard search order.

2.2.4 Services

On Linux and Windows, a Swish application can be started as a service that listens for system shutdown, suspend, and resume messages. To start a Swish application as a service, pass `/SERVICE` as the first command-line argument. On Windows, two additional command-line arguments are required: the service name and the path to the log file where stdout and stderr are redirected. See `swish_service` for details.

2.3 Running Tests

Use the `mat` and `isolate-mat` forms described in Section 2.4.4 to define automated tests. Use the `swish-test` script to run tests and report the results. This script treats each file with a `.ms` extension as a suite of tests. See `swish-test --help all` for details.

2.4 Programming Interface

2.4.1 Configuration

`(app:config` [*obj*]`)` **procedure**
returns: a hashtable

When called with no arguments, the `app:config` procedure returns the configuration data cached in a private process parameter. If the cache is empty, `app:config` first populates the cache by reading data from the file identified by `app:config-filename`. If that file does not exist, the procedure returns an empty hashtable. Otherwise, `app:config` expects the file to contain a single JSON object.

The optional *obj* must be `#f` or a hashtable in the form returned by `json:make-object`. Calling `app:config` with `#f` clears the cache. Calling `app:config` with a hashtable installs the hashtable as the cached value.

`(app:config-filename)` **procedure**
returns: the name of the application's configuration file

The `app:config-filename` procedure returns the name of the application configuration file that `app:config` will read. The filename returned depends on the value of `app:path`.

If `app:path` is set to a value of the form `"`*...*`/bin/`*app-name*[`.`*ext*]`"`, that ends with a `"bin"` directory, then the result is a `config` file in the corresponding `"lib"` directory `"`*...*`/lib/`*app-name*`/config"`. If `app:path` is set to a value of the form `"`*...*`/`*app-name*[`.`*ext*]`"`, then the result is a file with a `.config` extension in the same directory: `"`*...*`/`*app-name*`.config"`.

If `app:path` is not set, the result is a `.config` file in the base directory identified by value of the `base-dir` parameter. This can be useful when loading program code at the REPL during interactive development.

`(app:name` [*name*]`)` **parameter**
value: a string or `#f`

The `app:name` parameter returns the short descriptive name of the application, if known. This value is used by the `app-exception-handler` to identify the program that is reporting an error. The value is also useful as an argument to `display-usage` and `display-help`.

When called with a string, `app:name` treats the value as a file-system path and stores only the last element of the path, dropping any file extension.

For stand-alone programs, `swish-start` sets this parameter to the path of the running executable. For scripts or linked programs, `swish-start` sets this parameter to the path of the script or program file.

`(app:path` [*path*]`)` **parameter**

value: an absolute path or `#f`

The `app:path` parameter returns the absolute path to the script or program, if known. This value is used by `app:config-filename` to determine the location of the application's configuration file. For scripts and linked programs, where the actual executable is Swish, this value is different from that returned by `osi_get_executable_path`.

When called with a string, `app:path` calls `get-real-path` to obtain an absolute path and stores the result.

`(base-dir` [*path*]`)` **parameter**

value: a file-system path

When called without arguments, `base-dir` returns the file-system path of the application's base directory. Otherwise, the base directory is set to *path*, which must specify the file-system path of a directory that exists and is writable by the application. Setting `base-dir` sets the values of `data-dir`, `log-file`, and `tmp-dir` so that they refer to locations within the base directory.

When the application starts, `base-dir` is set to the current directory.

`(data-dir` [*path*]`)` **parameter**

value: a file-system path

The `data-dir` parameter holds the file-system path of a directory in which the application can write persistent data. Applications should not assume that this directory exists, but should instead use `make-directory-path` when creating files under `data-dir`. Its initial value is the path to a `"data"` subdirectory of the directory specified by `base-dir`.

`(log-file` [*filename*]`)` **parameter**

value: a file name

The `log-file` parameter holds the name of the file containing the SQLite log database managed by the `log-db` gen-server, which creates the file and the requisite directory structure when started. Its initial value is the path to a `"Log.db3"` file in the directory specified by `data-dir`.

`(tmp-dir` [*path*]`)` **parameter**

value: a file-system path

The `tmp-dir` parameter holds the file-system path of a directory in which the application can write ephemeral data. Applications should not assume that this directory exists, but should instead use `make-directory-path` when creating files under `tmp-dir`. Its initial value is the path to a `"tmp"` subdirectory of the directory specified by `base-dir`.

`(include-line` *filenme* [*not-found*]`)` **syntax**

returns: see below

The `include-line` macro expects a string constant *filename* identifying a file either via absolute path or as a path relative to a directory in `source-directories`. If such a file can be found at expand time, `include-line` expands to a string containing the first line of that file or the end-of-file object if the file is empty. If the file cannot be found, `include-line` expands to `(`*not-found filename*`)`, where *not-found* defaults to an expression that raises a syntax error indicating that the file could not be found.

`(software-info)` **procedure**

value: a JSON object

The `software-info` procedure returns a JSON object containing the values stored by `software-product-name`, `software-revision`, and `software-version`. Swish populates these parameters for the `swish` and `chezscheme` keys with values that are determined at compile time.

`(software-product-name` [*key*] [*value*]`)` **parameter**

value: a string or `#f`

The `software-product-name` procedure stores or retrieves the product name under the path `(`*key* `product-name)` in the `(software-info)` object. If *value* is omitted, this procedure returns the value, if any, associated with *key*, or else `#f`. If *key* and *value* are omitted, this procedure returns the value associated with the key `(string->symbol (or (app:name) "swish"))`. If specified, *key* must be a symbol and *value* must be a string. If *key* and *value* are specified and a value has already been set for *key*, this procedure has no effect.

`(software-revision` [*key*] [*value*]`)` **parameter**

value: a string or `#f`

The `software-revision` procedure stores or retrieves the software revision under the path `(`*key* `revision)` in the `(software-info)` object. If *value* is omitted, this procedure returns the value, if any, associated with *key*, or else `#f`. If *key* and *value* are omitted, this procedure returns the value associated with the key `(string->symbol (or (app:name) "swish"))`. If specified, *key* must be a symbol and *value* must be a string. If *key* and *value* are specified and a value has already been set for *key*, this procedure has no effect.

`(software-version` [*key*] [*value*]`)` **parameter**

value: a string or `#f`

The `software-version` procedure stores or retrieves the software version under the path `(`*key* `version)` in the `(software-info)` object. If *value* is omitted, this procedure returns the value, if any, associated with *key*, or else `#f`. If *key* and *value* are omitted, this procedure returns the value associated with the key `(string->symbol (or (app:name) "swish"))`. If specified, *key* must be a symbol and *value* must be a string. If *key* and *value* are specified and a value has already been set for *key*, this procedure has no effect.

2.4.2 Program Life Cycle

`(app-exception-handler obj)` **procedure**
returns: unspecified

When a Swish application starts, it sets the default value of the `base-exception-handler` parameter to `app-exception-handler`. This procedure expects a single argument, typically a condition or an object passed to `throw` or `raise`, which it saves in the parameter `debug-condition`.

If `app:name` is set, then `app-exception-handler` writes a message to the console error port, prefixed with the application name, before calling `reset`. If *obj* is a condition, then the message is formed by stripping the "Warning:" or "Exception:" prefix from the output of `display-condition`. Otherwise, the message is generated by calling `exit-reason->english` on *obj*.

If `app:name` is not set, then `app-exception-handler` calls the default exception handler.

`(app-sup-spec [start-specs])` **parameter**
value: a list of child specifications (see page 105)

The `app-sup-spec` parameter supplies the *start-specs* that define the tree of child processes that are started by `app:start` and supervised by the `main-sup` gen-server. The initial value of `app-sup-spec` is constructed by calling `make-swish-sup-spec` with `swish-event-logger` as the sole logger.

`(app:shutdown [exit-code])` **syntax**

This is an alias for `application:shutdown`.

`(app:start)` **procedure**
returns: `ok`

The `app:start` procedure calls `application:start` with a *starter* thunk that calls `supervisor:start&link` with the value of the `app-sup-spec` parameter.

`(make-swish-sup-spec loggers)` **procedure**
returns: a list of child specifications (see page 105)

The `make-swish-sup-spec` procedure builds a list of child specifications representing the default supervision tree described in Section 1.2. The *loggers* argument is a list of `<event-logger>` tuples that will be passed to `log-db:setup` when the `log-db` gen-server is started.

`(swish-start arg ...)` **procedure**
returns: see below

When a stand-alone program starts, it initializes several parameters: `app:name`, `app:path`, `command-line`, and `command-line-arguments`. When a Swish script, linked program, or REPL is started, it performs this initialization by applying `swish-start` to the original command-line arguments.

The `swish-start` procedure expects zero or more strings as arguments and scans them left to right. It adds each string that begins with a single dash (`-`) or double dash (`--`) to a set of options to be handled later. If it encounters the string `"--"`, it sets the REPL option and saves the remaining arguments as files to be loaded. If it encounters a string that does not begin with a dash, it stops scanning the arguments and marks that string and those that follow as ordinary arguments.

If `swish-start` finds `--help` among the options, it displays a summary of Swish's command-line options and returns. If it finds `--version` among the options, it displays version information and returns. If the REPL option is set or it finds no ordinary arguments, then it suppresses the startup message and waiter prompt if `--quiet` is among the options, then loads any files that were specified, and then calls `new-cafe`. Otherwise, it treats the first ordinary argument as a script or linked program. In this case, it installs the ordinary arguments as the value of `command-line`, sets `app:name` and `app:path` to the first of them, sets `command-line-arguments` to the remaining ones, and clears `app:config` before loading the script or linked program. If an error occurs while loading the script or linked program, Swish exits with an exit code of 1. Otherwise, Swish exits normally.

When it starts a REPL, script, or linked program, `swish-start` attempts to import any libraries found in `library-list` whose library path begins with the symbol `swish`.

`repl-level` **parameter**

value: nonnegative fixnum

The `repl-level` parameter returns the nesting depth of the `swish-start` REPL. The value of this parameter is initially zero; it is incremented when a process enters the `swish-start` REPL. It is not affected by `new-cafe`.

2.4.3 Foreign Interface

Scheme libraries that rely on shared objects must arrange to call `load-shared-object` before calling `foreign-procedure` or calling `make-ftype-pointer` on a function ftype. We must consider shared-object naming conventions and search paths on different platforms when calling `load-shared-object`. We can provide its filename argument through some combination of conditional compilation, hard-coded relative or absolute paths, and general computation.

The `provide-shared-object` and `require-shared-object` procedures in this section offer a simple way to: factor platform dependencies out of client code, load shared objects by absolute path to avoid search, specify shared object file names as paths relative to the application configuration file, and hook the operation that loads shared object code.

These procedures coordinate via the application configuration object stored in `app:config`, which is populated on demand by reading the file specified by `app:config-filename`. A Scheme library can call `require-shared-object` before its `foreign-procedure` definitions and rely on demand-loading of the configuration file to supply the absolute path to the shared object code when the library is invoked. This simplifies a process that can otherwise be complicated by the fact that Chez Scheme invokes a library lazily as soon as one of its exports may be referenced.

To see how these procedures are decoupled, consider the following REPL transcript on a 64-bit Linux machine:

```
Swish Version 0.0.0
> (provide-shared-object 'libc "/lib64/libc.so.6")
> (foreign-entry? "fork")
```

```
#f
> (require-shared-object 'libc)
> (foreign-entry? "fork")
#t
> (let ([op (open-file-to-write (app:config-filename))])
    (on-exit (close-port op)
      (json:write op (app:config) 0)))
> (exit)
```

If we start a new Swish REPL, we can call `require-shared-object` without first calling `provide-shared-object` because we explicitly wrote the value of `app:config` to the location specified by `app:config-filename` before exiting the original REPL.

```
Swish Version 0.0.0
> (foreign-entry? "fork")
#f
> (require-shared-object "libc")
> (foreign-entry? "fork")
#t
> (display (utf8->string (read-file (app:config-filename))))
{
  "swish": {
    "shared-object": {
      "libc": {
        "a6le": {
          "file": "/lib64/libc.so.6"
        }
      }
    }
  }
}
```

(`provide-shared-object` *so-name* *filename*) **procedure**
returns: unspecified

The `provide-shared-object` procedure expects a symbol *so-name* as the generic name of a shared object and a string *filename* that is the name of the actual file containing the shared object code. This procedure records *filename* in `app:config` under a path that includes `so-name` and the host `machine-type`.

(`require-shared-object` *so-name* [*handler*]) **procedure**
returns: unspecified

The `require-shared-object` procedure loads a shared object via an absolute path to prevent `load-shared-object` from searching for the shared object file in a system-specific set of directories.

The *so-name* argument is a symbol that specifies the generic name of a shared object. The optional *handler* is a procedure that expects three arguments: *filename* is the absolute path to the shared object file, *key* is *so-name*, and *dict* is the hashtable retrieved from `app:config` via a path that includes *so-name* and the host `machine-type`. This hashtable contains a key `file` whose value is the file name originally supplied by `provide-shared-object`. The handler might examine other keys within the hashtable before loading the shared object. The default handler simply calls `load-shared-object` on *filename*.

If the file name retrieved from `app:config` is a relative path, then `require-shared-object` determines the absolute path by treating the file name as a path relative to the parent directory containing the application configuration file name returned by `app:config-filename`.

`(define-foreign` *name* `(`*arg-name* *arg-type*`) ...)` **syntax**

expands to:

```
(begin
  (define name* (foreign-procedure (symbol->string 'name) (arg-type ...) ptr))
  (define (name arg-name ...)
    (match (name* arg-name ...)
      [(,who . ,err)
       (guard (symbol? who))
       (io-error 'name who err)]
      [,x x])))
```

The `define-foreign` macro defines two procedures: *name*`*` is a raw foreign procedure that expects the specified argument types and returns a `ptr` value, while *name* is a wrapper that calls *name*`*` and raises an `io-error` if it returns an error pair.

2.4.4 Testing

The `(swish mat)` library provides methods to define, iterate through, and run test cases, and to log the results. The `swish-test` script provides a convenient way to run tests and report the results. See `swish-test --help all` for details. To access the `(swish mat)` library directly, run `swish-test --repl` instead of `swish`, then import the library as usual.

Test cases are called *mats* and consist of a name, a set of tags, and a test procedure of no arguments. The set of mats is stored in reverse order in a single, global list. The list of tags allows the user to group tests or mark them. For example, tags can be used to note that a test was created for a particular change request.

`(mat` *name* `(`*tag* ...`)` e_1 e_2 ...`)` **syntax**

expands to: `(add-mat '`*name* `'(`*tag* ...`) (lambda ()` e_1 e_2 ...`))`

The `mat` macro creates a mat with the given *name*, *tag*s, and test procedure e_1 e_2 ... using the `add-mat` procedure.

`(isolate-mat` *name* `(`*tag* ...`)` e_1 e_2 ...`)` **syntax**
`(isolate-mat` *name* `(settings [`*key* *val*`] ...)` e_1 e_2 ...`)`

expands to: `(mat` *name* `(`*tag* ...`) ($isolate-mat (lambda ()` e_1 e_2 ...`)))`

Tests involving process operations, such as `spawn`, `send`, and `receive`, should use `isolate-mat` in place of `mat` to isolate the host system from the test code. The `isolate-mat` macro is provided by the `(swish testing)` library, which can be accessed via `swish-test`.

The `settings` form can be used within `isolate-mat` to override the following defaults:

key	default	description
`tags`	`()`	a list of *tag* ...
`timeout`	60000	deadline for test to complete, in milliseconds
`process-cleanup-deadline`	500	maximum time to wait for spawned processes to terminate, in milliseconds
`process-kill-delay`	100	delay in milliseconds before killing each spawned process that did not terminate by the `process-cleanup-deadline`

`(add-mat` *name tags test*`)` **procedure**
returns: unspecified

The `add-mat` procedure adds a mat to the front of the global list. *name* is a symbol, *tags* is a list, and *test* is a procedure of no arguments.

If *name* is already used, an exception is raised.

`(run-mat` *name reporter*`)` **procedure**
returns: see below

The `run-mat` procedure runs the mat of the given *name* by executing its test procedure with an altered exception handler. If the test procedure completes without raising an exception, the mat result is `pass`. If the test procedure raises exception *e*, the mat result is `(fail .` *e*`)`.

After the mat completes, the `run-mat` procedure tail calls `(`*reporter name tags result statistics*`)`.

If no mat with the given *name* exists, an exception is raised.

`(run-mats` [*name*] ...`)` **syntax**
returns: unspecified

The `run-mats` macro runs each mat specified by symbols *name* .... When no names are supplied, all mats are executed. After each mat is executed, its result, name, and exception if it failed are displayed. When the mats are finished, a summary of the number run, passed, and failed is displayed.

`(run-mats-to-file` *filename*`)` **procedure**
returns: unspecified

The `run-mats-to-file` procedure executes all mats and writes the results into the file specified by the string *filename*. If the file exists, its contents are overwritten. The file format is a sequence of JSON objects readable with `load-results` and `summarize`.

`(for-each-mat` *procedure*`)` **procedure**
returns: unspecified

The `for-each-mat` procedure calls (*procedure name tags*) for each mat, in no particular order.

`(load-results` *filename*`)`	**procedure**

returns: a JSON object

The `load-results` procedure reads the contents of the file specified by string *filename* and returns a JSON object with the following keys:

`meta-data`	a JSON object
`report-file`	*filename*
`results`	a list of JSON objects

The `meta-data` object contains at least the following keys:

`completed`	`#t` if test suite completed, `#f` otherwise
`hostname`	`(osi_get_hostname)` of the host system
`machine-type`	`(machine-type)` of the host system
`test-file`	name of the file containing the tests
`test-run`	uuid generated by `swish-test` for the set of tests run

If the test suite completed, the `meta-data` object also contains the following keys:

`date`	`(format-rfc2822 (current-date))` at the start of the test suite
`software-info`	`(software-info)` for the tested code
`timestamp`	`(erlang:now)` at the start of the test suite

Each result is a JSON object with the following keys:

`message`	error message from failing test, or empty string
`sstats`	a JSON object representing the `sstats-difference` for the test
`stacks`	if test failed with exception *e*, then `(map stack->json (exit-reason->stacks` *e*`))`
`tags`	a list of strings corresponding to the symbolic tags in the `mat` form
`test`	a string corresponding to the symbolic `mat` name
`test-file`	the name of the test file
`type`	the type of result: `"pass"`, `"fail"`, `"skip"`

`(summarize` *files*`)`	**procedure**

returns: five values: the number of passing mats, the number of failing mats, the number of skipped mats, the number of completed suites, and the length of *files*.

The `summarize` procedure reads the contents of each file in *files*, a list of string filenames, and returns the number of passing mats, the number of failing mats, the number of skipped mats, the number of completed test suites, and the number of files specified. An error is raised if any entry is malformed.

Chapter 3

Operating System Interface

3.1 Introduction

This chapter describes the operating system interface. Swish is written in Chez Scheme and runs on Linux, macOS, and Windows. It provides asynchronous I/O via libuv [21] and database support via SQLite [22].

3.2 Theory of Operation

The operating system interface is written in C99 [5] as a shared library that links to the Chez Scheme, libuv, and SQLite libraries. Please refer to Chapter 4 of the *Chez Scheme Version 9 User's Guide* [6] for information on the foreign function interface. C++ is not used because C++ destructors may interact badly with setjmp/longjmp, used by Chez Scheme and Swish.

The single-threaded version of Chez Scheme is used because of its simplicity. All Scheme code runs in the main thread, and all C code must call Scheme functions from the main thread only. In order to keep this thread responsive, operations that block for more than a couple milliseconds are performed asynchronously.

Operations that take longer should be run in a worker thread. Results are communicated back to the main thread using a libuv async handle. Beware of running long operations in the libuv thread pool because there are only a few worker threads (four by default).

For each asynchronous function in the operating system interface, a Scheme callback procedure is passed as the last argument. This callback procedure is later returned to Scheme in a list that includes the results of the asynchronous function call. This approach is simpler and more efficient than calling the callback procedure directly from the C side.

Any time C code stores a pointer to a non-immediate Scheme object, the object must be locked. The operating system interface locks Scheme objects when it stores them in data structures managed in the C heap and unlocks them when the data structures are deallocated.

The operating system interface uses port objects for files, console input, pipes to other processes, and TCP/IP connections. A port object is created by the various open functions, which return a port handle that is used for read, write, and close operations. Once a port is closed, its port object is freed.

Whenever Scheme receives a handle to an object allocated in the C heap, the handle is wrapped in a Scheme record and registered with a guardian. Each type of handle has an associated finalizer (see `add-finalizer`) that uses its guardian to free the objects from the C heap after each garbage collection (see the finalizer process in §4.3).

For interface functions that can fail, an error pair `(`*who* `.` *errno*`)` is returned, where *who* is a symbol representing the name of the particular function that failed and *errno* is either an error number or, in the case of certain SQLite functions, a pair whose car is the error number and cdr is the English error string.

Section 3.3 describes the programming interface from the C side. The Scheme library `(osi)` provides foreign procedures for each C function using the same name. For functions that may return an error pair, two Scheme procedures are defined: one that converts the error pair into an exception, and one with an asterisk suffix that returns the error pair. For example, if the `osi_read_port*` procedure returns error pair `(`*who* `.` *errno*`)`, the `osi_read_port` procedure raises exception `#(osi-error osi_read_port` *who* *errno*`)`.

3.3 Programming Interface

Unless otherwise noted, all C strings are encoded in UTF-8.

3.3.1 C Interface

A single libuv I/O loop is used, `osi_loop`, which is unique to the operating system interface in order to avoid collisions with other libuv integrations.

`void osi_init(void);` **function**

The `osi_init` function disables libuv stdio inheritance, initializes `osi_loop`, initializes the timer used by `osi_get_callbacks`, and sets the list of callbacks to `()`. On Windows, it calls `timeBeginPeriod` to set the timer resolution to 1 ms. This function must be called exactly once from the main thread before any other `osi_*` functions are called.

`void osi_add_callback_list(ptr` *callback*`, ptr` *args*`);` **function**

The `osi_add_callback_list` function adds the callback list `(`*callback* `.` *args*`)` to the list of callbacks. This function must be called only on the main thread and only within the context of the event-loop's call to `osi_get_callbacks`, e.g., within a libuv callback such as `uv_async_cb`.

`void osi_add_callback1(ptr` *callback*`, ptr` *arg*`);` **function**

The `osi_add_callback1` function adds the callback list `(`*callback* *arg*`)` to the list of callbacks. This function must be called only on the main thread and only within the context of the event-loop's call to `osi_get_callbacks`, e.g., within a libuv callback such as `uv_async_cb`.

`void osi_add_callback2(ptr` *callback*`, ptr` *arg1*`, ptr` *arg2*`);` **function**

The `osi_add_callback2` function adds the callback list `(`*callback arg1 arg2*`)` to the list of callbacks. This function must be called only on the main thread and only within the context of the event-loop's call to `osi_get_callbacks`, e.g., within a libuv callback such as `uv_async_cb`.

`void osi_add_callback3(ptr` *callback*`, ptr` *arg1*`, ptr` *arg2*`, ptr` *arg3*`);` **function**

The `osi_add_callback3` function adds the callback list `(`*callback arg1 arg2 arg3*`)` to the list of callbacks. This function must be called only on the main thread and only within the context of the event-loop's call to `osi_get_callbacks`, e.g., within a libuv callback such as `uv_async_cb`.

`ptr osi_make_error_pair(const char*` *who*`, int` *error*`);` **function**

The `osi_make_error_pair` function returns the error pair `(`*who* `.` *error*`)`. This function must be called on the main thread only.

`int osi_send_request(handle_request_func` *handler*`, void*` *payload*`);` **function**

The `osi_send_request` function blocks the calling thread until the main thread returns from calling *handler*`(`*payload*`)` within `osi_get_callbacks`. The *handler* function should execute quickly to avoid blocking the event loop. Typical handlers call `osi_add_callback_list` or one of its variants. `osi_send_request` returns zero if successful. Otherwise, it returns a libuv error code. Calling it from the main thread returns `UV_EPERM`.

`char* osi_string_to_utf8(ptr` *s*`, size_t*` *utf8_len*`);` **function**

The `osi_string_to_utf8` function returns the address of a freshly allocated nul-terminated string representing the Scheme string *s*. The length in bytes of this string excluding the terminating nul is written to `*`*utf8_len*. It returns NULL if `malloc` fails. It is the caller's responsibility to call `free` when this memory is no longer needed.

`int swish_run(int` *argc*`, const char*` *argv*`[], void (*`*custom_init*`)(void));` **function**

The `swish_run` function:

1. resolves the boot file based on *argv*`[0]` by replacing the extension, if any, with `.boot`,
2. initializes Scheme by calling `Sscheme_init`, `Sregister_boot_file`, and `Sbuild_heap`,
3. initializes the operating system interface by calling `osi_init`,

4. establishes a context for `osi_exit`,
5. starts the application by invoking the value of the `scheme-start` parameter, and
6. returns the status provided by `osi_exit`.

```
#if defined(__linux__)                                                    function
int swish_service(int argc, const char* argv[]);
#elif defined(_WIN32)
int swish_service(const wchar_t* service_name, const wchar_t* logfile,
                  int argc, const char* argv[]);
#endif
```

On Linux, the `swish_service` function:

1. resolves the boot file based on *argv*`[0]` by replacing the extension, if any, with `.boot`,
2. initializes Scheme by calling `Sscheme_init`, `Sregister_boot_file`, and `Sbuild_heap`,
3. initializes the operating system interface by calling `osi_init`,
4. adds a handler to listen for messages from `systemd-logind` on the D-Bus system bus,
5. establishes a context for `osi_exit`,
6. starts the application by invoking the value of the `scheme-start` parameter,
7. tears down the handler on exit, and
8. returns the status provided to `osi_exit`.

If the handler receives a `PrepareForShutdown` message with the argument true, it calls `$shutdown`. If it receives a `PrepareForSleep` message, it calls `$suspend` if the argument is true and `$resume` if the argument is false. See Figure 3.1 for information on these callbacks.

On Windows, the `swish_service` function:

1. redirects stdout and stderr to *logfile* and stdin from NUL,
2. connects to the Service Control Manager, establishing a new thread of execution to:
   (a) resolve the boot file based on *argv*`[0]` by replacing the extension, if any, with `.boot`,
   (b) initialize Scheme by calling `Sscheme_init`, `Sregister_boot_file`, and `Sbuild_heap`,
   (c) initialize the operating system interface by calling `osi_init`,
   (d) register a control handler with the Service Control Manager,
   (e) establish a context for `osi_exit`, and
   (f) start the application by invoking the value of the `scheme-start` parameter
3. sets the service status to the status provided by `osi_exit`.

Once running as a service, the Service Control Manager may send control messages to the process via the control handler. For a stop message, the control handler calls the `$shutdown` top-level procedure. For a suspend power event or resume power event, the control handler calls `$suspend` or `$resume` respectively. See Figure 3.1 for information on these callbacks.

symbol	default top-level value	with statistics gen-server (Chapter 13)
`$shutdown`	`application:shutdown`	`application:shutdown`
`$suspend`	`void`	`statistics:suspend`
`$resume`	`void`	`statistics:resume`

Figure 3.1: Service callbacks.

3.3.2 System Functions

`ptr osi_get_argv(void);` **function**

The `osi_get_argv` function returns a Scheme vector of strings constructed from the most recent arguments passed to `osi_set_argv`.

`size_t osi_get_bytes_used(void);` **function**

The `osi_get_bytes_used` function returns the number of bytes used by the C run-time heap. On Linux, it calls the `mallinfo` function. On macOS, it calls the `mstats` function. On Windows, it calls the `_heapwalk` function.

`ptr osi_get_callbacks(uint64_t` *timeout*`);` **function**

The `osi_get_callbacks` function returns a list of callback lists in reverse order of time received. When the list is empty, it blocks up to *timeout* milliseconds before returning. Each callback list has the form `(`*callback result* ...`)`, where *callback* is the callback procedure passed to the asynchronous function that returned one or more *result*s.

`const char* osi_get_error_text(int` *err*`);` **function**

The `osi_get_error_text` function returns the English string for the given error number.

`ptr osi_get_hostname(void);` **function**

The `osi_get_hostname` function returns the host name from `uv_os_gethostname`.

uint64_t `osi_get_hrtime(void);` **function**

The `osi_get_hrtime` function returns the current high-resolution real time in nanoseconds from `uv_hrtime`. It is not related to the time of day and is not subject to clock drift.

uint64_t `osi_get_time(void);` **function**

The `osi_get_time` function returns the current clock time in milliseconds in UTC since the UNIX epoch January 1, 1970. On Windows, it calls the `GetSystemTimeAsFileTime` function in `kernel32.dll`. On all other systems, it calls the `clock_gettime` function with `CLOCK_REALTIME`.

`int osi_is_quantum_over(void);` **function**

The `osi_is_quantum_over` function returns 1 if the current time from `uv_hrtime` is greater than or equal to the threshold set by the most recent call to `osi_set_quantum` and 0 otherwise.

`ptr osi_list_uv_handles(void);` **function**

The `osi_list_uv_handles` function calls `uv_walk` and returns a list of pairs (*handle* . *type*), where *handle* is the address of the `uv_handle_t` and *type* is the `uv_handle_type`.

`ptr osi_make_uuid(void);` **function**

The `osi_make_uuid` function returns a new universally unique identifier (UUID) as a bytevector. On Windows, it calls the `UuidCreate` function in `rpcrt4.dll`. On all other systems, it calls the `uuid_generate` function.

`(string->uuid` *s*`)` **procedure**
returns: a UUID bytevector

The `string->uuid` procedure returns the bytevector *uuid* for string *s* such that `(uuid->string` *uuid*`)` is equivalent to *s*, ignoring case. If *s* is not a string with uppercase or lowercase hexadecimal digits and hyphens as shown in `uuid->string`, exception `#(bad-arg string->uuid` *s*`)` is raised.

`(uuid->string` *uuid*`)` **procedure**
returns: a string

The `uuid->string` procedure returns the uppercase hexadecimal string representation of *uuid*, $HH_3HH_2HH_1HH_0$-HH_5HH_4-HH_7HH_6-HH_8HH_9-$HH_{10}HH_{11}HH_{12}HH_{13}HH_{14}HH_{15}$, where HH_i is the 2-character uppercase hexadecimal representation of the octet at index i of bytevector *uuid*. If *uuid* is not a bytevector of length 16, exception `#(bad-arg uuid->string` *uuid*`)` is raised.

`void osi_set_argv(int` *argc*`, const char *`*argv*`[]);` **function**

The `osi_set_argv` function stores the *argv* pointer to a C vector of *argc* strings for use in the `osi_get_argv` function. It does not copy the strings, so the caller must not deallocate the memory for the arguments.

`void osi_set_quantum(uint64_t` *nanoseconds*`);` **function**

The `osi_set_quantum` function sets the threshold for `osi_is_quantum_over` to be the current time from `uv_hrtime` plus the given number of *nanoseconds*.

3.3.3 Port Functions

The port functions in this section provide generic read, write, and close operations for port objects. The specific implementation depends on the type of port object.

Port handles point to structures whose first element is a pointer to a virtual function table whose type is `osi_port_vtable_t`. This table defines the specific `close`, `read`, and `write` procedures.

`ptr osi_read_port(uptr` *port*`, ptr` *buffer*`, size_t` *start_index*`, uint32_t` *size*`, int64_t` *offset*`, ptr` *callback*`);` **function**

The `osi_read_port` function issues a read on the given *port* of *size* bytes into the bytevector *buffer* at the zero-based *start_index*. For file ports, *offset* specifies the starting file position or -1 for the current position; for all other port types, *offset* must be -1. The function returns `#t` when the read operation is issued and an error pair otherwise. When the read operation finishes, it enqueues the callback list `(`*callback result*`)`, where *result* is the nonnegative number of bytes read when successful and a negative error code otherwise.

`ptr osi_write_port(uptr` *port*`, ptr` *buffer*`, size_t` *start_index*`, uint32_t` *size*`, int64_t` *offset*`, ptr` *callback*`);` **function**

The `osi_write_port` function issues a write on the given *port* of *size* bytes from the bytevector *buffer* at the zero-based *start_index*. For file ports, *offset* specifies the starting file position or -1 for the current position; for all other port types, *offset* must be -1. The function returns `#t` when the write operation is issued and an error pair otherwise. When the write operation finishes, it enqueues the callback list `(`*callback result*`)`, where *result* is the nonnegative number of bytes written when successful and a negative error code otherwise.

`ptr osi_close_port(uptr` *port*`, ptr` *callback*`);` **function**

The `osi_close_port` function issues a close on the given *port*. It returns `#t` when the close operation is issued and an error pair otherwise. When the close operation finishes, it deallocates the port object and enqueues the callback list `(`*callback errno*`)`, where *errno* is 0 when successful and a negative error code otherwise.

3.3.4 Process Functions

`void osi_exit(int` *status*`);` **function**

The `osi_exit` function returns *status* to the context established by `swish_run` or `swish_service`, if any. Otherwise it calls the `_exit` function to terminate the current process with the given exit *status*. It does not return. The `exit` function is not used because on Unix systems it blocks if there is an outstanding read on stdin.

`ptr osi_spawn(const char*` *path*`, ptr` *args*`, ptr` *callback*`);` **function**

The `osi_spawn` function uses the `uv_spawn` function to create a process with the list of string-valued *args* whose standard input, output and error are connected to pipes. It returns `#(`*to-stdin* *from-stdout from-stderr pid*) when the process has been successfully created and an error pair otherwise. *to-stdin* is a port handle for writing bytes to standard input, *from-stdout* is a port handle for reading bytes from standard output, *from-stderr* is a port handle for reading bytes from standard error, and *pid* is an integer identifying the process.

When the process exits, the callback list (*callback pid exit-status term-signal*) is enqueued, where *pid* is the integer process identifier, *exit-status* is the integer exit status, and *term-signal* is the integer termination signal or 0 if the process did not terminate because of a signal.

`ptr osi_spawn_detached(const char* path, ptr args);` **function**

The `osi_spawn_detached` function uses the `uv_spawn` function to create a process with the list of string-valued *args* and the `UV_PROCESS_DETACHED` flag set. It returns an integer identifying the process when the process has been successfully created and an error pair otherwise.

`int osi_get_pid();` **function**

The `osi_get_pid` function uses the `uv_os_getpid` function to return the current operating-system process ID.

`ptr osi_kill(int pid, int signum);` **function**

The `osi_kill` function uses the `uv_kill` function to send termination signal *signum* to the process identified by *pid*. It returns `#t` when successful and an error pair otherwise.

`ptr osi_start_signal(int signum);` **function**

The `osi_start_signal` function uses the `uv_signal_start` function to trap the signal *signum*. It returns a signal handle when successful and an error pair otherwise. Signals are delivered via the internal `@deliver-signal` procedure, which invokes the handler registered via `signal-handler`.

`ptr osi_stop_signal(uptr signal_handle);` **function**

The `osi_stop_signal` function stops trapping the signal and frees the signal handle returned by `osi_start_signal`.

3.3.5 File System Functions

`ptr osi_open_fd(int fd, int close);` **function**

The `osi_open_fd` function returns a port handle for the file descriptor *fd* when successful and an error pair otherwise. When the port is closed, the file descriptor *fd* is closed if and only if *close* is non-zero. It is an error to set *close* to a non-zero value on a standard I/O file descriptor ($0 \leq fd \leq 2$).

`ptr osi_open_file(const char*` *path*`, int` *flags*`, int` *mode*`, ptr` *callback*`);` **function**

The `osi_open_file` function issues an open using the `uv_fs_open` function and the given *path*, *flags*, and *mode*. It returns `#t` when the open operation is issued and an error pair otherwise. When the open operation finishes, it enqueues the callback list `(`*callback result*`)`, where *result* is the nonnegative port handle when successful and a negative error code otherwise.

The following constants are defined for *flags*:

O_APPEND	O_CREAT	O_DIRECT	O_DIRECTORY	O_DSYNC	O_EXCL
O_EXLOCK	O_NOATIME	O_NOCTTY	O_NOFOLLOW	O_NONBLOCK	O_RANDOM
O_RDONLY	O_RDWR	O_SEQUENTIAL	O_SHORT_LIVED	O_SYMLINK	O_SYNC
O_TEMPORARY	O_TRUNC	O_WRONLY			

The following constants are defined for *mode*:

`S_IFMT  S_IFIFO  S_IFCHR  S_IFDIR  S_IFBLK  S_IFREG  S_IFLNK  S_IFSOCK`

`ptr osi_get_executable_path(void);` **function**

The `osi_get_executable_path` function uses the `uv_exepath` function to return the full path string of the executable file of the current process when successful and an error pair otherwise.

`ptr osi_get_file_size(uptr` *port*`, ptr` *callback*`);` **function**

The `osi_get_file_size` function uses the `uv_fs_fstat` function to issue a status operation on the file associated with the given file *port*. It returns `#t` when the status operation is issued and an error pair otherwise. When the status operation finishes, it enqueues the callback list `(`*callback result*`)`, where *result* is the nonnegative file size when successful and a negative error code otherwise.

`ptr osi_get_real_path(const char*` *path*`, ptr` *callback*`);` **function**

The `osi_get_real_path` function uses the `uv_fs_realpath` function to issue a realpath operation on the given *path*. It returns `#t` when the realpath operation is issued and an error pair otherwise. When the realpath operation finishes, it enqueues the callback list `(`*callback result*`)`, where *result* is the string path when successful and a negative error code otherwise.

`ptr osi_get_temp_directory(void);` **function**

The `osi_get_temp_directory` function uses the `uv_os_tmpdir` function to return the string path of the temporary directory and an error pair otherwise.

`ptr osi_chmod(const char*` *path*`, int` *mode*`, ptr` *callback*`);` **function**

The `osi_chmod` function issues a chmod operation using the `uv_fs_chmod` function and the given *path* and *mode*. It returns `#t` when the chmod operation is issued and an error pair otherwise.

When the chmod operation finishes, it enqueues the callback list (*callback errno*), where *errno* is 0 when successful and a negative error code otherwise.

`ptr osi_make_directory(const char* path, int mode, ptr callback);` **function**

The `osi_make_directory` function issues a mkdir operation using the `uv_fs_mkdir` function with the given *path* and *mode*. It returns `#t` when the mkdir operation is issued and an error pair otherwise. When the mkdir operation finishes, it enqueues the callback list (*callback errno*), where *errno* is 0 when successful and a negative error code otherwise.

`ptr osi_list_directory(const char* path, ptr callback);` **function**

The `osi_list_directory` function issues a scandir operation using the `uv_fs_scandir` function with the given *path*. It returns `#t` when the scandir operation is issued and an error pair otherwise. When the scandir operation finishes, it enqueues the callback list (*callback result*), where *result* is ((*name* . *type*) ...) when successful and a negative error code otherwise.

name is the string name of the directory entry, and *type* is one of the following constants:

DIRENT_UNKNOWN	DIRENT_FILE	DIRENT_DIR	DIRENT_LINK	DIRENT_FIFO
DIRENT_SOCKET	DIRENT_CHAR	DIRENT_BLOCK		

`ptr osi_remove_directory(const char* path, ptr callback);` **function**

The `osi_remove_directory` function issues a rmdir operation using the `uv_fs_rmdir` function with the given *path*. It returns `#t` when the rmdir operation is issued and an error pair otherwise. When the rmdir operation finishes, it enqueues the callback list (*callback errno*), where *errno* is 0 when successful and a negative error code otherwise.

`ptr osi_rename(const char* path, const char* new_path, ptr callback);` **function**

The `osi_rename` function issues a rename operation using the `uv_fs_rename` function of *path* to *new_path*. It returns `#t` when the rename operation is issued and an error pair otherwise. When the rename operation finishes, it enqueues the callback list (*callback errno*), where *errno* is 0 when successful and a negative error code otherwise.

`ptr osi_get_stat(const char* path, int follow, ptr callback);` **function**

The `osi_get_stat` function issues a status operation on the given *path*. When *follow* is non-zero, it uses the `uv_fs_stat` function to follow a symbolic link; otherwise, it uses the `uv_fs_lstat` function. It returns `#t` when the status operation is issued and an error pair otherwise. When the status operation finishes, it enqueues the callback list (*callback result*), where *result* is a `<stat>`

tuple when successful and a negative error code otherwise.

`<stat>`	**tuple**

dev: device ID of the device containing the file
mode: mode of the file
nlink: number of hard links to the file
uid: user ID of the file
gid: group ID of the file
rdev: device ID if file is character or block special
ino: file serial number
size: For regular files, the file size in bytes. For symbolic links, the length in bytes of the path in the link.
blksize: optimal block size for I/O
blocks: number of blocks allocated for the file
flags: user-defined flags for the file
gen: file generation number
atime: time of last access
mtime: time of last data modification
ctime: time of last status change
birthtime: time of file creation

The time entries contain `(`*sec* `.` *nsec*`)`, where *sec* is the number of seconds in UTC since the UNIX epoch January 1, 1970, and *nsec* is the number of nanoseconds after *sec*.

`ptr osi_unlink(const char*` *path*`, ptr` *callback*`);`	**function**

The `osi_unlink` function issues an unlink operation using the `uv_fs_unlink` function with the given *path*. It returns `#t` when the unlink operation is issued and an error pair otherwise. When the unlink operation finishes, it enqueues the callback list `(`*callback errno*`)`, where *errno* is 0 when successful and a negative error code otherwise.

`ptr osi_watch_path(const char*` *path*`, ptr` *callback*`);`	**function**

The `osi_watch_path` function uses the `uv_fs_event_start` function to track changes to *path*. When *path* is a directory, its subdirectories are not tracked. Every time a change is detected, a callback list `(`*callback filename events*`)` is enqueued, where *events* is 1 for rename, 2 for change, and 3 for rename and change. If the watcher encounters an error, the callback list `(`*callback errno*`)` is enqueued.

The `osi_watch_path` function returns a path watcher handle when successful and an error pair otherwise.

`void osi_close_path_watcher(uptr` *watcher*`);`	**function**

The `osi_close_path_watcher` function stops and closes the path *watcher* from `osi_watch_path`.

3.3.6 TCP/IP Functions

`ptr osi_connect_tcp(const char* `*node*`, const char* `*service*`, ptr `*callback*`);` **function**

The `osi_connect_tcp` function initiates a TCP/IP connection to host *node* on port *service*. It returns `#t` when the operation starts and an error pair otherwise. The *node* string may be a host name or numeric host address string, and the *service* string may be a service name or port number represented as a string. The `uv_getaddrinfo` function is used to retrieve a list of addresses. For the first address for which a connection succeeds using the `uv_tcp_connect` function, the completion list `(`*callback port*`)` is enqueued, where *port* is a handle to a port that reads from and writes to this connection. When the operation fails, the callback list `(`*callback error-pair*`)` is enqueued.

`ptr osi_listen_tcp(const char* `*address*`, uint16_t `*port*`, ptr `*callback*`);` **function**

The `osi_listen_tcp` function starts a TCP/IP listener on the given *port* of the IPv4 or IPv6 interface specified by *address* using the `uv_listen` function. It returns a TCP/IP listener handle when successful and an error pair otherwise.

Specify an IPv4 interface *address* using dot-decimal notation, e.g. `127.0.0.1`. Use `0.0.0.0` to specify all IPv4 interfaces.

Specify an IPv6 interface *address* using colon-hexadecimal notation, e.g. `::1`. Use `::` to specify all IPv6 interfaces.

Specify *port* 0 to have the operating system choose an available port number, which can be queried using `osi_get_tcp_listener_port`.

When a connection is accepted, the callback list `(`*callback port*`)` is enqueued, where *port* is a handle to a port that reads from and writes to this connection. When a connection fails, the callback list `(`*callback error-pair*`)` is enqueued.

`void osi_close_tcp_listener(uptr `*listener*`);` **function**

The `osi_close_tcp_listener` function closes the given TCP/IP *listener* opened by `osi_listen_tcp`.

`ptr osi_get_tcp_listener_port(uptr `*listener*`);` **function**

The `osi_get_tcp_listener_port` function returns the port number of the given TCP/IP *listener* opened by `osi_listen_tcp` when successful and an error pair otherwise.

`ptr osi_get_ip_address(uptr `*port*`);` **function**

The `osi_get_ip_address` function uses the `uv_tcp_getpeername` function to return a string representation of the address of the peer of a TCP/IP *port* opened by `osi_connect_tcp` or `osi_listen_tcp` when successful and an error pair otherwise.

An IPv4 address is shown in dot-decimal notation followed by a colon and the port number, e.g. `127.0.0.1:80`.

An IPv6 address is shown in bracketed colon-hexadecimal notation followed by a colon and the port number, e.g. `[::1]:80`.

3.3.7 SQLite Functions

For each open SQLite database, a single worker thread performs the operations so that the main thread is not blocked. SQLite is compiled in multi-thread mode. The documentation states: "In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads." Concern: Two threads simultaneously access a SQLite database connection. Mitigation: The operating system interface maintains a busy bit for each database handle. Functions attempting to access a busy database return the error pair (*function-name* . UV_EBUSY).

SQLite has five data types, which are mapped as follows to Scheme data types:

SQLite	*Scheme*
NULL	`#f`
INTEGER	exact integer
REAL	flonum
TEXT	string
BLOB	bytevector

SQLite extended result codes are enabled. Because the error codes overlap system error codes, the operating system interface maps them to system error codes by negating the sum of the result code and 6,000,000. The `osi_get_error_text` function supports these mapped error codes.

SQLite returns additional error information in English strings, so error pairs from SQLite are often of the form (*who* . (*errno* . *text*)), where *errno* is the mapped SQLite extended result code and *text* is the English error string.

`ptr osi_open_database(const char*` *filename*, `int` *flags*, `ptr` *callback*`);` **function**

The `osi_open_database` function starts a worker thread that uses the `sqlite3_open_v2` function to open the database specified by the *filename* string and *flags*. The *flags* specify, for example, whether the database should be opened in read-only mode or whether it should be created when the file does not exist. The function returns `#t` when the thread is created and an error pair otherwise.

When the open operation finishes, it enqueues the callback list (*callback* *result*), where *result* is the database handle when successful and an error pair otherwise.

`ptr osi_close_database(uptr` *database*, `ptr` *callback*`);` **function**

The `osi_close_database` function starts a close operation in the given *database* worker thread. It returns `#t` when the operation is started and an error pair otherwise.

After the worker thread finalizes all prepared statements, it uses the `sqlite3_close` function to close the *database*. When finished, it enqueues the callback list (*callback result*), where *result* is `#t` when successful and an error pair otherwise.

`ptr osi_prepare_statement(uptr` *database*`, ptr` *sql*`, ptr` *callback*`);` **function**

The `osi_prepare_statement` function starts a prepare operation on the given *database* worker thread. It returns `#t` when the operation is started and an error pair otherwise.

The worker thread uses the `sqlite3_prepare_v2` function to prepare the given *sql* statement. It enqueues the callback list (*callback result*), where *result* is the statement handle when successful and an error pair otherwise.

`ptr osi_finalize_statement(uptr` *statement*`);` **function**

The `osi_finalize_statement` function uses the `sqlite3_finalize` function to finalize the *statement*. It returns `#t` when successful and an error pair otherwise. The return code from `sqlite3_finalize` is not checked because the statement is finalized regardless of the return value.

`ptr osi_bind_statement(uptr` *statement*`, int` *index*`, ptr` *datum*`);` **function**

The `osi_bind_statement` function maps the Scheme *datum* to SQLite and binds it to the *statement* at the zero-based SQL parameter *index*. It returns `#t` when successful and an error pair otherwise. The error pair `(osi_bind_statement .` UV_EINVAL`)` is returned when `datum` cannot be mapped to SQLite.

`ptr osi_clear_statement_bindings(uptr` *statement*`);` **function**

The `osi_clear_statement_bindings` function uses the `sqlite3_clear_bindings` function to clear the bindings for the *statement*. It returns `#t` when successful and an error pair otherwise.

`ptr osi_get_last_insert_rowid(uptr` *database*`);` **function**

The `osi_get_last_insert_rowid` function uses the `sqlite3_last_insert_rowid` function to return the last insert rowid of the *database* when successful and an error pair otherwise.

`ptr osi_get_statement_columns(uptr` *statement*`);` **function**

The `osi_get_statement_columns` function uses the `sqlite3_column_count` and `sqlite3_column_name` functions to return a vector of column name strings for the *statement* when successful and an error pair otherwise.

`ptr osi_get_statement_expanded_sql(uptr` *statement*`);` **function**

The `osi_get_statement_expanded_sql` function uses the `sqlite3_expanded_sql` function to return the expanded SQL string associated with the *statement* when successful and an error pair otherwise.

`ptr osi_reset_statement(uptr` *statement*`);` **function**

The `osi_reset_statement` function uses the `sqlite3_reset` function to reset the *statement*. It returns `#t` when successful and an error pair otherwise.

`ptr osi_step_statement(uptr` *statement*`, ptr` *callback*`);` **function**

The `osi_step_statement` function issues a step operation on the database worker thread associated with *statement*. It returns `#t` when the operation is started and an error pair otherwise.

The worker thread uses the `sqlite3_step` function to execute the *statement*. If it returns SQLITE_-DONE, the callback list `(`*callback* `#f)` is enqueued. If it returns SQLITE_ROW, the callback list `(`*callback* `#(`*value* ...`))` is enqueued with the vector of column values mapped from SQLite to Scheme. Otherwise, the callback list `(`*callback* *error-pair*`)` is enqueued.

`void osi_interrupt_database(uptr` *database*`);` **function**

The `osi_interrupt_database` function calls the `sqlite3_interrupt` function to interrupt the current operation of the *database*.

`ptr osi_get_sqlite_status(int` *operation*`, int` *resetp*`);` **function**

The `osi_get_sqlite_status` function uses the `sqlite3_status64` function with the given *operation* and *reset* flag to return `#(`*current* *highwater*`)` when successful and an error pair otherwise.

3.3.8 Message-Digest Functions

`ptr osi_open_SHA1();` **function**

The `osi_open_SHA1` function returns an error pair or a context for computing the SHA1 message digest.

`ptr osi_hash_data(uptr` *ctxt*`, ptr` *bv*`, size_t` *start_index*`, uint32_t` *size*`);` **function**

The `osi_hash_data` function computes the SHA1 message digest incrementally on the *size* bytes at the zero-based *start_index* of bytevector *bv* updating the context *ctxt*. It returns `#t` when successful and an error pair otherwise.

`ptr osi_get_SHA1(uptr` *ctxt*`);` **function**

The `osi_get_SHA1` function takes a *ctxt* that was created by `osi_open_SHA1` and updated by calling `osi_hash_data` on a set of buffers and returns as a bytevector the SHA1 message digest of the buffers. If unsuccessful, it returns an error pair.

`ptr osi_close_SHA1(uptr` *ctxt*`);` **function**

The `osi_close_SHA1` function frees a *ctxt* that was allocated by `osi_open_SHA1`.

Chapter 4

Erlang Embedding

4.1 Introduction

This chapter describes the design of the message-passing concurrency model. It provides a Scheme embedding of a significant subset of the Erlang programming language [1, 2].[1] Tuple and pattern matching macros provide succinct ways of composing and decomposing data structures.

The basic unit of sequential computation is the *process*. Each process has independent state and communicates with other processes by message passing. Because processes share no mutable state, one process cannot corrupt the state of another process—a problem that plagues software using shared-state concurrency. Concern: System procedures that mutate data can cause state corruption. Mitigation: The code is inspected for use of these procedures.

An uncaught exception in one process does not affect any other process. A process can be monitored for termination, and it can be linked to another process so that, when either process exits, the other one receives an exit signal. Processes are implemented with one-shot continuations [4], and the concurrent system is simulated by the single-threaded program using software timer interrupts. The operating system interface (see Chapter 3) provides asynchronous input/output (I/O) so that processes waiting for I/O do not stop other processes from executing.

For exceptions, we use Erlang's approach of encoding the information in a machine-readable datum rather than a formatted string. Doing so makes it possible to write code that matches particular exceptions without having to parse strings, and the exception is human language independent.

The rest of this chapter is organized as follows. Section 4.2 introduces the main data structures, Section 4.3 describes how the concurrency model works, and Section 4.4 gives the programming interface.

4.2 Data Structures

q Queues are used in several key places: the inbox of messages for each process, the list of processes ready to run, and the list of sleeping processes. A *queue* is a doubly-linked list with a sentinel value,

[1] Tuples, denoted by $\{e_1, \ldots, e_n\}$ in Erlang, are implemented as vectors: `#(`e_1 ... e_n`)`. Similarly records, defined as syntactic sugar over tuples in Erlang, are implemented as syntactic sugar over vectors.

the queue's identity. Both the sentinel value and the elements of the queue are instances of `q`, a Scheme record type with mutable `prev` and `next` fields. This representation enables constant-time insertion and deletion operations.

`msg` When a *message* is sent to a process, its contents are wrapped in an instance of `msg`, a Scheme record type that extends `q` with an immutable `contents` field. This `msg` is inserted into the process's inbox and removed when the process receives it.

`pcb` A *process* is an instance of `pcb`, a Scheme record type that extends `q` with an immutable `id` field, the process's unique positive exact integer, an immutable `create-time` field, the process's create time from `erlang:now`, an immutable `parameters` field, the process's weak eq-hashtable mapping process parameters to values, and the following mutable fields:

- `name`: registered name or `#f`
- `cont`: one-shot continuation if live and not currently running or `#f` otherwise
- `sic`: system interrupt count
- `winders`: list of winders if live and not currently running or `()` otherwise
- `exception-state`: exception state if live and not currently running, exit reason if dead, or `#f` if currently running
- `inbox`: queue of `msg` if live or `#f` if dead
- `precedence`: wake time if sleeping or 0 if ready to run
- `flags`: fixnum with bit 0 set when sleeping, bit 1 set when the process traps exits, bit 2 set when the process is blocked for I/O, and bit 3 set when the process is pending a keyboard interrupt
- `links`: list of linked processes
- `monitors`: list of monitors
- `src`: source location `#(at` *char-offset filename*`)` when available if waiting in a `receive` macro, a string if blocked for I/O, or `#f`

`mon` A *monitor* is an instance of `mon`, a Scheme record type with two immutable fields, `origin` and `target`, each of which is a process.

`osi-port` An *osi-port* is an instance of `osi-port`, a Scheme record type with an immutable `name` field, an immutable `create-time` field, and a mutable `handle` field that wraps an operating system interface port. The `handle` field is set to `#f` when the osi-port is closed.

`path-watcher` A *path watcher* is an instance of `path-watcher`, a Scheme record type with an immutable `path` field, an immutable `create-time` field, and a mutable `handle` field. The `handle` field is set to `#f` when the path watcher is closed.

`listener` A *TCP listener* is an instance of `listener`, a Scheme record type with immutable `address`, `port-number`, and `create-time` fields and a mutable `handle` field. The `handle` field is set to `#f` when the listener is closed.

4.3 Theory of Operation

The system uses a *scheduler* to execute one process at a time. Each process holds its own system interrupt count (updated by `enable-interrupts` and `disable-interrupts`), list of winders (maintained by `dynamic-wind` and the system primitive `$current-winders`), and exception state (maintained by `current-exception-state`). The scheduler captures the one-shot continuation for a process with an empty list of winders so that, when it invokes the continuation of another process, it does not run any winders. Concern: Using a system procedure that relies on the global winders list may lead to incorrect behavior. Mitigation: System procedures that rely on the global winders list are called from only one process at a time using the *gatekeeper* described in Chapter 8. The gatekeeper hooks the `$cp0`, `$np-compile`, `pretty-print`, and `sc-expand` system primitives.

Spawning a new process is not as simple as capturing a one-shot continuation and creating a `pcb` record, because the continuation's stack link [20] would be the continuation of the caller, and its list of winders would be the caller's. Thus, the scheduler remembers the current list of winders and then sets it to the empty list before capturing a one-shot continuation. This return continuation is stored in a mutable variable so that it is not closed over by the new process. Next, a full continuation is captured to create the initial exception state that will terminate the new process when an uncaught exception is raised. So that this full continuation does not refer to the caller's continuation, the current stack link is set to the null continuation before capturing it. After capturing the full continuation, a one-shot continuation for the new process is captured and returned to the caller via the return continuation.

Each process runs until it waits in a `receive` macro or `wait-for-io` procedure, is preempted by the `timer-interrupt-handler`, or exits. The operating system interface (see Chapter 3) provides asynchronous I/O operations so that the scheduler can execute other processes while the system is performing I/O. The timer interrupt handler runs every 1000 procedure calls.[2] The scheduler uses `osi_set_tick` and `osi_is_tick_over` to determine when the time quantum for a process has elapsed.

When process p exits with reason r, a message matching `` `(DOWN `` m p r`)` is sent to each of its monitor m's `origin` processes. A message matching `` `(EXIT `` p r`)` is sent to each linked process that traps exits. If r is not `normal`, each linked process that does not trap exits is killed with reason r.

A process can be registered with a global name, a symbol. This name can be used instead of the process record itself to send it messages. A global *registrar* maintains an eq-hashtable mapping names to processes. The reverse mapping is maintained in the `pcb` record through the `name` field.

There are two system processes: the *event-loop* and the *finalizer*.

The event-loop process calls `osi_get_callbacks` to retrieve callback lists from the operating system interface. It executes each callback with interrupts disabled. Event-loop callbacks are designed to execute quickly without failing or causing new completion packets to be enqueued. Typical callbacks

[2] 1000 was chosen because Chez Scheme performs its internal interrupt checks every 1000 ticks.

register objects that wrap operating system interface handles with a guardian and send messages to a process. If the event-loop process exits with reason r, the system logs the event `#(event-loop-process-terminated` r`)` with `console-event-handler` and calls `osi_exit` with exit code 80.

The scheduler maintains the *run queue*, a queue of ready-to-run processes, and the *sleep queue*, a queue of sleeping processes. Both are ordered by increasing precedence and preserve the order of insertion for processes with the same precedence. For the run queue, each process has precedence 0 in order to implement round-robin scheduling. For the sleep queue, each process uses its wake time as the precedence.

When the run queue is empty, the event-loop process calls `osi_get_callbacks` with a non-zero timeout based on the first entry in the sleep queue to avoid busy waiting. When the event-loop process finishes processing all completion packets, it places itself at the end of the run queue.

Concern: Some process may starve another process. Mitigation: The run queue is managed with round-robin scheduling to prevent starvation. The event-loop process does not starve other processes because it drains the completion queue without causing new completion packets to be enqueued.

The finalizer process runs the finalizers registered via `add-finalizer`. These finalizers typically close operating system interface handles to objects that are no longer accessible. Concern: Ill-behaved finalizers may cause memory and handle leaks. Mitigation: Finalizers are designed to execute quickly without failing. Typical finalizers guard against errors when closing handles. If the finalizer process exits with reason r, the system logs the event `#(finalizer-process-terminated` r`)` with `console-event-handler` and calls `osi_exit` with exit code 80.

Once the finalizer process runs all the finalizers, it waits until another garbage collection has occurred before running again. The system hooks the `collect` procedure so that it sends a wake-up message to the finalizer process every time a garbage collection occurs. When the finalizer receives the wake-up message, it pumps all other wake-up messages from its inbox, since there may have been more than one garbage collection since it last ran.

Asynchronous I/O operations for COM ports, named pipes, external operating system processes, files, console input, and TCP connections are implemented with custom binary ports so that they have the same interface as the system I/O procedures. The system I/O procedures are not used because they perform synchronous I/O. The custom port buffer size is set to 1024[3] with `custom-port-buffer-size`. The custom binary port read and write procedures call `osi_read_port` and `osi_write_port` with callbacks that send a message to the calling process, which waits until it receives the message.

Concern: Using a port from more than one process at the same time may cause errors including buffer corruption. Mitigation: The code is inspected for concurrent use of ports. Port visibility is typically limited to a single process.

For two-way communication ports, we use two custom ports: one exclusively for input, and one exclusively for output. We do not use custom input/output ports for two reasons. First, textual input/output ports created with `transcoded-port` are not safe to use from two concurrent processes because one transcoding buffer is used for both reading and writing. Second, the input side of a port is commonly used only by a reader process, and the output side of a port is commonly used

[3] 1024 was chosen because Chez Scheme uses 1024 for the buffer size of buffered transcoded ports.

only by a writer process. Keeping the input and output sides separate prevents concurrent use. The underlying handle is closed when the output port is closed.

Concern: Failing to close a handle from the operating system interface that is no longer used causes resource leaks. Mitigation: An *osi-port guardian* created by `make-foreign-handle-guardian` with type name `osi-ports` is used to identify and close inaccessible osi-ports. A *path-watcher guardian* created by `make-foreign-handle-guardian` with type name `path-watchers` is used to identify and close inaccessible path watchers. A *listener guardian* created by `make-foreign-handle-guardian` with type name `tcp-listeners` is used to identify and close inaccessible TCP listeners. In all cases, interrupts are disabled around code that wraps handles and registers objects with guardians in order to prevent the current process from being killed during this critical time.

4.4 Programming Interface

4.4.1 Process Creation

`(spawn` *thunk*`)` **procedure**

returns: a process

The `spawn` procedure creates and returns a new process that executes *thunk*, a procedure of no arguments. The new process starts with `name` = `#f`, `sic` = 0 (interrupts enabled), `winders` = `()`, an `exception-state` that terminates the process on an unhandled exception, an empty `inbox`, `precedence` = 0, `flags` = 0 (the process is not sleeping and does not trap exits), `links` = `()`, `monitors` = `()`, and `src` = `#f`.

`(spawn&link` *thunk*`)` **procedure**

returns: a process

Like `spawn`, the `spawn&link` procedure creates and returns a new process that executes *thunk*. In addition, it links the new process to the calling process.

4.4.2 Process Registration

`(get-registered)` **procedure**

returns: a list of registered process names

The `get-registered` procedure returns a list of currently registered process names from the registrar.

`(register` *name* *process*`)` **procedure**

returns: `#t`

The `register` procedure adds *name* $\rightarrow$ *process* to the registrar and sets *process*.`name` = *name*. When a registered process exits, its registration is removed. If *name* is not a symbol, exception `#(bad-arg register` *name*`)` is raised. If *process* is not a process, exception `#(bad-arg register` *process*`)` is raised. If *process* is dead, exception `#(process-dead` *process*`)` is raised. If *process* is already registered to name *n*, exception `#(process-already-registered` *n*`)` is raised. If *name* is already registered to process *p*, exception `#(name-already-registered` *p*`)` is raised.

`(unregister` *name*`)` **procedure**

returns: `#t`

The `unregister` procedure removes *name* → *process* from the registrar and sets *process*.`name` = `#f`. If *name* is not registered, exception `#(bad-arg unregister` *name*`)` is raised.

`(whereis` *name*`)` **procedure**

returns: a process | `#f`

The `whereis` procedure returns the process associated with *name* or `#f` if *name* is not registered. If *name* is not a symbol, exception `#(bad-arg whereis` *name*`)` is raised.

4.4.3 Process Termination, Links, and Monitors

`(catch` *e1 e2* ...`)` **syntax**

expands to: `($trap (lambda ()` *e1 e2* ...`) ->EXIT)`

The `catch` macro evaluates expressions *e1 e2* ... in a dynamic context that traps exceptions. If no exception is raised, the return value is the value of the last expression. If exception *reason* is raised, `#(EXIT` *reason*`)` is returned.

`(try` *e1 e2* ...`)` **syntax**

expands to: `($trap (lambda ()` *e1 e2* ...`) ->fault-condition)`

The `try` macro evaluates expressions *e1 e2* ... in a dynamic context that traps exceptions. If no exception is raised, the return value is the value of the last expression. If exception *reason* is raised, the return value is a fault condition matching the extended match pattern `` `(catch`` *reason* [*e*]`)`.

`` `(catch`` *r* [*e*]`)` **match-extension**

matches: exceptions trapped by `try` or `catch`

The extended match pattern `` `(catch`` *r* [*e*]`)` matches exceptions trapped by `try`. For compatibility with older code, this pattern also matches exceptions trapped by `catch`. The *r* pattern is matched against the exit reason in the trapped exception. The optional *e* pattern is typically a `,`*variable* pattern that binds *variable* for use as an argument to `throw` or `raise`. If the trapped exception is a fault condition generated by `throw`, `make-fault`, or `make-fault/no-cc`, then *e* is matched against the fault condition, which may contain additional debugging context. Otherwise, *e* is matched against the exit reason.

`(throw` *r* [*inner*]`)` **procedure**

returns: does not return

The `throw` procedure raises a fault condition containing reason *r*, an optional inner exception *inner*, and the current continuation, which may provide useful debugging context. The exception raised may be trapped by `try` and matched using the extended match pattern `` `(catch `` *r* [*e*]`)`.

`(make-fault` *r* [*inner*]`)` **procedure**

returns: a fault condition

The `make-fault` procedure returns a fault condition containing reason *r*, an optional inner exception *inner*, and the current continuation, which may provide useful debugging context. The return value matches the extended match pattern `` `(catch `` *r* [*e*]`)`.

`(make-fault/no-cc` *r* [*inner*]`)` **procedure**

returns: a fault condition

The `make-fault/no-cc` procedure returns a fault condition containing reason *r*, and an optional inner exception *inner*, but omits the current continuation. The return value matches the extended match pattern `` `(catch `` *r* [*e*]`)`.

`(demonitor` *monitor*`)` **procedure**

returns: `#t`

The `demonitor` procedure removes a *monitor* created by the calling process (*self*) from *self*.`monitors` and *monitor*.`target`.`monitors` if present. If *monitor* is not a monitor with `origin` = *self*, exception `#(bad-arg demonitor` *monitor*`)` is raised.

`(demonitor&flush` *monitor*`)` **procedure**

returns: `#t`

The `demonitor&flush` procedure provides a convenient way to demonitor and flush any remaining `DOWN` message from the calling process's `inbox`. It performs the following operations:

```
(demonitor monitor)
(receive (until 0 #t)
  [`(DOWN ,@monitor ,_ ,_) #t])
```

`(kill` *process reason*`)` **procedure**

returns: `#t`

The `kill` procedure is used to terminate a process.

1. If *process* is not a process, exception `#(bad-arg kill` *process*`)` is raised.
2. If *process* has already exited, nothing happens.
3. If *reason* is `kill`, *process* is terminated with reason `killed`, even if it traps exits.
4. If *process* traps exits, a message matching `` `(EXIT `` *self reason*`)` is sent to *process*, where *self* is the calling process.
5. If *process* does not trap exits and *reason* is `normal`, nothing happens.
6. Otherwise, *process* is terminated with *reason*.

(link *process*) **procedure**

returns: `#t`

The `link` procedure creates a bi-directional link between the calling process (*self*) and *process*. No more than one link can exist between two processes, but it is not an error to call `link` more than once on the same two processes.

1. If *process* is not a process, exception `#(bad-arg link` *process*`)` is raised.
2. If *process* is *self*, nothing happens.
3. If *process* has not exited, then if the two processes are already linked, nothing happens; otherwise, *self* is added to *process*.`links`, and *process* is added to *self*.`links`.
4. Otherwise, *process* has exited with reason r = *process*.`exception-state`.
   (a) If *self* traps exits, a message matching `` `(EXIT `` *process* r`)` is sent to *self*.
   (b) If *self* does not trap exits and *reason* is `normal`, nothing happens.
   (c) Otherwise, *self* is terminated with reason r.

(monitor *process*) **procedure**

returns: a monitor

The `monitor` procedure creates and returns a new monitor m with `origin` = the calling process (*self*) and `target` = *process*. Unlike `link`, `monitor` can create more than one connection between the same processes. It adds m to *self*.`monitors` and *process*.`monitors`. When *process* exits or has already exited with reason r, a message matching `` `(DOWN `` m *process* r`)` is sent to *self*. If *process* is not a process, exception `#(bad-arg monitor` *process*`)` is raised.

(monitor? *x*) **procedure**

returns: a boolean

The `monitor?` procedure determines whether or not the datum x is a monitor.

(unlink *process*) **procedure**

returns: `#t`

The `unlink` procedure removes the bi-directional link if present between the calling process (*self*) and *process* by removing *self* from *process*.`links` and *process* from *self*.`links`. If *process* is not a process, exception `#(bad-arg unlink` *process*`)` is raised.

`` `(EXIT `` *p* *r* [*e*]`)` **match-extension**

matches: exit messages generated by `kill` or `link`

The extended match pattern `` `(EXIT `` *p* *r* [*e*]`)` matches exit messages generated by `kill` or `link`. The pattern *p* is matched against the *process* in the message. If the *reason* in the message is

pattern	matches
symbol	itself
number	itself
boolean	itself
character	itself
string	itself
bytevector	itself
`()`	itself
`(`p_1 `.` p_2`)`	a pair whose car matches p_1 and cdr matches p_2
`#(`p_1 ... p_n`)`	a vector of n elements whose elements match p_1 ... p_n
`#!eof`	a datum satisfying `eof-object?`
`,_`	any datum
`,`*variable*	any datum and binds a fresh *variable* to it
`,@`*variable*	any datum `equal?` to the bound *variable*
`,(`*variable* `<=` *pattern*`)`	any datum that matches *pattern* and binds a fresh *variable* to it
`` `(``*type* {`,`*field* | `,@`*field* | `[`*field* *pattern*`]`} ...`)`	an instance of the tuple or native record *type*, each *field* of which is bound to fresh variable *field* or matches the corresponding *pattern*; `,@`*field* is treated as `[`*field* `,@`*field*`]`; *type* must be known at expand time
`` `(``*ext* *spec* ...`)`	as specified by `define-match-extension` for *ext*

Figure 4.1: Pattern Grammar

an exception trapped by `try` or `catch`, or a fault condition generated by `make-fault` or `make-fault/no-cc`, then the patterns r and e are matched against *reason* as if by the extended match pattern `` `(catch `` r $[e]$`)`. Otherwise, both r and e are matched against *reason* directly.

`` `(DOWN `` m p r $[e]$`)` **match-extension**

matches: down messages generated by `monitor`

The extended match pattern `` `(DOWN `` m p r $[e]$`)` matches down messages generated by `monitor`. The pattern m is matched against the *monitor* in the message. The pattern p is matched against the monitored *process* in the message. If the *reason* in the message is an exception trapped by `try` or `catch`, or a fault condition generated by `make-fault` or `make-fault/no-cc`, then the patterns r and e are matched against *reason* as if by the extended match pattern `` `(catch `` r $[e]$`)`. Otherwise, both r and e are matched against *reason* directly.

4.4.4 Messages and Pattern Matching

The pattern matching syntax of Figure 4.1 provides a concise and expressive way to match data structures and bind variables to parts. The `receive`, `match`, `match-define`, and `match-let*` macros use this pattern language. The implementation makes a structurally recursive pass over the pattern to check for duplicate pattern variables as it emits code that matches the input against the pattern left to right.

```
(match exp                                                    syntax
  (<pattern> [(guard g)] b1 b2 ...)
  ...)
```

returns: the value of the last expression *b1 b2 ...* for the matched pattern

The `match` macro evaluates *exp* once and tests its value *v* against each pattern and optional guard. Each guard expression *g* is evaluated in the scope of its associated pattern variables. When *g* returns `#f`, *v* fails to match that clause. For the first pattern and guard that matches *v*, the expressions *b1 b2 ...* are evaluated in the scope of its pattern variables. If *v* fails to match all patterns, exception `#(bad-match` *v src*`)` is raised, where *src* is the source location of the `match` clause if available.

See Figure 4.1 for the pattern grammar.

```
(match-define <pattern> exp)                                  syntax
```

expands to: see below

The `match-define` macro evaluates *exp* and matches the resulting input against the pattern. Pattern-variable bindings are established via `define` and inhabit the same scope in which the `match-define` form appears. The `match-define` macro does not support guard expressions. If the pattern fails to match, exception `#(bad-match` *v src*`)` is raised, where *v* is the datum that failed to match the pattern at source location *src* if available.

See Figure 4.1 for the pattern grammar.

```
(match-let* ([<pattern> [(guard g)] exp]                      syntax
             ...)
  b1 b2 ...)
```

returns: the value of the last expression *b1 b2 ...*

The `match-let*` macro evaluates each *exp* in the order specified and matches its value against its pattern and guard. The pattern variables of each clause extend the scope of its guard expression *g* and all subsequent pattern clauses and body expressions *b1 b2 ...*. The `match-let*` macro returns the value of the last body expression. If any pattern fails to match or any *g* returns `#f`, exception `#(bad-match` *v src*`)` is raised, where *v* is the datum that failed to match the pattern or guard at source location *src* if available.

See Figure 4.1 for the pattern grammar.

```
(receive                                                      syntax
  [(after timeout t1 t2 ...) | (until time t1 t2 ...)]
  (<pattern> [(guard g)] b1 b2 ...)
  ...)
```

returns: the value of the last evaluated expression

The `receive` macro examines each message *m* in the calling process's `inbox` by testing it against each pattern and optional guard. Each guard expression *g* is evaluated in the scope of its associated pattern variables. When *g* returns `#f`, *m* fails to match that clause. For the first pattern and guard that matches *m*, *m* is removed from `inbox`, and the expressions *b1 b2 ...* are evaluated in the scope of its pattern variables. If *m* fails to match all patterns, the examination continues with the next message in `inbox`. When all messages have been examined, the calling process waits with its `src` field set to the source location of the `receive` macro if available. The process awakens when a new message or the time specified by the optional `after` or `until` clause arrives. If a new message arrives before the timeout, the examination process continues as before. Otherwise, the timeout expressions *t1 t2 ...* are evaluated.

The optional `after` clause specifies a *timeout* in milliseconds from the time at which control enters the `receive` macro. Similarly, the optional `until` clause specifies a clock *time* in milliseconds as measured by `erlang:now`. In addition, *timeout* and *time* can be `infinity` to indicate no timeout. If t = *timeout* or *time* is not a non-negative exact integer or `infinity`, exception `#(timeout-value` *t src*`)` is returned, where *src* is the source location of the `receive` macro if available.

See Figure 4.1 for the pattern grammar.

`(define-match-extension` *ext handle-object* [*handle-field*]`)` **syntax**

expands to: see below

The `define-match-extension` macro attaches a property to the identifier *ext*, via `define-property`, so that the expander calls *handle-object* to translate `` `( ``*ext spec ...*`)` patterns when generating code for `match`, `match-define`, `match-let*`, or `receive`. The *handle-object* procedure takes two arguments: *v*, an identifier that will be bound in the generated code to the value to be matched, and *pattern*, a syntax object for an expression of the form `` `( ``*ext spec ...*`)`. The *handle-object* procedure can return `#f` to report an invalid *pattern*. Otherwise, *handle-object* should translate the given *pattern* to a list of one or more instructions in the following simple language:

`(bind` *v e*`)`	binds *v* to the value of *e* via `let` or `define`
`(guard` *g*`)`	rejects the match if *g* evaluates to `#f`
`(sub-match` *e pattern*`)`	matches the value of *e* against *pattern*
`(handle-fields` *input field-spec ...*`)`	invokes *handle-field* to translate each *field-spec*

The generated code evaluates the instructions in the order they are returned. For example, a `guard` expression may refer to a binding established by a `bind` earlier in the list of instructions. The `sub-match` and `handle-fields` instructions are processed at expand time and may appear only as the final instruction in the list returned by *handle-object*.

The `(handle-fields` *input field-spec ...*`)` instruction parses each *field-spec* from left to right and calls *handle-field* with five arguments: the *input* from the instruction, the *field* identified, the *var* that should be bound to the value of *field*, a list of *options* appearing in the *field-spec*, and the original pattern *context*. The following table shows how each *field-spec* is parsed into arguments for *handle-field*:

field-spec	*field*	*var*	*options*	notes
`,`*field*	*field*	*field*	`()`	*field* must be an identifier
`,@`*field*	*field*	*unique*	`()`	*field* must be an identifier
`[`*field pattern option ...*`]`	*field*	*unique*	`(`*option ...*`)`	*unique* is matched against *pattern*

The *handle-field* procedure can return `#f` to report an invalid *field*. Otherwise, *handle-field* should return a list of `bind` or `guard` instructions that bind *var* and perform any checks needed to confirm a match. The resulting instructions are evaluated in the order they are returned.

Where temporaries are introduced in the generated output, the *handle-object* and *handle-field* procedures should use `with-temporaries` to avoid unintended variable capture.

(`send` *destination* *message*) **procedure**
returns: unspecified

The `send` procedure sends *message* to a process or registered name, *destination*. If *destination* is not a process or registered name, exception `#(bad-arg send` *destination*`)` is raised. If *destination* has exited, nothing else happens. Otherwise, *message* is added to the end of *destination*.`inbox`. If *destination* is sleeping, it is awakened. If *destination* is not blocked for I/O and not on the run queue, it is placed on the run queue with precedence 0.

4.4.5 Process Properties

(`pps` [*op*]) **procedure**
returns: unspecified

The `pps` procedure prints information about all processes to textual output port *op*, which defaults to the current output port. If *op* is not an output port, exception `#(bad-arg pps` *op*`)` is raised.

(`process?` *x*) **procedure**
returns: a boolean

The `process?` procedure determines whether or not the datum *x* is a process.

(`process-id` [*process*]) **procedure**
returns: the process id

The `process-id` procedure returns *process*.`id`, where *process* defaults to `self`. If *process* is not a process, exception `#(bad-arg process-id` *process*`)` is raised.

(`global-process-id` *process*) **procedure**
returns: a string; see below

The `global-process-id` procedure returns a string of the form `"`*session-id*`:`*process-id*`"`. The *process-id* is the `process-id` of *process*. If the `log-db` gen-server has been started, then *session-id* is an integer session identifier that is unique in that database instance. Otherwise, *session-id* is omitted. If *process* is not a process, exception `#(bad-arg global-process-id` *process*`)` is raised.

(`process-name` [*process*]) **procedure**
returns: the process name or `#f`

The `process-name` procedure returns *process*.`name`, where *process* defaults to `self`. If *process* is not a process, exception `#(bad-arg process-name` *process*`)` is raised.

(`process-parent`) **procedure**
returns: the parent process or `#f`

The `process-parent` procedure returns the process which spawned the current process or `#f` if the parent process has been garbage collected.

`process-trap-exit` **parameter**

value: boolean

The `process-trap-exit` parameter specifies whether or not the calling process traps exit signals as messages. Processes start with this parameter set to `#f`.

`self` **syntax**

returns: the current process

The `self` macro uses `identifier-syntax` to expand into code that retrieves the global self variable's top-level value. The global variable cannot be used directly because library bindings are immutable.

4.4.6 Miscellaneous

`(add-finalizer` *finalizer*`)` **procedure**

returns: unspecified

The `add-finalizer` procedure adds *finalizer* to the global list of finalizers. *finalizer* is a procedure of no arguments that runs in the finalizer process after garbage collections. If *finalizer* is not a procedure, exception `#(bad-arg add-finalizer` *finalizer*`)` is raised.

`(make-foreign-handle-guardian` *type-name get-handle set-handle! get-create-time close-handle print*`)` **procedure**

returns: a procedure that expects *r* and *handle*

The `make-foreign-handle-guardian` procedure assists in managing handles returned by foreign procedures. A common pattern is to wrap each foreign handle in a record, register the record with a guardian, and add a finalizer that closes the handles of records the guardian identifies as inaccessible. The `make-foreign-handle-guardian` procedure supports this pattern by creating a guardian and installing a suitable finalizer via `add-finalizer`. It also adds *type-name* to the set of foreign types recognized by `count-foreign-handles`, `foreign-handle-count`, `foreign-handle-print`, and `print-foreign-handles` to help monitor foreign handles that are still open.

The *type-name* must be a symbol; the remaining arguments must be procedures. It is an error if *type-name* has already been registered. The *get-handle* and *get-create-time* procedures should expect a single argument *r*; *get-create-time* must return an integer. The *set-handle!* procedure should expect *r* and *handle* and store *handle* in *r*, where it can be retrieved by *get-handle*.

The *close-handle* procedure runs in the finalizer process. It should take *r* and disable interrupts before checking whether `(`*get-handle r*`)` is already `#f`, since the guardian may return inaccessible objects that are already closed. If not, it should close the foreign handle and call the foreign-handle guardian with *r* and `#f` to clear the association of *r* with *handle*.

The *print* procedure should expect a textual output port *op*, *r*, and *handle* and print diagnostic information about the *handle* to *op* followed by a newline. See the output of `print-foreign-handles` for examples.

The `make-foreign-handle-guardian` procedure returns a procedure that expects two arguments: *r* and a *handle* that is either an integer or `#f`. Other code should not call *set-handle!* but should instead call the resulting procedure with interrupts disabled. If *handle* is `#f`, this procedure calls `(`*set-handle! r* `#f)` and removes *r* from the weak eq-hashtable consulted by `foreign-handle-count`, etc. If *handle* is not `#f`, then the procedure checks whether `(`*get-handle r*`)` is `#f`. If so, it calls `(`*set-handle! r handle*`)`. Otherwise, it registers *r* with the guardian. In either case, it adds *r* to the weak eq-hashtable consulted by `foreign-handle-count`, etc. To ensure that *r* is registered with the guardian just once, *r* should already contain *handle* the first time the procedure is called with *r* and a *handle* that is not `#f`.

`(count-foreign-handles` *obj report-count*`)` **procedure**
returns: *obj*

The `count-foreign-handles` procedure takes an arbitrary *obj* and a procedure *report-count* that takes *obj*, a symbol *type* identifying a source of foreign handles, and *count*, the number of open handles of that type. The `count-foreign-handles` procedure calls *report-count* once for each foreign-handle type registered with `make-foreign-handle-guardian`, in an unspecified order.

`(foreign-handle-count` *type-name*`)` **procedure**
returns: a procedure

The `foreign-handle-count` procedure takes a symbol *type-name* that must already have been registered with `make-foreign-handle-guardian` and returns a procedure of no arguments that returns the number of open foreign handles of that type.

`(foreign-handle-print` *type-name*`)` **procedure**
returns: a procedure

The `foreign-handle-print` procedure takes a symbol *type-name* that must already have been registered with `make-foreign-handle-guardian` and returns a procedure that prints information about open handles of that type. The procedure returned uses the *print* and *get-create-time* procedures registered with `make-foreign-handle-guardian`. The procedure takes an optional textual output port *op*, which defaults to the current output port, and calls *print* for each open handle of the designated type in increasing order of the creation time from *get-create-time* and then by increasing handle.

`(print-foreign-handles` [*op*]`)` **procedure**
returns: unspecified

The `print-foreign-handles` procedure prints information about open foreign handles to textual output port *op*, which defaults to the current output port. The `print-foreign-handles` procedure calls the procedure returned by `(foreign-handle-print` *type-name*`)` for each foreign-handle type registered with `make-foreign-handle-guardian`.

`(arg-check` *who* `[`*arg pred* `...] ...)` **syntax**
expands to:

```
(let ([who who])
  (let ([arg arg])
```

```
    (unless (and (pred arg) ...)
      (profile-me-as arg-check)
      (bad-arg who arg)))
  ...
  (void))
```

The `arg-check` macro raises a `bad-arg` exception if any *arg* fails any *pred* specified for that *arg*. Within coverage reports, profile counts on the `arg-check` keyword indicate the number of `bad-arg` cases encountered.

`(bad-arg` *who arg*`)` **procedure**
returns: never

The `bad-arg` procedure raises exception `#(bad-arg` *who arg*`)`.

`(complete-io` *process*`)` **procedure**
returns: unspecified

The `complete-io` procedure is used in callback functions to unblock a *process* from a call to `wait-for-io`. If *process* is not a process, exception `#(bad-arg complete-io` *process*`)` is raised.

`(console-event-handler` *event*`)` **procedure**
returns: unspecified

The `console-event-handler` procedure prints an *event* to the console error port. It is used when the event manager is not available. It disables interrupts so that it can be called from multiple processes safely. The output is designed to be machine readable. The output looks like this:

```
Date: Fri Aug 06 11:54:59 2010
Timestamp: 1281110099144
Event: event
```

The date is the local time from the `date-and-time` procedure, the timestamp is the clock time from `erlang:now`, and *event* is printed as with `write`.

`(dbg)` **procedure**
`(dbg` *id*`)`
returns: see below

The `dbg` procedure is used to debug processes that exit with a continuation condition.

`(dbg)` prints to the current output port the process id and exception message for each process that exited with a continuation condition.

`(dbg` *id*`)` enters the interactive debugger using the exception state of process *id*. If process *id* does not exist or did not exit with a continuation condition, the following message is printed: "Nothing to debug."

`(ps-fold-left` *id<? base f*`)` **procedure**
returns: see below

The `ps-fold-left` procedure folds over the process table ordered by *id<?* on `process-id` and calls *f* with the accumulator value (initially *base*) and the process for each entry in the table.

`(with-process-details` *p f*`)` **procedure**
returns: see below

The `with-process-details` procedure takes a process *p* and returns the value obtained by calling the procedure *f* with four values: the process id, the process name, the value of `erlang:now` when the process was created, and a representation of the process state that may be passed to `print-process-state`.

`(print-process-state` *state op*`)` **procedure**
returns: unspecified

The `print-process-state` procedure takes a *state* from `with-process-details` and prints a description of the process state to the textual output port *op* in the format used by `pps`.

`(dump-stack` [*op*]`)` **procedure**
`(dump-stack` *k op max-depth*`)`
returns: unspecified

The `dump-stack` procedure calls `walk-stack` to print information about the stack to textual output port *op*, which defaults to the current output port.

k is a continuation, and *max-depth* is either the symbol `default` or a positive fixnum. See `walk-stack` for details on the *max-depth* argument.

`(dump-stack` *op*`)` calls `(call/cc (lambda (`*k*`) (dump-stack` *k op* `'default)))`.

`(limit-stack` *e0 e1 ...*`)` **syntax**
expands to: `($limit-stack (lambda ()` *e0 e1 ...*`)` *source*`)`

The `limit-stack` macro adds a stack frame that may be recognized by `limit-stack?`. By default, `walk-stack` avoids descending below such frames. The `limit-stack` macro evaluates expressions *e0 e1 ...* from left to right and returns the values of the last expression.

`(limit-stack?` *x*`)` **procedure**
returns: see below

The `limit-stack?` procedure returns true if *x* is a continuation whose top frame is a `limit-stack` frame. Otherwise it returns `#f`.

`(walk-stack` *k base handle-frame combine* [*who max-depth truncated*]`)` **procedure**
returns: see below

The `walk-stack` procedure walks the stack of continuation *k* by calling the *handle-frame* and *combine* procedures for each stack frame until it reaches the base of the stack or a `limit-stack` frame, or depth reaches the optional *max-depth*, or the *next* argument to *combine* is not called.

The *handle-frame* procedure is called with four arguments:

description	a string describing the stack frame, e.g., `"#<continuation in g>"`
source	a source object identifying the return point or `#f`
proc-source	a source object identifying the procedure containing the return point or `#f`
vars	a list associating live free variables by name (or index) with their values

If *max-depth* is omitted or is the symbol `default`, then `walk-stack` uses the value of `walk-stack-max-depth` as *max-depth* and stops if recognizes a `limit-stack` frame. If *max-depth* is specified explicitly, then *walk-stack* does not stop at `limit-stack` frames. If `walk-stack` reaches a depth of *max-depth*, it calls the optional *truncated* procedure with *base* and *depth*. Otherwise, `walk-stack` calls the *combine* procedure with four arguments:

frame	the value returned by *handle-frame* for the current frame
base	the accumulator
depth	the zero-based depth of the current frame
next	a procedure that takes *base* and continues with the next frame

If `walk-stack` receives an invalid argument *val*, it calls `(bad-arg` *who val*`)` with the symbol `walk-stack` as the default value for the optional *who* argument. The default *truncated* procedure simply returns the value of *base* passed in.

`walk-stack-max-depth` **parameter**

returns: a nonnegative fixnum

The `walk-stack-max-depth` parameter specifies the default maximum depth to which `walk-stack` descends when the optional *max-depth* argument is omitted or is the symbol `default`.

`(exit-reason->stacks` *x*`)` **procedure**

returns: a list of continuations

The `exit-reason->stacks` procedure takes a Swish condition *x*, as created by `throw` or trapped by `try`, and returns a list of continuations recorded in *x*. The continuations are listed innermost to outermost.

`(erlang:now)` **procedure**

returns: the current clock time in milliseconds

The `erlang:now` procedure calls `osi_get_time` to return the number of milliseconds in UTC since the UNIX epoch January 1, 1970.

`(make-process-parameter` *initial* [*filter*]`)` **procedure**

returns: a process-parameter procedure

The `make-process-parameter` procedure creates a parameter procedure *p* that provides per-process, mutable storage via the `parameters` weak eq-hashtable of each process. Calling *p* with no arguments returns the current value of the parameter for the calling process, and calling *p* with one argument sets the value of the parameter for the calling process. The *filter*, if present, is a procedure of one argument that is applied to the *initial* and all subsequent values. If *filter* is not a procedure, exception `#(bad-arg make-process-parameter` *filter*`)` is raised.

The following system parameters are not process safe and have been redefined to use `make-process-parameter`: `command-line`, `command-line-arguments`, `custom-port-buffer-size`, `exit-handler`, `keyboard-interrupt-handler`. `pretty-initial-indent`, `pretty-line-length`, `pretty-maximum-lines`, `pretty-one-line-limit`, `pretty-standard-indent`, `print-brackets`, `print-char-name`, `print-gensym`, `print-graph`, `print-length`, `print-level`, `print-precision`, `print-radix`, `print-record`, `print-unicode`, `print-vector-length`, `reset-handler`, and `waiter-prompt-and-read`, `waiter-prompt-string`.

`inherited-parameters` **parameter**
value: a list of process-parameter procedures

The `inherited-parameters` parameter contains a list of process parameters whose values are propagated into spawned processes. Before creating a process, `spawn` and `spawn&link` record the current values of the `inherited-parameters` and install these values in the new process just before it calls the thunk.

`(make-inherited-parameter` *initial* [*filter*]`)` **procedure**
returns: a process-parameter procedure

The `make-inherited-parameter` procedure calls `make-process-parameter` to create a per-process parameter procedure *p* and adds *p* to the `inherited-parameters` list before returning *p*.

`(keyboard-interrupt` *process*`)` **procedure**
returns: unspecified

The `keyboard-interrupt` procedure causes *process* to call `((keyboard-interrupt-handler))` as soon as possible.

`(on-exit` *finally b1 b2 ...*`)` **syntax**
expands to:

```
(dynamic-wind
  void
  (lambda () b1 b2 ...)
  (lambda () finally))
```

The `on-exit` macro executes the body expressions *b1 b2 ...* in a dynamic context that executes the *finally* expression whenever control leaves the body.

`(profile-me)` **procedure**
returns: unspecified

The `profile-me` procedure does nothing but provide a place-holder for the system profiler to count the call site. When profiling is turned off, `(profile-me)` expands to `(void)`, and the system optimizer eliminates it.

`(profile-me-as` *form*`)` **syntax**
returns: unspecified

The `profile-me-as` macro does nothing but provide a place-holder for the system profiler to count the call site. If source information is present on *form*, the profile count for this call site is attributed to that *form*. When profiling is turned off or when source information is not present on *form*, `profile-me-as` expands to `(void)`, and the system optimizer eliminates it.

`(wait-for-io` *name*`)`	**procedure**

returns: unspecified

The `wait-for-io` procedure blocks the current process for I/O. The *name* string indicates the target of the I/O operation. To unblock the process, call `complete-io` from a callback function.

`windows?`	**syntax**

expands to: a boolean

The `windows?` macro expands to `#t` if the host is running Microsoft Windows and `#f` if not.

4.4.7 Tuples

For users of the concurrency model, a *tuple* is a container of named, immutable fields implemented as a vector whose first element is the tuple name and remaining elements are the fields. Each tuple definition is a macro that provides all tuple operations using field names only, not field indices. The macro makes it easy to copy a tuple without having to specify the fields that don't change. We decided not to use the Scheme record facility because it does not provide name-based constructors, copy operators, or convenient serialization.

`(define-tuple` *name* *field* ...`)`	**syntax**

expands to: a macro definition of *name* described below

The `define-tuple` macro defines a macro for creating, copying, identifying, and accessing tuple type *name*. *name* and *field* ... must be identifiers. No two field names can be the same. The following field names are reserved: `make`, `copy`, `copy*`, and `is?`.

`(`*name* `make [`*field* *value*`]` ...`)`	**syntax**

returns: a new instance of tuple type *name* with *field* = *value* ...

The `make` form creates a new instance of the tuple type *name*. *field* bindings may appear in any order. All fields from the tuple definition must be specified.

`(`*name* *field* *instance*`)`	**syntax**

returns: *instance.field*

The field accessor form retrieves the value of the specified *field* of *instance*. If $r = instance$ is not a tuple of type *name*, exception `#(bad-tuple` *name* *r* *src*`)` is raised, where *src* is the source location of the field accessor form if available.

`(`*name* *field*`)`	**syntax**

returns: a procedure that, given *instance*, returns *instance.field*

The `(`*name* *field*`)` form expands to `(lambda (instance) (`*name* *field* `instance))`.

`(`*name* `open` *instance* [*prefix*] `(`*field* ...`))`	**syntax**

expands to: definitions for *field* ... or *prefixfield* ... described below

The `open` form defines identifier syntax for each specified *field* so that a reference to *field* expands to `(`*name field r*`)` where *r* is the value of *instance*. If *r* is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the `open` form if avalable. The `open` form is equivalent to the following, except that it checks the tuple type only once:

```
(begin
  (define instance instance)
  (define-syntax field (identifier-syntax (name field instance)))
  ...)
```

The `open` form introduces definitions only for fields listed explicitly in `(`*field* ...`)`. If the optional *prefix* identifier is supplied, `open` produces a definition for *prefixfield* rather than *field* for each *field* specified.

`(`*name* `copy` *instance* `[`*field value*`]` ...`)` **syntax**

returns: a new instance of tuple type *name* with *field* = *value* ... and remaining fields copied from *instance*

The `copy` form creates a copy of *instance* except that each specified *field* is set to the associated *value*. If *r* = *instance* is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the `copy` form if avalable. *field* bindings may appear in any order.

`(`*name* `copy*` *instance* `[`*field value*`]` ...`)` **syntax**

returns: a new instance of tuple type *name* with *field* = *value* ... and remaining fields copied from *instance*

The `copy*` form is like `copy` except that, within the *value* expressions, each specified *field* is bound to an identifier macro that returns the value of *instance.field*. If *r* = *instance* is not a tuple of type *name*, exception `#(bad-tuple` *name r src*`)` is raised, where *src* is the source location of the `copy*` form if avalable. The `copy*` form is equivalent to the following, except that it checks the tuple type only once:

```
(let ([instance instance])
  (name open instance (field ...))
  (name copy instance [field value] ...))
```

`(`*name* `is?` *x*`)` **syntax**

returns: a boolean

The `is?` form determines whether or not the datum *x* is an instance of tuple type *name*.

4.4.8 I/O

`(binary->utf8` *bp*`)` **procedure**

returns: a transcoded textual port wrapping *bp*

The `binary->utf8` procedure takes a binary port *bp* and returns a textual port wrapping *bp* using `transcoded-port` and `(make-utf8-transcoder)`. The original port *bp* is marked closed so that it cannot be used except through the associated textual port.

`(close-osi-port` *port*`)` **procedure**

returns: unspecified

The `close-osi-port` procedure closes osi-port *port* using `osi_close_port`. If *port* has already been closed, `close-osi-port` does nothing.

`(close-path-watcher` *watcher*`)` **procedure**
returns: unspecified

The `close-path-watcher` procedure uses `osi_close_path_watcher` to close the given path *watcher*. If *watcher* is not a path watcher, exception `#(bad-arg close-path-watcher` *watcher*`)` is raised. If *watcher* has already been closed, `close-path-watcher` does nothing.

`(close-tcp-listener` *listener*`)` **procedure**
returns: unspecified

The `close-tcp-listener` procedure closes a TCP *listener* using `osi_close_tcp_listener`. If *listener* is not a TCP listener, exception `#(bad-arg close-tcp-listener` *listener*`)` is raised. If *listener* has already been closed, `close-tcp-listener` does nothing.

`(connect-tcp` *hostname port-spec*`)` **procedure**
returns: two values: a binary input port and a binary output port

The `connect-tcp` procedure calls `osi_connect_tcp` and blocks while the TCP connection to *hostname* on *port-spec* is established or fails to be established. The *port-spec* may be a port number or a string service name such as "http". The procedure returns a custom binary input port that reads from the new connection and a custom binary output port that writes to the new connection. These ports support `port-position` but not `set-port-position!`, and the underlying osi-ports are registered with the osi-port guardian.

If `osi_connect_tcp` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error "[`*hostname*`]:`*port-spec*`"` *who errno*`)` is raised. If *hostname* is not a string, exception `#(bad-arg connect-tcp` *hostname*`)` is raised. If *port-spec* is not a fixnum between 0 and 65535 inclusive or a string, exception `#(bad-arg connect-tcp` *port-spec*`)` is raised.

`(directory?` *path*`)` **procedure**
returns: a boolean

The `directory?` procedure calls `(get-stat` *path*`)` to determine whether or not *path* is a directory.

`(force-close-output-port` *op*`)` **procedure**
returns: unspecified

The `force-close-output-port` procedure is used to close an output port, even if it has unflushed output that would otherwise cause it to fail to close. If *op* is not already closed, `force-close-output-port` tries to close it with `(close-output-port` *op*`)`. If it fails, the output buffer is cleared with `(clear-output-port` *op*`)`, and `(close-output-port` *op*`)` is called again.

`(get-datum/annotations-all` *ip sfd bfp*`)` **procedure**
returns: a list of annotated objects

The `get-datum/annotations-all` procedure takes a textual input port *ip*, a source-file descriptor *sfd*, and an exact nonnegative integer *bfp* representing the character position of the next character to be read from *ip*. The procedure returns a list of the annotated objects, in order, obtained by repeatedly calling `get-datum/annotations` with the advancing *bfp*, until *ip* reaches the end of file.

`(get-file-size` *port*`)` **procedure**

returns: the number of bytes in the file associated with osi-port *port*

The `get-file-size` procedure calls `osi_get_file_size` to return the number of bytes in the file associated with osi-port *port*.

If `osi_get_file_size` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *filename who errno*`)` is raised.

`(get-real-path` *path*`)` **procedure**

returns: the canonicalized absolute pathname of *path*

The `get-real-path` procedure calls `osi_get_real_path` and returns the canonicalized absolute pathname of *path*.

`(get-source-offset` *ip*`)` **procedure**

returns: an exact nonnegative integer

The `get-source-offset` procedure takes a binary input port *ip* that supports `port-position`, skips over the `#!`*interpreter-directive* line, if any, and returns the resulting `port-position`.

`(get-stat` *path* [*follow?*]`)` **procedure**

returns: a `<stat>` tuple

The `get-stat` procedure calls `osi_get_stat` and returns the `<stat>` tuple for *path*, following a symbolic link unless *follow?* is `#f`. If `osi_get_stat` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(hook-console-input)` **procedure**

returns: unspecified

The `hook-console-input` procedure replaces the system console input port, which uses synchronous I/O, with a custom textual input port that uses asynchronous I/O. It builds a custom binary input port with `osi_get_stdin`, wraps it with `binary->utf8`, and sets the result as the `console-input-port`, `current-input-port`, and the system internal `$console-input-port`. It does nothing after it has been called once.

`(io-error` *name who errno*`)` **procedure**

returns: never

The `io-error` procedure raises exception `#(io-error` *name who errno*`)`. The string *name* identifies the port. The symbol *who* specifies the procedure that raised an error, and the number *errno* specifies the error code. The `read-osi-port` procedure raises this exception with *who*=`osi_read_port`, and the `write-osi-port` procedure raises it with *who*=`osi_write_port`.

`(list-directory` *path*`)` **procedure**

returns: `((`*name* `.` *type*`) ...)`

The `list-directory` procedure calls `osi_list_directory` and returns `((`*name* `.` *type*`) ...)`, the list of directory entries of *path*. It does not include "." and "..". *name* is the string name of the directory entry, and *type* is one of the following constants:

`DIRENT_UNKNOWN`	`DIRENT_FILE`	`DIRENT_DIR`	`DIRENT_LINK`	`DIRENT_FIFO`
`DIRENT_SOCKET`	`DIRENT_CHAR`	`DIRENT_BLOCK`		

If `osi_list_directory` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path* *who* *errno*`)` is raised.

`(listen-tcp` *address* *port-number* *process*`)` **procedure**

returns: a TCP listener

The `listen-tcp` procedure calls `osi_listen_tcp` to create a TCP listener on the given *address* and *port-number* and returns a TCP listener that is registered with the listener guardian.

For each accepted connection, the message `#(accept-tcp` *listener* *ip* *op*`)` is sent to *process*, where *ip* is the custom binary input port and *op* is the custom binary output port. Both ports support `port-position` but not `set-port-position!`.

For each failed connection, the message `#(accept-tcp-failed` *listener* *who* *errno*`)` is sent to *process*, where *who* and *errno* specify the error.

The *address* is a dotted quad IPv4 address or an IPv6 address. Use `"::"` to listen on all IPv4 and IPv6 interfaces. Use `"0.0.0.0"` to listen on all IPv4 interfaces. Otherwise, it listens on the given *address* only. If *address* is not a string, exception `#(bad-arg listen-tcp` *address*`)` is raised.

If *port-number* is zero, the operating system will choose an available port number, which can be queried with `listener-port-number`. If *port-number* is not a fixnum between 0 and 65535 inclusive, exception `#(bad-arg listen-tcp` *port-number*`)` is raised.

If `osi_listen_tcp` fails with error pair `(`*who* `.` *errno*`)`, exception `#(listen-tcp-failed` *address* *port-number* *who* *errno*`)` is raised.

`(listener-address` *listener*`)` **procedure**

returns: the `address` field of *listener*

The `listener-address` procedure returns the `address` of the given TCP *listener*.

`(listener-create-time` *listener*`)` **procedure**

returns: a clock time in milliseconds

The `listener-create-time` procedure returns the clock time from `erlang:now` when the given TCP *listener* was created.

`(listener-port-number` *listener*`)` **procedure**

returns: the `port-number` field of *listener*

The `listener-port-number` procedure returns the `port-number` of the given TCP *listener*.

`(listener?` *x*`)` **procedure**

returns: a boolean

The `listener?` procedure determines whether or not the datum *x* is a TCP listener.

`(make-directory` *path* [*mode*]`)` **procedure**
returns: unspecified

The `make-directory` procedure calls `osi_make_directory` to make directory *path* with *mode*, which defaults to `#o777`.

If `osi_make_directory` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(make-directory-path` *path* [*mode*]`)` **procedure**
returns: *path*

The `make-directory-path` procedure creates directories as needed for the file *path* using *mode*, which defaults to `#o777`. It returns *path*.

`(make-osi-input-port` *p*`)` **procedure**
returns: a binary input port

The `make-osi-input-port` procedure returns a custom binary input port that reads from osi-port *p* and supports `port-position` but not `set-port-position!`. Closing the input port closes the underlying osi-port *p*.

`(make-osi-output-port` *p*`)` **procedure**
returns: a binary output port

The `make-osi-output-port` procedure returns a custom binary output port that writes to osi-port *p* and supports `port-position` but not `set-port-position!`. Closing the output port closes the underlying osi-port *p*.

`(make-utf8-transcoder)` **procedure**
returns: a UTF-8 transcoder

The `make-utf8-transcoder` procedure creates a UTF-8 transcoder with end-of-line style `none` and error-handling mode `replace`.

`(open-fd-port` *name fd close?*`)` **procedure**
returns: an osi-port

The `open-fd-port` procedure creates an osi-port with the given *name* by calling `osi_open_fd` with *fd* and *close?*. The osi-port is registered with the osi-port guardian. When the osi-port is closed, the underlying file descriptor *fd* is closed if and only if *close?* is not `#f`. When $0 \leq fd \leq 2$, *close?* must be `#f` for the standard I/O file descriptor.

`(open-file` *name flags mode type*`)` **procedure**
returns: a custom file port

The `open-file` procedure creates a custom file port by calling `(open-file-port` *name flags mode*`)`. The custom port supports both getting and setting the file position, except when *type*=`append`. The particular type of custom port returned is determined by *type*:

- `binary-input`: a binary input port
- `binary-output`: a binary output port
- `binary-append`: a binary output port. Each write appends to the file by specifying position −1.
- `input`: a textual input port wrapping a binary input port with `binary->utf8`
- `output`: a textual output port wrapping a binary output port with `binary->utf8`
- `append`: a textual output port wrapping a binary output port with `binary->utf8`. Each write appends to the file by specifying position −1.

If *type* is any other value, exception `#(bad-arg open-file` *type*`)` is raised.

`(open-file-port` *name flags mode*`)` **procedure**
returns: an osi-port

The `open-file-port` procedure creates an osi-port by calling `osi_open_file` with *name*, *flags*, and *mode*. The osi-port is registered with the osi-port guardian.

The following constants are defined for *flags*:

O_APPEND	O_CREAT	O_DIRECT	O_DIRECTORY	O_DSYNC	O_EXCL
O_EXLOCK	O_NOATIME	O_NOCTTY	O_NOFOLLOW	O_NONBLOCK	O_RANDOM
O_RDONLY	O_RDWR	O_SEQUENTIAL	O_SHORT_LIVED	O_SYMLINK	O_SYNC
O_TEMPORARY	O_TRUNC	O_WRONLY			

The following constants are defined for *mode*:

S_IFMT S_IFIFO S_IFCHR S_IFDIR S_IFBLK S_IFREG S_IFLNK S_IFSOCK

If `osi_open_file` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *name who errno*`)` is raised.

`(open-binary-file-to-append` *name*`)` **procedure**
returns: a binary file port

The `open-binary-file-to-append` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_APPEND) #o666 'binary-append)`.

`(open-binary-file-to-read` *name*`)` **procedure**
returns: a binary file port

The `open-binary-file-to-read` procedure calls `(open-file` *name* `O_RDONLY 0 'binary-input)`.

`(open-binary-file-to-replace` *name*`)` **procedure**
returns: a binary file port

The `open-binary-file-to-replace` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_TRUNC) #o666 'binary-output)`.

`(open-binary-file-to-write` *name*`)` **procedure**
returns: a binary file port

The `open-binary-file-to-write` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_EXCL) #o666 'binary-output)`.

`(open-file-to-append` *name*`)` **procedure**

returns: a textual file port

The `open-file-to-append` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_APPEND) #o666 'append)`.

`(open-file-to-read` *name*`)` **procedure**

returns: a textual file port

The `open-file-to-read` procedure calls `(open-file` *name* `O_RDONLY 0 'input)`.

`(open-file-to-replace` *name*`)` **procedure**

returns: a textual file port

The `open-file-to-replace` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_TRUNC) #o666 'output)`.

`(open-file-to-write` *name*`)` **procedure**

returns: a textual file port

The `open-file-to-write` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_EXCL) #o666 'output)`.

`(open-utf8-bytevector` *bv*`)` **procedure**

returns: a transcoded textual input port wrapping *bv*

The `open-utf8-bytevector` procedure calls `(binary->utf8 (open-bytevector-input-port` *bv*`))`.

`(osi-port-closed?` *p*`)` **procedure**

returns: a boolean

The `osi-port-closed?` procedure determines whether or not the osi-port *p* is closed.

`(osi-port-count)` **procedure**

returns: the number of open osi-ports

The `osi-port-count` procedure returns the number of open osi-ports.

`(osi-port-create-time` *p*`)` **procedure**

returns: a clock time in milliseconds

The `osi-port-create-time` procedure returns the clock time from `erlang:now` when the osi-port *p* was created.

`(osi-port-name` *p*`)` **procedure**

returns: a string

The `osi-port-name` procedure returns the name of osi-port *p*.

`(osi-port?` *x*`)` **procedure**
returns: a boolean

The `osi-port?` procedure determines whether or not the datum *x* is an osi-port.

`(path-combine` $path_1$ $path_2$ ...`)` **procedure**
returns: the string combining the paths

The `path-combine` procedure appends one or more paths, inserting the directory-separator character between each pair of paths as needed.

`(path-watcher-count)` **procedure**
returns: the number of open path watchers

The `path-watcher-count` procedure returns the number of open path watchers.

`(path-watcher-create-time` *watcher*`)` **procedure**
returns: a clock time in milliseconds

The `path-watcher-create-time` procedure returns the clock time from `erlang:now` when the given path *watcher* was created.

`(path-watcher-path` *watcher*`)` **procedure**
returns: the `path` field of *watcher*

The `path-watcher-path` procedure returns the `path` of the given path *watcher*.

`(path-watcher?` *x*`)` **procedure**
returns: a boolean

The `path-watcher?` procedure determines whether or not the datum *x* is a path watcher.

`(print-osi-ports` [*op*]`)` **procedure**
returns: unspecified

The `print-osi-ports` procedure prints information about all open osi-ports to textual output port *op*, which defaults to the current output port.

`(print-path-watchers` [*op*]`)` **procedure**
returns: unspecified

The `print-path-watchers` procedure prints information about all open path watchers to textual output port *op*, which defaults to the current output port.

`(print-signal-handlers` [*op*]`)` **procedure**
returns: unspecified

The `print-signal-handlers` procedure prints information about all active signal handlers to textual output port *op*, which defaults to the current output port.

(`print-tcp-listeners` [*op*]) **procedure**
returns: unspecified

The `print-tcp-listeners` procedure prints information about all open TCP listeners to textual output port *op*, which defaults to the current output port.

(`read-bytevector` *name* *contents*) **procedure**
returns: a list of annotations

The `read-bytevector` procedure takes a filename *name* and *contents* bytevector and returns a list of annotations read using `get-datum/annotations` from the *contents* bytevector transcoded with `(make-utf8-transcoder)`.

(`read-file` *name*) **procedure**
returns: a bytevector with the contents of *name*

The `read-file` procedure calls (`open-file-port` *name* `O_RDONLY 0`) to open the file *name* and returns the contents as a bytevector.

(`read-osi-port` *port* *bv* *start* *n* *fp*) **procedure**
returns: the number of bytes read

The `read-osi-port` procedure calls `osi_read_port` with the handle from the given osi-port *port*, bytevector buffer *bv*, starting 0-based buffer index *start*, maximum number of bytes to read *n*, and starting 0-based file position *fp*. To specify the current position, use *fp*=−1. The calling process blocks for the I/O to complete. If the read fails with error pair (*who* . *errno*), exception `#(io-error` *name* *who* *errno*) is raised, where *name* is the name of *port*. Otherwise, the number of bytes read is returned. Error code UV_EOF (end of file) is not considered an error, and 0 is returned.

(`regular-file?` *path*) **procedure**
returns: a boolean

The `regular-file?` procedure calls (`get-stat` *path*) to determine whether or not *path* is a regular file.

(`remove-directory` *path*) **procedure**
returns: unspecified

The `remove-directory` procedure calls `osi_remove_directory` to remove directory *path*.

If `osi_remove_directory` fails with error pair (*who* . *errno*), exception `#(io-error` *path* *who* *errno*) is raised.

(`remove-file` *path*) **procedure**
returns: unspecified

The `remove-file` procedure calls `osi_unlink` to remove file *path*.

If `osi_unlink` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(rename-path` *path new-path*`)` **procedure**
returns: unspecified

The `rename-path` procedure calls `osi_rename` to rename *path* to *new-path*.

If `osi_rename` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(set-file-mode` *path mode*`)` **procedure**
returns: unspecified

The `set-file-mode` procedure calls `osi_chmod` to set the file mode of *path* to *mode*.

If `osi_chmod` fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(signal-handler` *signum* [*callback*]`)` **procedure**
returns: see below

The `signal-handler` procedure manages an internal table of global handlers for low-level signals. The *signum* argument must be a positive fixnum. If no *callback* is supplied, `signal-handler` returns the callback, if any, registered to handle that signal, or else `#f`. If the optional *callback* argument is supplied, it must be `#f` or a procedure of one argument that is called with the signal number when that signal is delivered. Since the *callback* procedure is called on the event loop, it must obey the restrictions on event-loop callbacks (see page 39). Do not call `app:shutdown` from *callback*, because it is not process-safe when no application is running. Instead, `(spawn app:shutdown)`. At startup, Swish installs handlers that call `app:shutdown` safely when certain signals are delivered. The set of signals trapped at startup depends on the platform.

This procedure is like Chez Scheme's `register-signal-handler`, except that `signal-handler` is integrated into the Swish event loop. In particular, the callback supplied to `signal-handler` can wake a sleeping process via `send`. When a signal *signum* is delivered to the Swish operating-system process, e.g., via `osi_kill`, `osi_get_callbacks` returns a list of callbacks that includes `(@deliver-signal` *signum*`)`. If *callback* has been established to handle *signum*, then `@deliver-signal` calls *callback* with *signum*.

The `(swish io)` library exports constants for the available signal numbers, which vary among platforms. For platform-specific notes on signal handling, see [21]. Some signals cannot be handled even though a handler may be established. Handling some signals may result in undefined behavior.

`(signal-handler-count)` **procedure**
returns: the number of open signal handlers

The `signal-handler-count` procedure returns the number of open signal handlers.

`(spawn-os-process` *path args process*`)` **procedure**
returns: four values: a binary output port *to-stdin*, a binary input port *from-stdout*, a binary

input port *from-stderr*, and an integer process identifier *os-pid*

The `spawn-os-process` procedure calls `osi_spawn` to spawn an operating system process with the string *path* and list of string-valued *args*. It returns a custom binary output port *to-stdin* that writes to the standard input of the process, custom binary input ports *from-stdout* and *from-stderr* that read from the standard output and standard error of the process, respectively, and a process identifier *os-pid*. These ports support `port-position` but not `set-port-position!`, and the underlying osi-ports are registered with the osi-port guardian.

When the spawned process terminates, `#(process-terminated` *os-pid exit-status term-signal*`)` is sent to *process*.

If `osi_spawn` returns error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(spawn-os-process-detached` *path args*`)` **procedure**

returns: an integer operating-system process identifier

The `spawn-os-process-detached` procedure calls `osi_spawn_detached` to spawn an operating system process with the string *path* and list of string-valued *args*. It returns an integer operating-system process identifier.

If `osi_spawn_detached` returns error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(stat-directory?` *x*`)` **procedure**

returns: a boolean

The `stat-directory?` procedure determines whether or not the datum *x* is a `<stat>` tuple for a directory.

`(stat-regular-file?` *x*`)` **procedure**

returns: a boolean

The `stat-regular-file?` procedure determines whether or not the datum *x* is a `<stat>` tuple for a regular file.

`(tcp-listener-count)` **procedure**

returns: the number of open TCP listeners

The `tcp-listener-count` procedure returns the number of open TCP listeners.

`(watch-path` *path process*`)` **procedure**

returns: a path watcher

The `watch-path` procedure calls `osi_watch_path` to track changes to *path* and returns a path watcher that is registered with the path-watcher guardian.

Every time a change is detected, `#(path-changed` *path filename events*`)` is sent to *process*, where *events* is 1 for rename, 2 for change, and 3 for rename and change. If the watcher encounters an error, `#(path-watcher-failed` *path errno*`)` is sent to *process*.

If `osi_watch_path` returns error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *path who errno*`)` is raised.

`(with-sfd-source-offset` *name handler*`)` **procedure**
returns: see below

The `with-sfd-source-offset` procedure takes a filename *name* and returns the result of calling the procedure *handler* with three arguments: *ip*, a textual port transcoded with `(make-utf8-transcoder)`, *sfd*, a source-file descriptor that refers to *name*, and *source-offset*, the value returned by `get-source-offset`. Before returning, `with-sfd-source-offset` closes the textual port.

`(write-osi-port` *port bv start n fp*`)` **procedure**
returns: the number of bytes written

The `write-osi-port` procedure calls `osi_write_port` with the handle from the given osi-port *port*, bytevector buffer *bv*, starting 0-based buffer index *start*, maximum number of bytes to write *n*, and starting 0-based file position *fp*. To specify the current position, use *fp*=−1. The calling process blocks for the I/O to complete. If the write fails with error pair `(`*who* `.` *errno*`)`, exception `#(io-error` *name who errno*`)` is raised, where *name* is the name of *port*. Otherwise, the number of bytes written is returned.

4.4.9 Queues

A queue is represented as a pair of lists, `(`*in* `.` *out*`)`. The *out* list contains the first elements of the queue, and the *in* list contains the last elements of the queue in reverse. This representation allows for O(1) amortized insertion and removal times. The implementation is based on the Erlang queue module [11].

`(queue:add` *x q*`)` **procedure**
returns: a queue that adds *x* to the rear of *q*

`(queue:add-front` *x q*`)` **procedure**
returns: a queue that adds *x* to the front of *q*

`(queue:drop` *q*`)` **procedure**
returns: a queue without the first element of *q*

`queue:empty` **syntax**
returns: the empty queue

`(queue:empty?` *q*`)` **procedure**
returns: `#t` if *q* is `queue:empty`, `#f` otherwise

`(queue:get` *q*`)` **procedure**
returns: the first element of *q*

4.4.10 Hash Tables

The implementation of functional hash tables is based on the Erlang dict module [9, 16].

(`ht:delete` *ht* *key*) **procedure**
returns: a hash table formed by dropping any association of *key* from *ht*

(`ht:fold` *ht* *f* *init*) **procedure**
returns: see below

The `ht:fold` procedure accumulates a value by applying f to each key/value association in *ht* and the accumulator, which is initially *init*. It can be defined recursively as follows, where n is the size of *ht*, and the result of `ht:fold` is F_n:

$$F_0 = init$$
$$F_i = (f\ \ key_i\ \ val_i\ \ F_{i-1}) \text{ for } 1 \le i \le n$$

(`ht:is?` *x*) **procedure**
returns: `#t` if x is a hash table, `#f` otherwise

(`ht:keys` *ht*) **procedure**
returns: a list of the keys of *ht*

(`ht:make` *hash-key* *equal-key?* *valid-key?*) **procedure**
returns: an empty hash table

The `ht:make` procedure returns an empty hash table.

The *hash-key* procedure takes a key and returns an exact integer. It must return the same integer for equivalent keys.

The *equal-key?* procedure takes two keys and returns a true value if they are equivalent and `#f` otherwise.

The *valid-key?* procedure takes a datum and returns a true value if it a valid key and `#f` otherwise.

(`ht:ref` *ht* *key* *default*) **procedure**
returns: the value associated with *key* in *ht*, *default* if none

(`ht:set` *ht* *key* *val*) **procedure**
returns: a hash table formed by associating *key* with *val* in *ht*

(`ht:size` *ht*) **procedure**
returns: the number of entries in *ht*

4.4.11 Error Strings

`current-exit-reason->english` **parameter**

value: a procedure of one argument that returns an English string

The `current-exit-reason->english` parameter specifies the conversion procedure used by `exit-reason->english`. It defaults to `swish-exit-reason->english`.

`(exit-reason->english` *x*`)` **procedure**

returns: a string in U.S. English

The `exit-reason->english` procedure converts an exit reason into an English string using the procedure stored in parameter `current-exit-reason->english`.

`(swish-exit-reason->english` *x*`)` **procedure**

returns: a string in U.S. English

The `swish-exit-reason->english` procedure converts an exit reason from Swish into an English string.

4.4.12 String Utilities

The string utilities below are found in the `(swish string-utils)` library.

`(ends-with?` *s p*`)` **procedure**

returns: a boolean

The `ends-with?` procedure determines whether or not the string *s* ends with string *p* using case-sensitive comparisons.

`(ends-with-ci?` *s p*`)` **procedure**

returns: a boolean

The `ends-with-ci?` procedure determines whether or not the string *s* ends with string *p* using case-insensitive comparisons.

`(format-rfc2822` *d*`)` **procedure**

returns: a string like "Thu, 28 Jul 2016 17:20:11 -0400"

The `format-rfc2822` procedure returns a string representation of the date object *d* in the form specified in Section 3.3 of RFC 2822 [23].

`(join` *ls separator* [*last-separator*]`)` **procedure**

returns: a string

The `join` procedure returns the string formed by displaying each of the elements of list *ls* separated by displaying *separator*. When *last-separator* is specified, it is used as the last separator.

`(oxford-comma` [*prefix*] *elt-fmt conj* [*suffix*]`)` **procedure**

returns: a string

The `oxford-comma` procedure constructs a format string for use with `errorf`, `format`, `printf`, etc., to join the elements of a list with commas and/or *conj*, as appropriate. The *elt-fmt* argument is the format string for individual items of the list. The *conj* argument is a string used to separate the final two elements of the list. The *prefix* and *suffix* arguments must be supplied together or omitted. If omitted, *prefix* defaults to `"~{"` and *suffix* defaults to `"~}"`.

(`split` *str separator*) **procedure**

returns: a list of strings

The `split` procedure divides the *str* string by the *separator* character into a list of strings, none of which contain *separator*.

(`split-n` *str separator n*) **procedure**

returns: a list of no more than *n* strings

The `split-n` procedure divides the *str* string by the *separator* character into a list of at most *n* strings. The last string may contain *separator*.

(`starts-with?` *s p*) **procedure**

returns: a boolean

The `starts-with?` procedure determines whether or not the string *s* starts with string *p* using case-sensitive comparisons.

(`starts-with-ci?` *s p*) **procedure**

returns: a boolean

The `starts-with-ci?` procedure determines whether or not the string *s* starts with string *p* using case-insensitive comparisons.

(`symbol-append .` *ls*) **procedure**

returns: a symbol

The `symbol-append` procedure returns the symbol formed by appending the symbols passed as arguments.

4.4.13 Message Digests

(`make-digest-provider` *name open hash! get-hash close*) **procedure**

returns: a digest provider record

The `make-digest-provider` procedure takes a symbol *name* and a set of procedures and returns a new digest provider that can be used as the value of `current-digest-provider` or as an argument to `open-digest`.

The *open* procedure takes a string *alg* and an *hmac-key* that is either `#f` or a bytevector to use for HMAC keyed hashing. If the digest provider does not support the specified message-digest algorithm *alg* or the *hmac-key*, it should return an error pair or one of the symbols `algorithm` or `hmac-key` to indicate which argument is invalid. Otherwise it should return a message-digest context that can be passed to *hash!*, *get-hash*, and *close*.

The *hash!* procedure has the same interface as `osi_hash_data` and performs the analogous function for the message-digest context initialized by *open*. It computes the message digest incrementally on the set of bytes specified and updates the message-digest context.

The *get-hash* procedure has the same interface as `osi_get_SHA1` and performs the analogous function for the message-digest context initialized by *open* and updated by *hash!*.

The *close* procedure frees a message-digest context initialized by *open*.

`current-digest-provider` **parameter**

returns: a digest provider record

The `current-digest-provider` parameter specifies the digest provider used by `bytevector->hex-string` and by `open-digest` when the digest provider is not explicit.

`default-digest-provider` **binding**

returns: the default digest provider record

The default value of `current-digest-provider` is bound to `default-digest-provider`. The default digest provider supports only the SHA1 message-digest algorithm; it does not support HMAC keyed hashing.

`(digest-provider-name` *dp*`)` **procedure**

returns: the name of the digest provider

The `digest-provider-name` procedure returns the symbol that was supplied to `make-digest-provider` when *dp* was created.

`(open-digest` *alg* [*hmac-key* [*dp*]]`)` **procedure**

returns: a message digest

The `open-digest` procedure takes a string or symbol *alg* naming a message-digest function supported by the digest provider *dp*, which defaults to the value of `current-digest-provider`. If *alg* is a symbol, it is converted to an upper-case string before proceeding. The optional *hmac-key* may be `#f` to disable HMAC keyed hashing; otherwise it must be a bytevector or a string. If *hmac-key* is a string, it is converted to a bytevector using `string->utf8`.

The `open-digest` procedure disables interrupts while it calls the *open* procedure that was registered with `make-digest-provider`, passing it the algorithm name as a string and the *hmac-key* as either `#f` or a bytevector. If successful, it wraps the message-digest context returned by *open* in a message-digest record *md*, registers *md* with a foreign-handle guardian using the type name `digests`, and returns *md*.

`(hash!` *md* *bv* [*start-index* [*size*]]`)` **procedure**

returns: unspecified

The `hash!` procedure disables interrupts while it calls the *hash!* procedure of the message-digest provider used in the `open-digest` call that returned *md*. The *hash!* procedure computes the message digest incrementally on the set of *size* bytes in the bytevector *bv* starting at the zero-based *start-index*. If omitted, *start-index* defaults to zero and *size* defaults to the size of *bv*. The *hash!* procedure updates the message-digest context within *md*.

`(get-hash md)` **procedure**

returns: a bytevector

The `get-hash` procedure disables interrupts while it calls the *get-hash* procedure of the message-digest provider used in the `open-digest` call that returned *md*. It returns a bytevector containing the message digest accumulated in *md* by zero or more calls to `hash!`.

`(close-digest md)` **procedure**

returns: unspecified

The `close-digest` procedure disables interrupts, unregisters *md* with the foreign-handle guardian, calls the *close* procedure of the message-digest provider used in the `open-digest` call that returned *md*, then enables interrupts.

`(hash->hex-string bv)` **procedure**

returns: a string of lower-case hexadecimal digits

The `hash->hex-string` procedure takes a bytevector *bv* and returns the unsigned bytes in *bv* as a string of lower-case hexadecimal digits.

`(hex-string->hash s)` **procedure**

returns: a bytevector

The `hex-string->hash` procedure takes a string *s* containing an even number of hexadecimal digits and returns a bytevector half that size containing the unsigned bytes specified by adjacent pairs of hexadecimal digits.

`(bytevector->hex-string bv alg [block-size])` **procedure**

returns: a string of lower-case hexadecimal digits

The `bytevector->hex-string` procedure takes a bytevector *bv* and a string or symbol *alg* naming a message-digest function supported by the `current-digest-provider` and returns the message digest of *bv* using *alg* as a string of lower-case hexadecimal digits. To keep the event loop responsive, `bytevector->hex-string` computes the message digest of *bv* incrementally in chunks of *block-size*, which defaults to 16384. When the size of *bv* is not more than *block-size*, this is functionally equivalent to the following:

```
(let ([md (open-digest alg)])
  (on-exit (close-digest md)
    (hash! md bv)
    (hash->hex-string (get-hash md))))
```

`(print-digests [op])` **procedure**

returns: unspecified

The `print-digests` procedure prints information about all open message-digest contexts to textual output port *op*, which defaults to the current output port. This is the procedure returned by `(foreign-handle-print 'digests)`.

`(digest-count)` **procedure**
returns: the number of open message-digest contexts

The `digest-count` procedure returns the number of open message-digest contexts. This is the procedure returned by `(foreign-handle-count 'digests)`.

4.4.14 Data-Encoding Utilities

`(base64-decode-bytevector` *bv*`)` **procedure**
`(base64url-decode-bytevector` *bv*`)`
returns: a bytevector

The `base64-decode-bytevector` and `base64url-decode-bytevector` procedures return a new bytevector containing the data decoded from bytevector *bv*. The data in *bv* must be in the form described in Section 4 or 5, respectively, of IETF RFC 4648 [24]. In keeping with Sections 3.1 and 3.3 of [24], line feeds and non-alphabetic characters are not permitted in *bv* and should be removed before calling these procedures.

`(base64-encode-bytevector` *bv*`)` **procedure**
`(base64url-encode-bytevector` *bv*`)`
returns: a bytevector

The `base64-encode-bytevector` and `base64url-encode-bytevector` procedures return a new bytevector containing the binary data from bytevector *bv* encoded as printable US-ASCII characters as described in Sections 4 and 5, respectively, of IETF RFC 4648 [24]. Both procedures encode data using an alphabet including `A-Z`, `a-z`, and `0-9`. For `base64-encode-bytevector`, the alphabet also includes `+` and `/`. For `base64url-encode-bytevector` it includes `-` and `_`. In keeping with Section 3.1 of [24], these procedures do not introduce line breaks in the output.

4.4.15 Macro Utilities

`(pretty-syntax-violation` *msg form* [*subform* [*who*]]`)` **procedure**
returns: never

The `pretty-syntax-violation` procedure raises a syntax violation. It differs from the native `syntax-violation` in that it formats *form* and *subform* using `pretty-format` abbreviations, and it does not attempt to infer a who condition when *who* is not provided, as this can produce confusing results in error messages involving match patterns. To provide more readable exception messages, it constructs the formatted message condition by calling `pretty-print` before raising the exception, and it prevents `display-condition` from formatting the `&syntax` condition within the compound condition it constructs.

`(with-temporaries (`*id* `...) e0 e1 ...)` **syntax**
expands to:

```
(with-syntax ([(id ...) (generate-temporaries '(id ...))])
  e0 e1 ...)
```

The `with-temporaries` macro binds each macro-language pattern variable *id* to a fresh generated identifier within the body `(begin` *e0 e1* `...)`.

Chapter 5

Regular Expressions

5.1 Introduction

The regular expressions library `(swish pregexp)` is a derivative of pregexp: Portable Regular Expressions for Scheme and Common Lisp [25]. It provides regular expressions modeled on Perl's [15, 27] and includes such powerful directives as numeric and non-greedy quantifiers, capturing and non-capturing clustering, POSIX character classes, selective case- and space-insensitivity, back-references, alternation, backtrack pruning, positive and negative look-ahead and look-behind, in addition to the more basic directives familiar to all regexp users.

A *regexp* is a string that describes a pattern. A regexp matcher tries to *match* this pattern against (a portion of) another string, which we will call the *text string*. The text string is treated as raw text and not as a pattern.

Most of the characters in a regexp pattern are meant to match occurrences of themselves in the text string. Thus, the pattern `"abc"` matches a string that contains the characters `a`, `b`, `c` in succession.

In the regexp pattern, some characters act as *metacharacters*, and some character sequences act as *metasequences*. That is, they specify something other than their literal selves. For example, in the pattern `"a.c"`, the characters `a` and `c` do stand for themselves but the *metacharacter* '`.`' can match *any* character (other than newline). Therefore, the pattern `"a.c"` matches an `a`, followed by *any* character, followed by a `c`.

If we needed to match the character '`.`' itself, we *escape* it, i.e., precede it with a backslash (\). The character sequence `\.` is thus a *metasequence*, since it doesn't match itself but rather just '`.`'. So, to match `a` followed by a literal '`.`' followed by `c`, we use the regexp pattern `"a\\.c"`.[1] Another example of a metasequence is `\t`, which is a readable way to represent the tab character.

We will call the string representation of a regexp the *U-regexp*, where *U* can be taken to mean *Unix-style* or *universal*, because this notation for regexps is universally familiar. Our implementation uses an intermediate tree-like representation called the *S-regexp*, where *S* can stand for *Scheme*, *symbolic*, or *s-expression*. S-regexps are more verbose and less readable than U-regexps, but they are much easier for Scheme's recursive procedures to navigate.

[1] The double backslash is an artifact of Scheme strings, not the regexp pattern itself. When we want a literal backslash inside a Scheme string, we must escape it so that it shows up in the string at all. Scheme strings use backslash as the escape character, so we end up with two backslashes.

5.2 Programming Interface

`(pregexp` *regexp*`)` **procedure**

returns: an S-regexp

The `pregexp` procedure takes a U-regexp string *regexp* and returns an S-regexp.

`(re` *regexp*`)` **syntax**

expands to: `(pregexp` *regexp*`)`

If *regexp* is a literal string, the `re` macro expands to the result of evaluating `(pregexp` *regexp*`)` at expand time. Otherwise it expands into a run-time call to `pregexp`.

`(pregexp-match-positions` *pat str* [*start* [*end*]]`)` **procedure**

returns: `((`*s* `.` *e*`) ...)` or `#f`

The `pregexp-match-positions` procedure takes a regexp pattern *pat* and a text string *str* and returns a *match* if the regexp matches (some part of) the text string between the inclusive *start* index (defaults to 0) and the exclusive *end* index (defaults to the length of *str*).

The regexp may be either a U- or an S-regexp. `pregexp-match-positions` will internally compile a U-regexp to an S-regexp before proceeding with the matching. If you find yourself calling `pregexp-match-positions` repeatedly with the same U-regexp, it may be advisable to explicitly convert the latter into an S-regexp once beforehand, using `pregexp`, to save needless recompilation.

`pregexp-match-positions` returns a list of *index pairs* if the regexp matches the string and `#f` if it does not match. Index pair `(`*s* `.` *e*`)` gives the inclusive starting index *s* and exclusive ending index *e* of the matching substring with respect to *str*. The first index pair indicates the entire match, and subsequent pairs indicate submatches. Some of the submatches may be `#f`.

`(pregexp-match` *pat str* [*start* [*end*]]`)` **procedure**

returns: list of matching substrings or `#f`

The `pregexp-match` procedure is called like `pregexp-match-positions`, but instead of returning index pairs, it returns the matching substrings. The first substring is the entire match, and subsequent substrings are submatches, some of which may be `#f`.

`(pregexp-split` *pat str*`)` **procedure**

returns: list of substrings from *str*

The `pregexp-split` procedure takes two arguments, a regexp pattern *pat* and a text string *str*, and returns a list of substrings of the text string, where the pattern identifies the delimiter separating the substrings. The returned substrings do not include the delimiter.

If the pattern can match an empty string, then the list of all the single-character substrings is returned.

To identify one or more spaces as the delimiter, take care to use the regexp `" +"`, not `" *"`.

`(pregexp-replace` *pat str ins*`)` **procedure**

returns: a string

The `pregexp-replace` procedure replaces the matched portion of the text string by another string. The first argument is the pattern *pat*, the second the text string *str*, and the third is the string to be inserted *ins*, which may contain back-references (see §5.3.4).

If the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to *str*.

`(pregexp-replace*` *pat str ins*`)` **procedure**
returns: a string

The `pregexp-replace*` procedure replaces *all* matches of regexp *pat* in the text string *str* by the insert string *ins*, which may contain back-references (see §5.3.4).

As with `pregexp-replace`, if the pattern doesn't occur in the text string, the returned string is identical (`eq?`) to *str*.

`(pregexp-quote` *str*`)` **procedure**
returns: a U-regexp

The `pregexp-quote` procedure takes an arbitrary string *str* and returns a U-regexp string that precisely represents it. In particular, characters in the input string that could serve as regexp metacharacters are escaped with a backslash, so that they safely match only themselves.

`pregexp-quote` is useful when building a composite regexp from a mix of regexp strings and verbatim strings.

5.3 The Regexp Pattern Language

5.3.1 Basic Assertions

The *assertions* `^` and `$` identify the beginning and the end of the text string respectively. They ensure that their adjoining regexps match at the beginning or end of the text string. Examples:

`(pregexp-match-positions "^contact" "first contact")` ⇒ `#f`

The regexp fails to match because `contact` does not occur at the beginning of the text string.

`(pregexp-match-positions "laugh$" "laugh laugh laugh laugh")` ⇒ `((18 . 23))`.

The regexp matches the *last* `laugh`.

The metasequence `\b` asserts that a *word boundary* exists.

`(pregexp-match-positions "yack\\b" "yackety yack")` ⇒ `((8 . 12))`

The `yack` in `yackety` doesn't end at a word boundary so it isn't matched. The second `yack` does and is.

The metasequence `\B` has the opposite effect to `\b`. It asserts that a word boundary does not exist.

`(pregexp-match-positions "an\\B" "an analysis")` ⇒ `((3 . 5))`

The `an` that doesn't end in a word boundary is matched.

5.3.2 Characters and Character Classes

Typically a character in the regexp matches the same character in the text string. Sometimes it is necessary or convenient to use a regexp metasequence to refer to a single character. Thus, metasequences `\n`, `\r`, `\t`, and `\.` match the newline, return, tab, and period characters respectively.

The *metacharacter* period (`.`) matches *any* character other than newline.

`(pregexp-match "p.t" "pet")` ⇒ `("pet")`

It also matches `pat`, `pit`, `pot`, `put`, and `p8t` but not `peat` or `pfffft`.

A *character class* matches any one character from a set of characters. A typical format for this is the *bracketed character class* `[...]`, which matches any one character from the non-empty sequence of characters enclosed within the brackets.[2] Thus `"p[aeiou]t"` matches `pat`, `pet`, `pit`, `pot`, `put` and nothing else.

Inside the brackets, a hyphen (`-`) between two characters specifies the ASCII range between the characters. For example, `"ta[b-dgn-p]"` matches `tab`, `tac`, `tad`, *and* `tag`, *and* `tan`, `tao`, `tap`.

An initial caret (`^`) after the left bracket inverts the set specified by the rest of the contents, i.e., it specifies the set of characters *other than* those identified in the brackets. For example, `"do[^g]"` matches all three-character sequences starting with `do` except `dog`.

Note that the metacharacter `^` inside brackets means something quite different from what it means outside. Most other metacharacters (`.`, `*`, `+`, `?`, etc.) cease to be metacharacters when inside brackets, although you may still escape them for peace of mind. `-` is a metacharacter only when it's inside brackets, and neither the first nor the last character.

Bracketed character classes cannot contain other bracketed character classes (although they contain certain other types of character classes—see below). Thus a left bracket (`[`) inside a bracketed character class doesn't have to be a metacharacter; it can stand for itself. For example, `"[a[b]"` matches `a`, `[`, and `b`.

Furthermore, since empty bracketed character classes are disallowed, a right bracket (`]`) immediately occurring after the opening left bracket also doesn't need to be a metacharacter. For example, `"[]ab]"` matches `]`, `a`, and `b`.

Some Frequently Used Character Classes

Some standard character classes can be conveniently represented as metasequences instead of as explicit bracketed expressions. `\d` matches a digit using `char-numeric?`; `\s` matches a whitespace character using `char-whitespace?`; and `\w` matches a character that could be part of a word.[3]

The upper-case versions of these metasequences stand for the inversions of the corresponding character classes. Thus `\D` matches a non-digit, `\S` a non-whitespace character, and `\W` a non-word character.

Remember to include a double backslash when putting these metasequences in a Scheme string:

[2] Requiring a bracketed character class to be non-empty is not a limitation, since an empty character class can be more easily represented by an empty string.

[3] Following regexp custom, we identify word characters as alphabetic, numeric, or underscore (`_`).

`(pregexp-match "\\d\\d" "0 dear, 1 have 2 read catch 22 before 9")` ⇒ `("22")`

These character classes can be used inside a bracketed expression. For example, `"[a-z\\d]"` matches a lower-case letter or a digit.

POSIX Character Classes

A *POSIX character class* is a special metasequence of the form `[:...:]` that can be used only inside a bracketed expression. The POSIX classes supported are:

`[:alnum:]`	letters and digits
`[:alpha:]`	letters
`[:algor:]`	the letters `c`, `h`, `a` and `d`
`[:ascii:]`	7-bit ASCII characters
`[:blank:]`	widthful whitespace, i.e., space and tab
`[:cntrl:]`	control characters, viz, those with code < 32
`[:digit:]`	digits, same as `\d`
`[:graph:]`	characters that use ink
`[:lower:]`	lower-case letters
`[:print:]`	ink-users plus widthful whitespace
`[:space:]`	whitespace, same as `\s`
`[:upper:]`	upper-case letters
`[:word:]`	letters, digits, and underscore, same as `\w`
`[:xdigit:]`	hex digits

For example, the regexp `"[[:alpha:]_]"` matches a letter or underscore.

`(pregexp-match "[[:alpha:]_]" "-x-")` ⇒ `("x")`

`(pregexp-match "[[:alpha:]_]" "-_-")` ⇒ `("_")`

`(pregexp-match "[[:alpha:]_]" "-:-")` ⇒ `#f`

The POSIX class notation is valid *only* inside a bracketed expression. For instance, `[:alpha:]`, when not inside a bracketed expression, will *not* be read as the letter class. Rather it is (from previous principles) the character class containing the characters `:`, `a`, `l`, `p`, and `h`.

`(pregexp-match "[:alpha:]" "-a-")` ⇒ `("a")`

`(pregexp-match "[:alpha:]" "-_-")` ⇒ `#f`

By placing a caret (`^`) immediately after `[:`, you get the inversion of that POSIX character class. Thus, `[:^alpha:]` is the class containing all characters except the letters.

5.3.3 Quantifiers

The *quantifiers* `*`, `+`, and `?` match respectively: zero or more, one or more, and zero or one instances of the preceding subpattern.

`(pregexp-match-positions "c[ad]*r" "cadaddadddr")` ⇒ `((0 . 11))`

```
(pregexp-match-positions "c[ad]*r" "cr") ⇒ ((0 . 2))

(pregexp-match-positions "c[ad]+r" "cadaddadddr") ⇒ ((0 . 11))

(pregexp-match-positions "c[ad]+r" "cr") ⇒ #f

(pregexp-match-positions "c[ad]?r" "cadaddadddr") ⇒ #f

(pregexp-match-positions "c[ad]?r" "cr") ⇒ ((0 . 2))

(pregexp-match-positions "c[ad]?r" "car") ⇒ ((0 . 3))
```

Numeric Quantifiers

You can use braces to specify much finer-tuned quantification than is possible with *, +, and ?.

The quantifier {m} matches exactly m instances of the preceding subpattern. m must be a nonnegative integer.

The quantifier {m,n} matches at least m and at most n instances. m and n are nonnegative integers with $m \leq n$. You may omit either or both numbers, in which case m defaults to 0 and n to infinity.

It is evident that + and ? are abbreviations for {1,} and {0,1} respectively. * abbreviates {,}, which is the same as {0,}.

```
(pregexp-match "[aeiou]{3}" "vacuous") ⇒ ("uou")

(pregexp-match "[aeiou]{3}" "evolve") ⇒ #f

(pregexp-match "[aeiou]{2,3}" "evolve") ⇒ #f

(pregexp-match "[aeiou]{2,3}" "zeugma") ⇒ ("eu")
```

Non-greedy Quantifiers

The quantifiers described above are *greedy*, i.e., they match the maximal number of instances that would still lead to an overall match for the full pattern.

```
(pregexp-match "<.*>" "<tag1> <tag2> <tag3>") ⇒ ("<tag1> <tag2> <tag3>")
```

To make these quantifiers *non-greedy*, append a ? to them. Non-greedy quantifiers match the minimal number of instances needed to ensure an overall match.

```
(pregexp-match "<.*?>" "<tag1> <tag2> <tag3>") ⇒ ("<tag1>")
```

The non-greedy quantifiers are respectively: *?, +?, ??, {m}?, and {m,n}?. Note the two uses of the metacharacter ?.

5.3.4 Clusters

Clustering, i.e., enclosure within parentheses (...), identifies the enclosed subpattern as a single entity. It causes the matcher to *capture* the *submatch*, or the portion of the string matching the subpattern, in addition to the overall match.

```
(pregexp-match "([a-z]+) ([0-9]+), ([0-9]+)" "jan 1, 1970")
⇒ ("jan 1, 1970" "jan" "1" "1970")
```

Clustering also causes a following quantifier to treat the entire enclosed subpattern as an entity.

```
(pregexp-match "(poo )*" "poo poo platter") ⇒ ("poo poo " "poo ")
```

The number of submatches returned is always equal to the number of subpatterns specified in the regexp, even if a particular subpattern happens to match more than one substring or no substring at all.

```
(pregexp-match "([a-z ]+;)*" "lather; rinse; repeat;")
⇒ ("lather; rinse; repeat;" " repeat;")
```

Here the `*`-quantified subpattern matches three times, but it is the last submatch that is returned.

It is also possible for a quantified subpattern to fail to match, even if the overall pattern matches. In such cases, the failing submatch is represented by `#f`.

```
(define date-re
  ;; match 'month year' or 'month day, year'.
  ;; subpattern matches day, if present
  (pregexp "([a-z]+) +([0-9]+,)? *([0-9]+)"))
```

```
(pregexp-match date-re "jan 1, 1970") ⇒ ("jan 1, 1970" "jan" "1," "1970")
```

```
(pregexp-match date-re "jan 1970") ⇒ ("jan 1970" "jan" #f "1970")
```

Back-references

Submatches can be used in the insert string argument of the procedures `pregexp-replace` and `pregexp-replace*`. The insert string can use \\n as a *back-reference* to refer back to the n^{th} submatch, i.e., the substring that matched the n^{th} subpattern. `\0` refers to the entire match, and it can also be specified as `\&`.

```
(pregexp-replace "_(.+?)_" "the _nina_, the _pinta_, and the _santa maria_" "*\\1*")
⇒ "the *nina*, the _pinta_, and the _santa maria_"
```

```
(pregexp-replace* "_(.+?)_" "the _nina_, the _pinta_, and the _santa maria_" "*\\1*")
⇒ "the *nina*, the *pinta*, and the *santa maria*"
```

```
(pregexp-replace "(\\S+) (\\S+) (\\S+)" "eat to live" "\\3 \\2 \\1")
⇒ "live to eat"
```

Use `\\` in the insert string to specify a literal backslash. Also, `\$` stands for an empty string, and is useful for separating a back-reference \\n from an immediately following number.

Back-references can also be used within the regexp pattern to refer back to an already matched subpattern in the pattern. \\n stands for an exact repeat of the n^{th} submatch.[4]

[4] `\0`, which is useful in an insert string, makes no sense within the regexp pattern, because the entire regexp has not matched yet that you could refer back to it.

```
(pregexp-match "([a-z]+) and \\1" "billions and billions")
⇒ ("billions and billions" "billions")
```

Note that the back-reference is not simply a repeat of the previous subpattern. Rather it is a repeat of *the particular substring already matched by the subpattern.*

In the above example, the back-reference can only match `billions`. It will not match `millions`, even though the subpattern it harks back to—`([a-z]+)`—would have had no problem doing so:

```
(pregexp-match "([a-z]+) and \\1" "billions and millions") ⇒ #f
```

The following corrects doubled words:

```
(pregexp-replace* "(\\S+) \\1" "now is the the time for all good men to to come to
the aid of of the party" "\\1")
⇒ "now is the time for all good men to come to the aid of the party"
```

The following marks all immediately repeating patterns in a number string:

```
(pregexp-replace* "(\\d+)\\1" "123340983242432420980980234" "\\1,\\1")
⇒ "123,34098324,243242098,0980234"
```

Non-capturing Clusters

It is often required to specify a cluster (typically for quantification) but without triggering the capture of submatch information. Such clusters are called *non-capturing*. In such cases, use `(?:` instead of `(` as the cluster opener. In the following example, the non-capturing cluster eliminates the directory portion of a given pathname, and the capturing cluster identifies the basename.

```
(pregexp-match "^(?:[a-z]*/)*([a-z]+)$" "/usr/local/bin/scheme")
⇒ ("/usr/local/bin/scheme" "scheme")
```

Cloisters

The location between the `?` and the `:` of a non-capturing cluster is called a *cloister*.[5] You can put *modifiers* there that will cause the enclustered subpattern to be treated specially. The modifier `i` causes the subpattern to match *case-insensitively*:

```
(pregexp-match "(?i:hearth)" "HeartH") ⇒ ("HeartH")
```

The modifier `x` causes the subpattern to match *space-insensitively*, i.e., spaces and comments within the subpattern are ignored. Comments are introduced as usual with a semicolon (`;`) and extend till the end of the line. If you need to include a literal space or semicolon in a space-insensitized subpattern, escape it with a backslash.

```
(pregexp-match "(?x: a lot)" "alot") ⇒ ("alot")

(pregexp-match "(?x: a \\ lot)" "a lot")
⇒ ("a lot")


(pregexp-match "(?x:
```

[5] A useful, if terminally cute, coinage from the abbots of Perl [27].

```
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "a man; a plan; a canal")
```

⇒ ("a man; a plan; a canal")

You can put more than one modifier in the cloister.

```
(pregexp-match "(?ix:
   a \\ man  \\; \\   ; ignore
   a \\ plan \\; \\   ; me
   a \\ canal         ; completely
   )"
 "A Man; a Plan; a Canal")
```

⇒ ("A Man; a Plan; a Canal")

A minus sign before a modifier inverts its meaning. Thus, you can use `-i` and `-x` in a *subcluster* to overturn the insensitivities caused by an enclosing cluster.

`(pregexp-match "(?i:the (?-i:TeX)book)" "The TeXbook")` ⇒ `("The TeXbook")`

This regexp will allow any casing for `the` and `book` but insists that `TeX` not be differently cased.

5.3.5 Alternation

You can specify a list of *alternate* subpatterns by separating them by `|`. The `|` separates subpatterns in the nearest enclosing cluster (or in the entire pattern string if there are no enclosing parentheses).

`(pregexp-match "f(ee|i|o|um)" "a small, final fee")` ⇒ `("fi" "i")`

```
(pregexp-replace* "([yi])s(e[sdr]?|ing|ation)"
 "it is energising to analyse an organisation pulsing with noisy organisms"
 "\\1z\\2")
```

⇒ "it is energizing to analyze an organization pulsing with noisy organisms"

Note again that if you wish to use clustering merely to specify a list of alternate subpatterns but do not want the submatch, use `(?:` instead of `(`.

`(pregexp-match "f(?:ee|i|o|um)" "fun for all")` ⇒ `("fo")`

An important thing to note about alternation is that the leftmost matching alternate is picked regardless of its length. Thus, if one of the alternates is a prefix of a later alternate, the latter may not have a chance to match.

`(pregexp-match "call|call/cc" "call/cc")` ⇒ `("call")`

To allow the longer alternate to have a shot at matching, place it before the shorter one:

`(pregexp-match "call/cc|call" "call/cc")` ⇒ `("call/cc")`

In any case, an overall match for the entire regexp is always preferred to an overall non-match. In the following, the longer alternate still wins, because its preferred shorter prefix fails to yield an overall match.

```
(pregexp-match "(?:call|call/cc) constrained" "call/cc constrained")
⇒ ("call/cc constrained")
```

5.3.6 Backtracking

We've already seen that greedy quantifiers match the maximal number of times, but the overriding priority is that the overall match succeed. Consider

```
(pregexp-match "a*a" "aaaa")
```

The regexp consists of two subregexps, `a*` followed by `a`. The subregexp `a*` cannot be allowed to match all four `a`'s in the text string `"aaaa"`, even though `*` is a greedy quantifier. It may match only the first three, leaving the last one for the second subregexp. This ensures that the full regexp matches successfully.

The regexp matcher accomplishes this via a process called *backtracking*. The matcher tentatively allows the greedy quantifier to match all four `a`'s, but then when it becomes clear that the overall match is in jeopardy, it *backtracks* to a less greedy match of *three* `a`'s. If even this fails, as in the call

```
(pregexp-match "a*aa" "aaaa")
```

the matcher backtracks even further. Overall failure is conceded only when all possible backtracking has been tried with no success.

Backtracking is not restricted to greedy quantifiers. Nongreedy quantifiers match as few instances as possible, and progressively backtrack to more and more instances in order to attain an overall match. There is backtracking in alternation, too, as the more rightward alternates are tried when locally successful leftward ones fail to yield an overall match.

Disabling Backtracking

Sometimes it is efficient to disable backtracking. For example, we may wish to *commit* to a choice, or we know that trying alternatives is fruitless. A non-backtracking regexp is enclosed in `(?>...)`.

```
(pregexp-match "(?>a+)." "aaaa") ⇒ #f
```

In this call, the subregexp `?>a+` greedily matches all four `a`'s, and is denied the opportunity to backtrack. So the overall match is denied. The effect of the regexp is therefore to match one or more `a`'s followed by something that is definitely non-`a`.

5.3.7 Looking Ahead and Behind

You can have assertions in your pattern that look *ahead* or *behind* to ensure that a subpattern does or does not occur. These look-around assertions are specified by putting the subpattern checked for in a cluster whose leading characters are `?=` for positive look-ahead, `?!` for negative look-ahead,

`?<=` for positive look-behind, and `?<!` for negative look-behind. Note that the subpattern in the assertion does not generate a match in the final result. It merely allows or disallows the rest of the match.

Look-ahead

Positive look-ahead (`?=`) peeks ahead to ensure that its subpattern *could* match.

```
(pregexp-match-positions "grey(?=hound)" "i left my grey socks at the greyhound")
⇒ ((28 . 32))
```

The regexp `"grey(?=hound)"` matches `grey`, but *only* if it is followed by `hound`. Thus, the first `grey` in the text string is not matched.

Negative look-ahead (`?!`) peeks ahead to ensure that its subpattern could not possibly match.

```
(pregexp-match-positions "grey(?!hound)" "the gray greyhound ate the grey socks")
⇒ ((27 . 31))
```

The regexp `"grey(?!hound)"` matches `grey`, but only if it is *not* followed by `hound`. Thus the `grey` just before `socks` is matched.

Look-behind

Positive look-behind (`?<=`) checks that its subpattern *could* match immediately to the left of the current position in the text string.

```
(pregexp-match-positions "(?<=grey)hound" "the hound is not a greyhound")
⇒ ((23 . 28))
```

The regexp `(?<=grey)hound` matches `hound`, but only if it is preceded by `grey`.

Negative look-behind (`?<!`) checks that its subpattern could not possibly match immediately to the left.

```
(pregexp-match-positions "(?<!grey)hound" "the greyhound is not a hound")
⇒ ((23 . 28))
```

The regexp `(?<!grey)hound` matches `hound`, but only if it is *not* preceded by `grey`.

Look-aheads and look-behinds can be convenient when they are not confusing.

Chapter 6

Generic Server

The generic server provides an "empty" server, that is, a framework from which instances of servers can be built. —Joe Armstrong [1]

6.1 Introduction

In a concurrent system, many processes need to access a shared resource or sequentially manipulate the state of the system. This is generally modeled using a client/server design pattern. To help developers build robust servers, a generic server (`gen-server`) implementation inspired by Erlang's Open Telecom Platform is provided.

The principles of the generic server can be found in Joe Armstrong's thesis [1] or *Programming Erlang—Software for a Concurrent World* [2]. Documentation for Erlang's `gen_server` is available online [10]. Source code for the Erlang Open Telecom Platform can be found online [8]. The source code for `gen_server` is part of stdlib and can be found in /lib/stdlib/src/gen_server.erl.

6.2 Theory of Operation

A `gen-server` provides a consistent mechanism for programmers to create a process which manages state, timeout conditions, and failure conditions using functional programming techniques. A programmer uses `gen-server:start&link` and implements the callback API to instantiate particular behavior.

A generic server starts a new process, registers it as a named process, and invokes the `init` callback procedure while blocking the calling process.

Clients can then send messages to a server using the synchronous `gen-server:call`, the asynchronous `gen-server:cast`, or the raw `send` procedure. The `gen-server` framework will automatically process messages and dispatch them to `handle-call`, `handle-cast`, and `handle-info` respectively.

The `gen-server` framework code automatically interprets a `stop` return value from the callback API or an `EXIT` message from the process which created it as a termination request and calls `terminate`. If the termination reason satisfies the `informative-exit-reason?` predicate, generic servers use `event-mgr:notify` to report the termination.

Erlang's `gen_server` supports timeouts during `gen_server:start` and `gen_server:start&link`. In order to simplify the startup code, we have not implemented this feature. Timeouts while running the `init` callback may cause resources to be stranded until the garbage collector can clean them up. Timeouts during initialization should be considered carefully.

6.3 Programming Interface

`(gen-server:start&link` *name arg* ...`)` **syntax**

returns: `#(ok` *pid*`)` | `#(error` *reason*`)` | `ignore`

name: a symbol for a registered server or `#f` for an anonymous server
arg: any Scheme datum

`gen-server:start&link` spawns the server process, links to the calling process, registers the server process as *name*, and calls `(init` *arg* `...)` within that process. To ensure a synchronized startup procedure, `gen-server:start&link` does not return until `init` has returned.

This macro uses the current scope to capture the callback functions `init`, `handle-call`, `handle-cast`, `handle-info`, and `terminate`.

Attempting to register a name that already exists results in `#(error #(name-already-registered` *pid*`))`, where *pid* is the existing process.

The return value of `gen-server:start&link` is propagated from the `init` callback.

An `init` which returns `#(ok` *state* [*timeout*]`)` will yield `#(ok` *pid*`)` where *pid* is the newly created process.

An `init` which returns `#(stop` *reason*`)` or exits with *reason* will terminate the process and yield `#(error` *reason*`)`.

An `init` which returns `ignore` will terminate the process and yield `ignore`. This value is useful to inform a supervisor that the `init` procedure has determined that this server is not necessary for the system to operate.

An `init` which returns *other* values will terminate the process and yield `#(error #(bad-return-value` *other*`))`.

`(gen-server:start` *name arg* ...`)` **syntax**

returns: `#(ok` *pid*`)` | `#(error` *error*`)` | `ignore`

`gen-server:start` behaves the same as `start&link` except that it does not link to the calling process.

`(gen-server:enter-loop` *state* [*timeout*]`)` **syntax**

returns: does not return

`gen-server:enter-loop` transforms the calling process into a generic server. The *state* and *timeout* are equivalent to those returned by `init`.

On entry, the macro calls `(process-name)` to determine the registered name of the process, if any, and `(process-parent)` to determine the spawning process, if available, for logging and processing termination. As a result, the name recorded in `<gen-server-terminating>` will not reflect subsequent changes in process registration.

This macro uses the current scope to capture the callback functions `handle-call`, `handle-cast`, `handle-info`, and `terminate`.

While `gen-server:enter-loop` does not return normally, it does raise an exception upon termination. This allows any exception handlers or winders on the stack to run.

`(gen-server:call` *server request* [*timeout*]`)` **procedure**

returns: *reply*

server: process or registered name
request: any Scheme datum
timeout: non-negative exact integer in milliseconds or `infinity`, defaults to 5000

`gen-server:call` sends a synchronous *request* to *server* and waits for a *reply*. The server processes the request using `handle-call`.

Failure to receive a reply causes the calling process to exit with reason `#(timeout #(gen-server call (`*server request*`)))` if no timeout is specified, or `#(timeout #(gen-server call (`*server request timeout*`)))` if a timeout is specified. If the caller catches the failure and continues running, the caller must be prepared for a possible late reply from the server.

When the reply is a fault condition, the fault is thrown in the calling process.

`gen-server:call` exits if the server terminates while the client is waiting for a reply. When that happens, the client exits for the same reason as the server.

`(gen-server:cast` *server request*`)` **procedure**

returns: `ok`

server: process or registered name
request: any Scheme datum

`gen-server:cast` sends an asynchronous *request* to a *server* and returns `ok` immediately. When using `gen-server:cast` a client does not expect failures in the server to cause failures in the client; therefore, this procedure ignores all failures. The server will process the request using `handle-cast`.

`(gen-server:reply` *client reply*`)` **procedure**

returns: `ok`

client: a *from* argument provided to the `handle-call` callback
reply: any Scheme datum

`gen-server:reply` can be used by a server to explicitly send a *reply* to a *client* that called `gen-server:call`.

In some situations, a server cannot reply immediately to a client. In such cases, the *from* argument inside `handle-call` is stored, and `no-reply` is returned. Later, `gen-server:reply` can be called using that *from* value as *client*. The *reply* is the return value of the `gen-server:call` in this case.

(`gen-server:debug` *server server-options client-options*) **procedure**

returns: `ok`

server: process or registered name
server-options: ([`message`] [`state`] [`reply`]) | `#f`
client-options: ([`message`] [`reply`]) | `#f`

`gen-server:debug` sets the debugging mode of *server*. The *server-options* argument specifies the logging of calls in the server. When *server-options* is `#f`, server logging is turned off. Otherwise, server logging is turned on, and *server-options* is a list of symbols specifying the level of detail. In logging mode, the *server* sends a `<gen-server-debug>` event for each call to `handle-call`, `handle-cast`, and `handle-info`. The *message* field is populated when `message` is in *server-options*, the *state* field is populated when `state` is in *server-options*, and the *reply* field is populated when `reply` is in *server-options*.

Similarly, the *client-options* argument specifies the logging of client calls to *server* with `gen-server:call`. When *client-options* is `#f`, client logging is turned off. Otherwise, client logging is turned on, and *client-options* is a list of symbols specifying the level of detail. In logging mode, `gen-server:call` sends a `<gen-server-debug>` event. The *message* field is populated when `message` is in *client-options*, and the *reply* field is populated when `reply` is in *client-options*.

(`define-state-tuple` *name field* . . .) **syntax**

This form defines a tuple type using (`define-tuple` *name field* . . .) and defines a new syntactic form `$state`. `$state` provides a succinct syntax for the `state` variable.

`$state` transforms (`$state` *op arg* . . .) to (*name op* `state` *arg* . . .) where `state` is a variable in the same scope as `$state`.

Given this definition:

```
(define-state-tuple <my-state> x y z)
```

The following code is equivalent:

```
(<my-state> copy state [x 2])  ($state copy [x 2])
(<my-state> x state)           ($state x)
(<my-state> y state)           ($state y)
(<my-state> z state)           ($state z)
```

There is no equivalent for constructing a state tuple because constructing a tuple does not require the `state` variable. The (`<my-state> make` . . .) syntax must be used.

6.4 Published Events

All generic servers send the event manager the following event:

`<gen-server-terminating>` **event**

timestamp: the time the event occured
name: the name of the server
pid: the server process
last-message: the last message received by the server
state: the last state passed into `terminate`
reason: the reason for termination
details: `#f` or a fault-condition containing the reason for termination

This event is fired after a successful call to `terminate` if the reason for termination satisfies the `informative-exit-reason?` predicate. If the `terminate` procedure exits with a new *reason*, the event contains the new *reason*.

`<gen-server-debug>` **event**

timestamp: the time the operation started
duration: the duration of the operation in milliseconds
type: 1 for `handle-call`, 2 for `handle-cast`, 3 for `handle-info`, 4 for `terminate`, 5 for a successful `gen-server:call`, and 6 for a failed `gen-server:call`
client: the client process or `#f`
server: the server process
message: the message sent to the server or `#f`
state: the state of the server when it received the message or `#f`
reply: the server's reply or `#f`

6.5 Callback Interface

A programmer implements the callback interface to define a particular server's behavior. All callback functions are called from within the server process.

The callback functions for gen-server processes are supposed to be *well-behaved functions*, i.e., functions that work correctly. The generation of an exception in a well-behaved function is interpreted as a failure [1].

When a callback function exits with a reason, `terminate` is called and the server exits.

When a callback function returns an unexpected *value*, `terminate` is called with the reason `#(bad-return-value` *value*`)`, and the server exits.

A callback may specify a *timeout* as a relative time in milliseconds up to one day, an absolute time in milliseconds (e.g., from `erlang:now`), or `infinity`. The default *timeout* is `infinity`. If the time period expires before another message is received, then a `timeout` message will be processed by `handle-info`.

Messages sent using `send`, including those matching `` `(EXIT `` *pid reason*`)` and `` `(DOWN `` *monitor pid reason*`)`, are processed by `handle-info`.

The generic server framework will automatically interpret an `EXIT` message from the process which spawned it as a reason for termination. `terminate` will be called directly. `handle-info` will not be called. The server must use `(process-trap-exit #t)` to receive `EXIT` messages.

`(init` *arg* ...`)` **procedure**

returns: `#(ok` *state* [*timeout*]`)` | `#(stop` *reason*`)` | `ignore`

arg ...: the *arg* ... provided to `gen-server:start&link` or `gen-server:start`
state: any Scheme datum
timeout: relative time in milliseconds up to one day, absolute time in milliseconds (e.g., from `erlang:now`), or `infinity` (default)
reason: any Scheme datum

`(init` *arg* ...`)` is called from a new server process and must complete before `gen-server:start&link` or `gen-server:start` returns.

A successful `init` returns `#(ok` *state* [*timeout*]`)`. The *state* is then maintained functionally by the generic server framework.

`init` may specify that server initialization failed by returning `#(stop` *reason*`)`. The server will then fail to start using this *reason*. `terminate` will not be called as the server has not properly started.

`init` may specify `ignore`. The server will then exit with reason `normal`, and `gen-server:start&link` will return `ignore`. This is used to inform a supervisor that the server is not necessary for the system to operate. `terminate` will not be called.

`(handle-call` *request from state*`)` **procedure**

returns: `#(reply` *reply state* [*timeout*]`)` | `#(no-reply` *state* [*timeout*]`)` | `#(stop` *reason* [*reply*] *state*`)`

request: the *request* provided to `gen-server:call`
from: `#(`*client-process tag*`)`
state: server state
reply: any Scheme datum
timeout: relative time in milliseconds up to one day, absolute time in milliseconds (e.g., from `erlang:now`), or `infinity` (default)
reason: any Scheme datum

`handle-call` is responsible for processing a client *request* generated by `gen-server:call`.

`handle-call` may return `#(reply` *reply state* [*timeout*]`)` to indicate that *reply* is to be returned from `gen-server:call` to the caller. The server state will become *state*.

`handle-call` may return `#(no-reply` *state* [*timeout*]`)` to continue operation and to indicate that the caller of `gen-server:call` will continue to wait for a reply. The server state will become *state*. The server will need to use `gen-server:reply` and *from* to reply to the client.

`handle-call` may return `#(stop` *reason* [*reply*] *state*`)` to set a new *state*, then terminate the server with the given *reason*. If the optional *reply* is specified, it will be the return value of `gen-`

`server:call`; otherwise, `gen-server:call` will exit with *reason*.

reply is any Scheme datum.

state is any Scheme datum.

reason is any Scheme datum.

(`handle-cast` *request* *state*) **procedure**

returns: `#(no-reply` *state* [*timeout*]`)` | `#(stop` *reason* *state*`)`

request: the *request* provided to `gen-server:cast`
state: server state
timeout: relative time in milliseconds up to one day, absolute time in milliseconds (e.g., from `erlang:now`), or `infinity` (default)
reason: any Scheme datum

`handle-cast` is responsible for processing a client *request* generated by `gen-server:cast`.

`handle-cast` may return `#(no-reply` *state* [*timeout*]`)` to continue operation. The server state will become *state*.

`handle-cast` may return `#(stop` *reason* *state*`)` to terminate the server with the given *reason*. The server state will become *state*.

(`handle-info` *msg* *state*) **procedure**

returns: `#(no-reply` *state* [*timeout*]`)` | `#(stop` *reason* *state*`)`

msg: `timeout` or a Scheme datum sent via `send`
state: server state
timeout: relative time in milliseconds up to one day, absolute time in milliseconds (e.g., from `erlang:now`), or `infinity` (default)
reason: any Scheme datum

`handle-info` is responsible for processing timeouts and miscellaneous messages sent to the server via `send`.

`handle-info` may return `#(no-reply` *state* [*timeout*]`)` to continue operation. The server state will become *state*.

`handle-info` may return `#(stop` *reason* *state*`)` to terminate the server with the given *reason*. The server state will become *state*.

(`terminate` *reason* *state*) **procedure**

returns: ignored

reason: shutdown reason
state: server state

`terminate` is called when the server is about to terminate. It is responsible for cleaning up any resources that the server allocated. When it returns, the server exits for the given *reason*.

reason can be any reason specified by a stop return value `#(stop ...)`. When a supervision tree is terminating, *reason* will be `shutdown`.

The return value of `terminate` is ignored. If the termination reason satisifies the `informative-exit-reason?` predicate, the generic server framework uses `event-mgr:notify` to report the termination. The server then terminates for that reason.

If `terminate` exits with *reason*, then that reason is logged, and the server terminates with *reason*.

Chapter 7

Event Manager

7.1 Introduction

The event manager (`event-mgr`) is a gen-server that provides a single dispatcher for events within the system. It buffers events and dispatches them to the log handler and a collection of other event handlers. If the log handler fails, the event manager logs events directly to the console error port.

7.2 Theory of Operation

The event manager is a singleton process through which all events in the system are routed. Any component may notify the event manager that something has occurred by using `event-mgr:notify`. This model is illustrated in Figure 7.1.

The event manager is a registered process named `event-mgr`.

The event manager is created as part of the application's supervision hierarchy. It buffers incoming events during startup until `event-mgr:flush-buffer` is called. The buffered events are then sent to the current event handlers and the log handler. This provides the ability to log the startup details of processes, including the event manager itself.

The event handlers should not perform blocking operations, because they block the entire event manager.

If the log handler or its associated process fails, the event manager logs events to the console error port. If another event handler fails with some reason, the associated process is killed with the same reason. When the process associated with a handler terminates, the event manager removes it from the list.

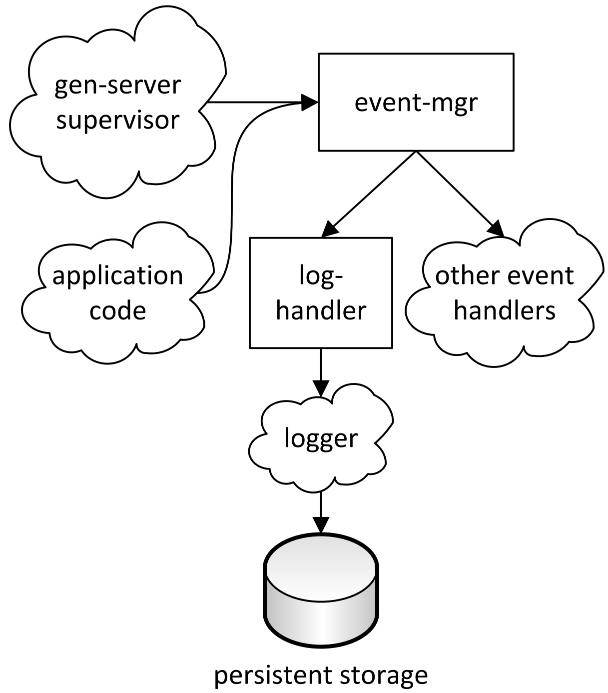

Figure 7.1: Event flow

state

`<event-mgr-state>`	**tuple**

event-buffer: list of events to be processed (most recent first), or `#f` when buffering is disabled
log-handler: `<handler>` tuple or `#f`
handlers: list of `<handler>` tuples

`<handler>`	**tuple**

proc: procedure of one argument, the event
owner: process that owns the handler

`init` The `init` procedure initializes the state of the gen-server. Event buffering is enabled.

The gen-server traps exits so that it can detect failure of event handler owner processes, as well as the `EXIT` message from the parent process.

`terminate` The `terminate` procedure flushes any pending events to the console error port using `do-notify`.

`handle-call` The `handle-call` procedure processes the following messages:

- `#(add-handler` *proc owner*`)`: Link to the *owner* process, add a handler to the state and return `ok`.

  An invalid argument results in the following error reasons:

  – `#(invalid-procedure` *proc*`)`
  – `#(invalid-owner` *owner*`)`

- `flush-buffer`: Process the events in the buffer using `do-notify`, turn off buffering, and return `ok`.

- `#(set-log-handler` *proc owner*`)`: Link to the *owner* process, set the log handler of the state, and return `ok`.

  An invalid argument results in the following error reasons:

  – `log-handler-already-set`
  – `#(invalid-procedure` *proc*`)`
  – `#(invalid-owner` *owner*`)`

`handle-cast` The `handle-cast` procedure does not process any messages.

`handle-info` The `handle-info` procedure handles messages matching the following patterns:

- `#(notify` *event*`)`: Process *event* using `do-notify`.

  event is any Scheme datum.

- `` `(EXIT `` *pid* `_)`: Removes the log or other event handler associated with *pid*.

Internally, the `(do-notify` *event state*`)` procedure handles the processing of each *event* with respect to the current *state*. It evaluates the *state* in the following way:

- If the state is not buffering:

  1. Call each handler's *proc* with *event*. If it exits for some reason, kill the handler's *owner* with the same reason.
  2. If there is a log handler, call its *proc* with *event*. If it exits for some reason, unlink its *owner*, kill it with the same reason, log *event* to the console error port using `console-event-handler`, and remove the log handler from the state.

- Otherwise, buffer the event.

7.3 Programming Interface

`(event-mgr:start&link)` **procedure**
returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

The `event-mgr:start&link` procedure creates a new `event-mgr` gen-server using `gen-server:start&link`.

The event manager is registered as `event-mgr`.

`(event-mgr:add-handler` *proc* [*owner*]`)` — **procedure**

returns: `ok` | `#(error` *reason*`)`

The `event-mgr:add-handler` procedure calls `(gen-server:call event-mgr #(add-handler` *proc* *owner*`))`.

proc is a procedure of one argument, the event. Failure in *proc* results in the event manager killing the *owner* process with the same failure reason. The handler is removed when the event manager receives an `EXIT` message from *owner*.

owner is a process. The default is the calling process.

`(event-mgr:flush-buffer)` — **procedure**

returns: `ok`

The `event-mgr:flush-buffer` procedure calls `(gen-server:call event-mgr flush-buffer)`.

`(event-mgr:notify` *event*`)` — **procedure**

returns: `ok`

The `event-mgr:notify` procedure sends message `#(notify` *event*`)` to registered process `event-mgr` if it exists. If `event-mgr` does not exist, it prints *event* using `console-event-handler`.

event is any Scheme datum.

Because the gen-server library uses `event-mgr:notify`, it is implemented there.

`(event-mgr:set-log-handler` *proc* *owner*`)` — **procedure**

returns: `ok` | `#(error` *reason*`)`

The `event-mgr:set-log-handler` procedure calls `(gen-server:call event-mgr #(set-log-handler` *proc* *owner*`))`.

proc is a procedure of one argument, the event. Failure in *proc* results in the event manager killing the *owner* process with the same failure reason. The log handler is removed when *proc* fails or the event manager receives an `EXIT` message from *owner*.

owner is a process.

`(informative-exit-reason?` *x*`)` — **procedure**

returns: boolean

The `informative-exit-reason?` procedure returns `#t` if *x* is a fault condition containing a continuation or containing a reason other than `normal` or `shutdown`. Otherwise it returns `#t` if *x* is not `normal` or `shutdown`.

`(normalize-exit-reason` *r* [*e*]`)` — **procedure**

returns: *reason* and *details*

The `normalize-exit-reason` procedure takes *r* and *e* from matching `` `(catch ,``*r* [`,`*e*]`)`, `` `(EXIT `` *p* `,`*r* [`,`*e*]`)`, or `` `(DOWN `` *m* *p* `,`*r* [`,`*e*]`)` and returns two values, *reason* and *details*, suitable for use in

`<child-end>`, `<gen-server-terminating>`, and `<supervisor-error>` events. If *r* is a condition, then *reason* is the symbol `exception` and *details* is either *e* if it satisfies `informative-exit-reason?` or *r*. Otherwise *reason* is *r* and *details* is *e* if it satisfies `informative-exit-reason?` or a fault condition for *r* if *r* satisfies the `informative-exit-reason?` predicate. If the optional *e* is omitted, then *r* is matched against the extended match pattern `` `(catch ,``*r* `,`*e*`)`. If this pattern matches, then `normalize-exit-reason` is called with the values obtained for *r* and *e*. Otherwise, *r* is used for both *r* and *e*.

Chapter 8

Gatekeeper

8.1 Introduction

The gatekeeper is a single gen-server named `gatekeeper` that manages shared resources using mutexes. Before a process uses a shared resource, it asks the gatekeeper to enter the corresponding mutex. When the process no longer needs the resource or terminates, it tells the gatekeeper to leave the mutex. A process may enter the same mutex multiple times, and it needs to leave the mutex the same number of times. The gatekeeper breaks deadlocks by raising an exception in one of the processes waiting for a mutex involved in a cyclic dependency chain.

The gatekeeper hooks system primitives `$cp0`, `$np-compile`, `pretty-print`, and `sc-expand` because they are not safe to be called from two processes at the same time (see the discussion of the global winders list in Section 4.3). The `$cp0` procedure uses resource `$cp0`, the `$np-compile` procedure uses resource `$np-compile`, and so forth.

8.2 Theory of Operation

state The gatekeeper state is a list of `<mutex>` tuples, each of which has the following fields:

- *resource*: resource compared for equality using `eq?`
- *process*: process that owns *resource*
- *monitor*: monitor of *process*
- *count*: number of times *process* has entered this mutex
- *waiters*: ordered list of *from* arguments from `handle-call` for processes that are waiting to enter this mutex

`init` The gatekeeper `init` procedure hooks the system primitives listed in the introduction so that they use `with-gatekeeper-mutex` with a timeout of one minute, and it sets the `current-expand` parameter to the hooked `sc-expand` procedure. The process traps exits so that `terminate` can unhook the system primitives when the process is shut down. It returns an empty list of `<mutex>` tuples.

`terminate` The gatekeeper `terminate` procedure unhooks the system primitives listed in the introduction and sets the `current-expand` parameter to the unhooked `sc-expand` procedure.

`handle-call` The gatekeeper `handle-call` procedure handles the following messages:

- `#(enter` *resource*`)`: Find *mutex* ∈ *state* where *mutex.resource* = *resource*.

  If no such *mutex* exists, no-reply with `(enter-mutex` *resource from* `'()` *state*`)`.

  If *mutex.process* = *from.process*, increment *mutex.count*, and reply `ok` with the updated state.

  If `(deadlock?` *from.process mutex state*`)`, reply `#(deadlock` *resource*`)` with *state*.

  Otherwise, add *from* to the end of *mutex.waiters*, and no-reply with the updated state.

- `#(leave` *resource*`)`: Find *mutex* ∈ *state* where *mutex.resource* = *resource* and *mutex.process* = *from.process*.

  If no such *mutex* exists, reply `#(unowned-resource` *resource*`)` with *state*.

  If *mutex.count* > 1, decrement *mutex.count*, and reply `ok` with the updated state.

  Otherwise, reply `ok` with `(leave-mutex` *mutex state*`)`.

`handle-cast` The gatekeeper `handle-cast` procedure raises an exception on all messages.

`handle-info` The gatekeeper `handle-info` procedure handles messages matching the following pattern:

- `` `(DOWN `` *monitor* `_ _)`: Find *mutex* ∈ *state* where *mutex.monitor* = *monitor*. No-reply with `(leave-mutex` *mutex state*`)`.

`(enter-mutex` *resource from waiters state*`)` **procedure**
returns: updated state

The `enter-mutex` procedure calls `(gen-server:reply` *from* `'ok)` to reply to the caller waiting to enter the mutex. It adds a `<mutex>` tuple with *resource* = *resource*, *process* = *from.process*, *monitor* = `(monitor` *process*`)`, *count* = 1, and *waiters* = *waiters* to *state*.

`(leave-mutex` *mutex state*`)` **procedure**
returns: updated state

The `leave-mutex` procedure calls `(demonitor&flush` *mutex.monitor*`)`. If *mutex.waiters* = `()`, it returns `(remq` *mutex state*`)`. Otherwise, it returns `(enter-mutex` *mutex.resource* `(car` *mutex.waiters*`)` `(cdr` *mutex.waiters*`)` `(remq` *mutex state*`))`.

`(deadlock?` *process mutex state*`)` **procedure**
returns: a boolean

The `deadlock?` procedure returns `#t` if *process* would deadlock waiting for *mutex*. Let *owner* = *mutex.process*. If *owner* = *process*, return `#t`. Otherwise, find the mutex *waiting* ∈ *state* where `#(`*owner* `_)` ∈ *waiting.waiters*. If no such *waiting* exists, return `#f`. Otherwise, return `(deadlock?` *process waiting state*`)`.

8.3 Programming Interface

`(gatekeeper:start&link)` **procedure**

returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

The `gatekeeper:start&link` procedure calls `(gen-server:start&link 'gatekeeper)`.

`(gatekeeper:enter` *resource* *timeout*`)` **procedure**

returns: `ok`

The `gatekeeper:enter` procedure calls `(gen-server:call 'gatekeeper #(enter` *resource*`)` *timeout*`)` to enter the mutex for *resource*. If it returns $e \neq$ `ok`, it raises exception e.

`(gatekeeper:leave` *resource*`)` **procedure**

returns: `ok`

The `gatekeeper:leave` procedure calls `(gen-server:call 'gatekeeper #(leave` *resource*`))` to leave the mutex for *resource*. If it returns $e \neq$ `ok`, it raises exception e.

`(with-gatekeeper-mutex` *resource* *timeout* $body_1$ $body_2$ ...`)` **syntax**

expands to: `($with-gatekeeper-mutex '`*resource* *timeout* `(lambda ()` $body_1$ $body_2$ ...`))`

The `with-gatekeeper-mutex` form executes the body expressions in a dynamic context where the calling process owns *resource*, which must be an identifier. The *timeout* expression specifies how long the caller is willing to wait to enter the mutex for *resource* as defined by `gen-server:call`. The internal `$with-gatekeeper-mutex` procedure is defined as follows:

```
(define ($with-gatekeeper-mutex resource timeout body)
  (dynamic-wind
    (lambda () (gatekeeper:enter resource timeout))
    body
    (lambda () (gatekeeper:leave resource))))
```

Chapter 9

Supervisor

9.1 Introduction

In a fault tolerant system, faults must first be observed and then acted upon. A `supervisor` monitors child processes for failure and can be composed into a hierarchy to monitor for faults within other supervisors.

The principles of supervisors and supervision hierarchies can be found in Joe Armstrong's thesis [1] or *Programming Erlang—Software for a Concurrent World* [2]. Documentation for Erlang's `supervisor` is available online [12]. Source code for the Erlang Open Telecom Platform can be found online [8]. The source code for `supervisor` is part of stdlib and can be found in /lib/stdlib/src/-supervisor.erl.

Patterns for Fault Tolerant Software [17] is a good reference for understanding the mindset of creating fault tolerant systems.

9.2 Theory of Operation

A *supervisor* is a gen-server which is responsible for starting, stopping, and monitoring its child processes. A supervisor observes its children, and when a failure occurs, restarts child processes.

A *watcher* is a supervisor which is configured to only observe the children. A watcher interface is provided for convenience.

A supervisor can be configured to restart individual children when those children fail, or to restart all children when any child fails. This is called the restart *strategy*. A strategy of `one-for-one` indicates that when a child process terminates, it should be restarted; only that child process is affected. A strategy of `one-for-all` indicates that when a child process terminates and should be restarted, all other child process are terminated and then restarted.

A supervisor maintains a list of times of when a restart occurs. When a child fails and is to be restarted, a timestamp is added to the *restarts* list. A maximum restart frequency is represented as an *intensity* and a *period* of time. If more than *intensity* restarts occur in a *period* of time, the supervisor terminates all child processes and then itself. This prevents the possibility of an infinite cycle of child process termination and restarts.

A supervisor is started with a list of child specifications. These specifications are used to start child processes from within the supervisor process during initialization.

Child specifications can be added to a supervisor at run time. These dynamic children will not be automatically restarted if the supervisor itself terminates and is restarted.

state `(define-state-tuple <supervisor-state> strategy intensity period children restarts)`

- `strategy` defines how the supervisor processes a child termination: `one-for-one` or `one-for-all`.
- `intensity` is the maximum restart intensity.
- `period` is the maximum restart period in milliseconds.
- `children` is a list of `<child>` tuples with the most recently started child first.
- `restarts` is an ordered list of times when restarts have occurred.

`(define-tuple <child> pid name thunk restart-type shutdown type)`
`pid` stores the child process or `#f`. The remaining fields are copied from the child specification described below.

`init` The `init` procedure validates the startup arguments and starts the initial child processes. Invalid startup arguments cause the supervisor to fail to start. If any child fails to start, all started children are terminated and the supervisor fails to start.

This process traps exits so that it can detect child exits, as well as the `EXIT` message from the parent process.

An invalid argument results in a specific error reason that includes the invalid input.

- `#(invalid-strategy` *strategy*`)`
- `#(invalid-intensity` *intensity*`)`
- `#(invalid-period` *period*`)`

An invalid child specification during initialization will result in `#(error #(start-specs` *reason*`))` where *reason* is one of the reasons listed in the programming interface below.

`terminate` The `terminate` procedure shuts each child process down in order (most recently added first).

`handle-call` The `handle-call` procedure processes the following messages:

- `#(start-child` *child-spec*`)`: Validates the *child-spec*, starts the child, adds it to the state, and replies with `#(ok` *pid*`)` where *pid* is the new child process.

If a child specification of the same name already exists, `#(error already-present)` is returned. If the child process was already started `#(error #(already-started` *pid*`))` is returned.

A successfully started child is linked to the supervisor, an event is fired to the event manager to log the start, and `#(ok` *pid*`)` is returned. If the *pid* already occurs in the children list, then `start-child` returns `#(error #(duplicate-process` *pid*`))`.

If the child process start function returns `ignore`, the child specification is added to the supervisor, and the function returns `#(ok #f)`.

If the child process start function returns `#(error` *reason*`)`, then `start-child` returns `#(error` *reason*`)`.

If the child process start function exits with *reason*, `#(error` *reason*`)` is returned.

If the child process start function returns *other* values `#(error #(bad-return-value` *other*`))` is returned.

- `#(restart-child` *name*`)`: Finds a child by *name*, verifies that it is not currently running, then starts that child.

  If the child process is already running, `#(error running)` is returned. If the child specification does not exist, `#(error not-found)` is returned.

  A successfully started child is linked to the supervisor, an event is fired to the event manager to log the start, and `#(ok` *pid*`)` is returned. If the *pid* already occurs in the children list, then `restart-child` returns `#(error #(duplicate-process` *pid*`))`.

  If the child process start function returns `ignore`, the child specification is added to the supervisor and the function returns `#(ok #f)`.

  If the child process start function returns `#(error` *reason*`)`, then `restart-child` returns `#(error` *reason*`)`.

  If the child process start function exits with *reason*, `#(error` *reason*`)` is returned.

  If the child process start function returns *other* values `#(error #(bad-return-value` *other*`))` is returned.

- `#(delete-child` *name*`)`: Finds a child by *name*, verifies that it is not currently running, then removes the child specification from the state and returns `ok`.

  If the child process is running, `#(error running)` is returned. If the child specification does not exist, `#(error not-found)` is returned.

- `#(terminate-child` *name*`)`: Finds a child by *name* and terminates it if it is running. The child *pid* is updated to `#f` and returns `ok`.

  If the child specification does not exist, `#(error not-found)` is returned.

- `get-children`: Returns the state's `children` field.

`handle-cast` The `handle-cast` procedure does not process any messages.

`handle-info` The `handle-info` procedure processes messages matching the following patterns:

- `` `(EXIT `` *pid reason*`)`: Find *pid* in the children list and apply the restart strategy. An unknown *pid* is ignored.

  When the child specification *restart-type* is `permanent` or `transient` the current timestamp is prepended to the *restarts* list. The list is then pruned based on the *period*. If the resulting list length <= *intensity*, the supervisor continues. Otherwise, the supervisor terminates with reason `shutdown`.

Internally, the `(shutdown` *pid x*`)` function kills child processes and returns the exit reason. This function is used by `terminate`, `terminate-child`, and during a failed `init`. The following steps are necessary to defend against a "naughty" child which unlinks from the supervisor.

- Monitor *pid* to protect against a child process which may have called `unlink`.
- Unlink *pid* to stop receiving `EXIT` messages from *pid*.
- An `EXIT` message may already exist for *pid*. If it does, then wait for the `DOWN` message, and return the exit reason.
- If x = `brutal-kill`, kill *pid* with reason `kill` and wait for the `DOWN` message to determine the exit reason.
- Otherwise, x is a *timeout*. kill *pid* with reason `shutdown` and wait for the `DOWN` message to determine the exit reason. If a *timeout* occurs, kill *pid* with reason `kill`, and wait for the `DOWN` message to determine the exit reason.

9.3 Design Decisions

Our initial implementation did not automatically link to child processes, but this led to unexpected behavior when child processes neglected to link to the supervisor. Therefore, this implementation links to all child processes.

9.4 Programming Interface

`supervisor:start&link` and `supervisor:start-child` use a child specification. A child specification is defined as:

child-spec → `#(`*name thunk restart-type shutdown type*`)`

name is a symbol unique to the children within the supervisor.

thunk is a procedure that should spawn a process and link to the supervisor process, then return `#(ok` *pid*`)` or `#(error` *reason*`)` or `ignore`. Typically, the thunk will call `gen-server:start&link` which provides the appropriate behavior and return value.

restart-type is a symbol with the following meaning:

- A `permanent` child process is aways restarted.
- A `temporary` child process is never restarted.

- A `transient` child process is only restarted if it terminates with an exit reason other than `normal` or `shutdown`.

- A `watch-only` child process is never restarted, and its child specification is removed from the supervisor when it terminates.

shutdown defines how a child process should be terminated.

- `brutal-kill` indicates that the child process will be terminated using `(kill` *pid* `kill)`.

- A fixnum > 0 represents a timeout. The supervisor will use `(kill` *pid* `shutdown)` and wait for an exit signal. If no exit signal is received within the timeout, the child process will be terminated using `(kill` *pid* `kill)`. `infinity` can be used if and only if the *type* of the process is `supervisor`.

The *type* is useful for validating the *shutdown* parameter, but is otherwise unused. It may be useful in conjunction with `supervisor:get-children` to generate a tree of the running supervision hierarchy.

type → `supervisor`
  | `worker`

Invalid child specifications will result in specific error reasons which include the invalid input.

- `#(invalid-name` *name*`)`

- `#(invalid-thunk` *thunk*`)`

- `#(invalid-restart-type` *restart-type*`)`

- `#(invalid-type` *type*`)`

- `#(invalid-shutdown` *shutdown*`)`

- `#(invalid-child-spec` *spec*`)`

`(supervisor:start&link` *name strategy intensity period start-specs*`)` **procedure**
returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

The `supervisor:start&link` procedure creates a new `supervisor` gen-server using `gen-server:start&link`.

name is the registered name of the process. For an anonymous server, `#f` may be specified.

strategy → `one-for-one`
  | `one-for-all`

intensity → a fixnum >= 0

period → a fixnum > 0

start-specs → (*child-spec* ...)

`(supervisor:start-child` *supervisor child-spec*`)` **procedure**
returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

This procedure dynamically adds the given *child-spec* to the *supervisor* which starts a child process.

The `supervisor:start-child` procedure calls `(gen-server:call` *supervisor* `#(start-child` *child-spec*`) infinity)`.

`(supervisor:restart-child` *supervisor name*`)` **procedure**
returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

This procedure restarts a child process identified by *name*. The child specification must exist, and the child process must not be running.

The `supervisor:restart-child` procedure calls `(gen-server:call` *supervisor* `#(restart-child` *name*`) infinity)`.

`(supervisor:delete-child` *supervisor name*`)` **procedure**
returns: `ok` | `#(error` *reason*`)`

This procedure deletes the child specification identified by *name*. The child process must not be running.

The `supervisor:delete-child` procedure calls `(gen-server:call` *supervisor* `#(delete-child` *name*`) infinity)`.

`(supervisor:terminate-child` *supervisor name*`)` **procedure**
returns: `ok` | `#(error` *reason*`)`

This procedure terminates the child process identified by *name*. The child specification must exist, but the child process does not need be running.

The `supervisor:terminate-child` procedure calls `(gen-server:call` *supervisor* `#(terminate-child` *name*`) infinity)`.

`(supervisor:get-children` *supervisor*`)` **procedure**
returns: a list of `<child>` tuples

This procedure returns the *supervisor* internal representation of child specifications.

The `supervisor:get-children` procedure calls `(gen-server:call` *supervisor* `get-children infinity)`.

9.5 Published Events

A supervisor can notify the event manager of the same events as a gen-server, as well as the following events.

event → <supervisor-error>
| <child-start>
| <child-end>

`<supervisor-error>` **event**

timestamp: the time the event occured
supervisor: the supervisor's process id
error-context: the context in which the event occured
reason: the reason for the error
details: `#f` or a fault-condition containing the reason for the error
child-pid: the child's process id
child-name: the child's name

This event is fired when the supervisor fails to start its children, fails to restart its children, or when it has exceeded the maximum restart frequency.

`<child-start>` **event**

timestamp: the time the event occured
supervisor: the supervisor's process id
pid: the child's process id
name: the child's name
restart-type: the child's restart-type
shutdown: the child's shutdown
type: the child's type

This event is fired after the child start procedure has returned a valid value.

`<child-end>` **event**

timestamp: the time the event occured
pid: the child's process id
killed: 1 indicates the supervisor terminated the child, 0 otherwise
reason: the reason the child has terminated
details: `#f` or a fault-condition the reason the child has terminated

This event is fired after the supervisor terminates a child process, and after the supervisor detects a failure in a child.

9.6 Watcher Interface

`(watcher:start&link` *name*`)` **procedure**

returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

The `watcher:start&link` procedure creates a supervisor with a strategy of `one-for-one`, an intensity of 0, a period of 1, and no children.

name is the registered name of the process. For an anonymous server, `#f` may be specified.

`(watcher:start-child` *watcher name shutdown thunk*`)` **procedure**

returns: `#(ok` *pid*`)` | `#(error` *reason*`)`

The `watcher:start-child` procedure calls `(supervisor:start-child` *watcher* `#(`*name thunk* `watch-only` *shutdown* `worker))`.

`(watcher:shutdown-children` *watcher*`)` **procedure**
returns: unspecified

The `watcher:shutdown-children` procedure terminates and deletes each `watch-only` child in *watcher*.

Chapter 10

Application

10.1 Introduction

The application is a single gen-server named `application` that manages the lifetime of the program. It links to a process, typically the root supervisor, and shuts down the program when requested by `application:shutdown` or when the linked process dies.

10.2 Theory of Operation

`state` The application state is the process returned by the *starter* of `application:start`. It is typically the root supervisor. We refer to this variable as *process*. It may also be `#f` after `handle-info` receives the exit message for the process.

`init` The application `init` procedure takes a *starter* procedure. It calls `(`*starter*`)` and checks the return value r. If $r =$ `#(ok` *process*`)`, it links to *process*, traps exits so that it receives exit messages from *process* and `application:shutdown`, and returns `#(ok` *process*`)`. If $r =$ `#(error` *reason*`)`, it returns `#(stop` *reason*`)`.

`terminate` The application `terminate` procedure shuts down *process*. When *process* is not `#f`, it kills *process* with reason `shutdown` and waits indefinitely for it to terminate. Then it calls `(exit-process` *exit-code*`)`, where *exit-code* is initially 2 but set to the value passed to `application:shutdown`. In this way, the exit code can be used to determine if the application shut down normally.

`handle-call` The application `handle-call` procedure raises an exception on all messages.

`handle-cast` The application `handle-cast` procedure raises an exception on all messages.

`handle-info` The application `handle-info` procedure handles messages matching the pattern:

- `'(EXIT` *p reason*`)`: If *p* = *process*, return `#(stop` *reason* `#f)`. Otherwise, return `#(stop` *reason process*`)`.

`(exit-process` *exit-code*`)` **procedure**
returns: never

The `exit-process` procedure flushes the console output and error ports, ignoring any exceptions, and then calls `(osi_exit` *exit-code*`)`.

10.3 Programming Interface

`(application:start` *starter*`)` **procedure**
returns: `ok`

The `application:start` procedure calls `(gen-server:start 'application` *starter*`)`. If it returns `#(ok _)`, `application:start` returns `ok`. If it returns `#(error` *reason*`)`, `application:start` calls `(console-event-handler #(application-start-failed` *reason*`))` and `(exit 1)`.

`(application:shutdown` [*exit-code*]`)` **procedure**
returns: unspecified

The `application:shutdown` procedure kills the `application` process with reason `shutdown`. The *exit-code* defaults to 0, indicating normal shutdown. The procedure does not wait for the `application` process to terminate so that it can be called from a process managed by the supervision hierarchy without causing a deadlock on shutdown. If the `application` process does not exist, `application:shutdown` calls `(exit-process` *exit-code*`)`.

Chapter 11

Database Interface

11.1 Introduction

The database (`db`) interface is a gen-server which provides a basic transaction framework to retrieve and store data in a SQLite database. It provides functions to use transactions (directly and lazily).

The low-level SQLite interface can be found in the operating system interface design (see Chapter 3).

Other SQLite resources are available online [26] or in The Definitive Guide to SQLite [22].

11.2 Theory of Operation

The `db` gen-server serializes internal requests to the database. For storage and retrieval of data, each transaction is processed in turn by a separate linked process. The gen-server does not block waiting for this process to finish so that it can maintain linear performance by keeping its inbox short. The return value of the transaction is returned to the caller or an error is generated without tearing down the gen-server.

To facilitate logging, the `db` gen-server can lazily open a transaction. In order to allow other processes access to the database, lazy transactions should be closed occasionally. To support this, it tracks a count of entries in the current transaction. A transaction is committed when the threshold of 10,000 is reached, the message queue of the `db` is empty, or when a direct transaction is requested. By default, each database is created with write-ahead logging enabled to prevent write operations from blocking on queries made from another connection.

SQLite has three types of transactions: deferred, immediate, and exclusive. This interface uses only immediate transactions to simplify the handling of the `SQLITE_BUSY` error. Using immediate transactions means that `SQLITE_BUSY` will only occur during `BEGIN IMMEDIATE`, `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK`[1] statements. For each of these statements, when a `SQLITE_BUSY` occurs, the code waits for a brief time, then retries the statement. The wait times in milliseconds follow the pattern `(2 3 6 11 16 21 26 26 26 51 51 . #0=(101 . #0#))`, and up to 500 retries are attempted before exiting with `#(db-retry-failed` *sql count*`)`. When the retry count is positive,

[1] Our testing showed that `ROLLBACK` returns `SQLITE_BUSY` only when a `COMMIT` for the same transaction returned `SQLITE_BUSY`. This framework never causes that situation to occur, but it guards against it anyway.

it is logged to the event manager along with the total duration with a `<transaction-retry>` event.

`<transaction-retry>` **event**

timestamp: timestamp from `erlang:now`
database: database filename
duration: duration in milliseconds
count: retry count
sql: query

The `db` gen-server uses the operating system interface to interact with SQLite. To prevent memory leaks, each raw database and statement handle is wrapped in a Scheme record and registered with a guardian via `make-foreign-handle-guardian`.

state `(define-state-tuple <db-state> filename db cache queue worker)`

- `filename` is the database specified when the server was started.
- `db` is the database record.
- `cache` is a hash table mapping SQL strings to SQLite prepared statements.
- `queue` is a queue of log and transaction requests.
- `worker` is the pid of the active worker or `#f`.

dictionary parameters

- `current-database` stores a Scheme record:

  ```
  (define-record-type database
    (fields
     (immutable filename)
     (immutable create-time)
     (mutable handle)))
  ```

  The `handle` is set to `#f` when the database is closed.

- `statement-cache` stores a Scheme record:

  ```
  (define-record-type cache
    (fields
     (immutable ht)
     (mutable waketime)
     (mutable lazy-statements)))
  ```

  The `waketime` is the next time the cache will attempt to remove dead entries.

  The hash table, `ht`, maps SQL strings to a Scheme record:

```
(define-record-type entry
  (fields
   (immutable stmt)
   (mutable timestamp)))
```

When a SQL string is not found in the cache, `osi_prepare_statement` is used with the `current-database` to make a SQLite statement. The raw statement handle is stored in a Scheme record:

```
(define-record-type statement
  (fields
   (immutable database)
   (immutable sql)
   (immutable create-time)
   (mutable handle)))
```

The statement is finalized using `osi_finalize_statement` when it is removed from the cache. `osi_close_database` will finalize any remaining statements associated with the database.

When a SQL string is found in the cache, the entry's `timestamp` is updated. Entries older than 5 minutes will be removed from the cache.

Accessing the cache may exit with reason reason `#(db-error prepare` *error sql*`)`, where *error* is a SQLite error pair.

The `lazy-statements` list contains `statement` records created by `lazy-execute`. These statements are finalized when a transaction completes.

`init` The `init` procedure takes a filename, mode symbol, and an initialization procedure and attempts to open that database and invoke the initialization procedure. The handle returned from `osi_open_database` is wrapped in a *database* record that is registered with a foreign-handle guardian using the type name `databases`. The foreign-handle guardian hooks the garbage collector so that dead databases are closed even if the `db` gen-server fails to close them for any reason.

The gen-server traps exits so that it can close the database in its `terminate` procedure.

`terminate` The `terminate` flushes the queue and closes the database.

`handle-call` The `handle-call` procedure processes the following messages:

- `#(transaction` *f*`)`: Add this transaction along with the *from* argument to `handle-call` to the queue. Process the queue.
- `filename`: Return the database filename.
- `stop`: Flush the queue and stop with reason `normal`, returning `stopped` to the caller.

`handle-cast` The `handle-cast` procedure processes the following message:

- `#(log` *sql bindings*`)`: Add this tuple to the queue. Process the queue.

`handle-info` The `handle-info` procedure processes messages matching the following patterns:

- `timeout`: Remove old entries from the statement cache.
- `` `(EXIT `` *worker-pid* `normal)`: The worker finished the previous request successfully. Process the queue.
- `` `(EXIT `` *worker-pid reason*`)`: The worker failed to process the previous request. Flush the queue and stop with *reason*.

11.3 Design Decisions

There is a one-to-one relationship between a SQLite database handle and the `db` gen-server. For clarity, the database handle and a SQLite statement cache are implemented in terms of Erlang process dictionary parameters.

An alternate approach for logging was already explored where a transaction was not lazily opened. Such an approach means that when a third party tool tries to access the database, it will hang until the transaction is complete.

A commit threshold of 10,000 was chosen because it was large enough to minimize the cost of a transaction but small enough to execute simple queries in less than one second.

11.4 Programming Interface

`(db:start&link` *name filename mode* [*db-init*]`)` **procedure**
returns: `#(ok` *pid*`)` | `#(error` *error*`)`

The `db:start&link` procedure creates a new `db` gen-server using `gen-server:start&link`.

name is the registered name of the process. For an anonymous server, `#f` may be specified.

filename is the path to a SQLite database.

mode is one of the following symbols used to pass SQLite flags to `osi_open_database`:

- `read-only` uses the SQLite flag `SQLITE_OPEN_READONLY`.
- `open` uses the SQLite flag `SQLITE_OPEN_READWRITE`.
- `create` combines the SQLite flags `(logor SQLITE_OPEN_READWRITE SQLITE_OPEN_CREATE)`.

db-init is a procedure that takes one argument, a database record instance. The return value is ignored. When the *mode* is `create` and *filename* is not a special SQLite filename, the default procedure sets `journal_mode` to "wal"; otherwise, no additional initialization occurs.

The SQLite constants can be found in `sqlite3.h` or online [26].

This procedure may return an *error* of `#(db-error open` *error filename*`)`, where *error* is a SQLite error pair.

`(db:stop` *who*`)` **procedure**
returns: `stopped`

The `db:stop` procedure calls `(gen-server:call` *who* `stop infinity)`.

`(with-db [`*db filename flags*`]` $body_1$ $body_2$ `...)` **syntax**

expands to:

```
(let ([db (sqlite:open filename flags)])
  (on-exit (sqlite:close db)
    body1 body2 ...))
```

The `with-db` macro opens the database in *filename*, executes the statements in the body, and closes the database before exiting. This is a suitable alternative to starting a `gen-server` when you need to query a database using a separate SQLite connection, and you do not need to cache prepared SQL statements.

`(db:filename` *who*`)` **procedure**

returns: the database filename

The `db:filename` procedure calls `(gen-server:call` *who* `filename)`.

`(db:log` *who sql* `.` *bindings*`)` **procedure**

returns: `ok`

The `db:log` procedure calls `(gen-server:cast` *who* `#(log` *sql bindings*`))`. *sql* is a SQL string, and *bindings* is a list of values to be bound in the query. Because `db:log` does not wait for a reply from the server, any error in processing the request will crash the server.

`(db:transaction` *who f*`)` **procedure**

returns: `#(ok` *result*`)` | `#(error` *error*`)`

The `db:transaction` procedure calls `(gen-server:call` *who* `#(transaction` *f*`) infinity)`.

f is a thunk which returns a single value, *result*. `execute`, `lazy-execute`, and `columns` can be used inside the procedure *f*.

result is the successful return value of *f*. Typically, this is a list of rows as returned by a `SELECT` query.

error is the failure reason of *f*.

`(transaction` *db body* `...)` **syntax**

expands to:

```
(match (db:transaction db (lambda () body ...))
  [#(ok ,result) result]
  [#(error ,reason) (throw reason)])
```

The `transaction` macro runs the body in a transaction and returns the result when successful and exits when unsuccessful.

`(execute` *sql* `.` *bindings*`)` **procedure**

returns: a list of rows where each row is a vector of data in column order as specified in the *sql* statement `execute` should only be used from within a thunk *f* provided to `db:transaction`.

sql is mapped to a SQLite statement using the `statement-cache`. The *bindings* are then applied using `osi_bind_statement`. The statement is then executed using `osi_step_statement`. The results are accumulated as a list, and the statement is reset using `osi_reset_statement` to prevent the statement from locking parts of the database.

This procedure may exit with reason `#(db-error prepare` *error sql*`)`, where *error* is a SQLite error pair.

`(lazy-execute` *sql* `.` *bindings*`)` **procedure**
returns: a thunk

`lazy-execute` should only be used from within a thunk *f* provided to `db:transaction`.

A new SQLite statement is created from *sql* using `osi_prepare_statement` so that the statement won't interfere with any other queries. The statement is added to the `lazy-statements` list of the `statement-cache` and is finalized when the transaction completes. The *bindings* are then applied using `osi_bind_statement`. A thunk is returned which, when called, executes the statement using `osi_step_statement`. The thunk returns one row of data or `#f`.

This procedure may exit with reason `#(db-error prepare` *error sql*`)`, where *error* is a SQLite error pair.

`(execute-sql` *db sql* `.` *bindings*`)` **procedure**
returns: a list of rows where each row is a vector of data in column order as specified in the *sql* statement

`execute-sql` should only be used for statements that do not need to be inside a transaction, such as a one-time query.

sql is prepared into a SQLite statement for use with *db*, executed via `sqlite:execute` with the specified *bindings*, and finalized.

This procedure may exit with reason `#(db-error prepare` *error sql*`)`, where *error* is a SQLite error pair.

`(columns` *sql*`)` **procedure**
returns: a vector of column names in order as specified in the *sql* statement

`columns` should only be used from within a thunk *f* provided to `db:transaction`.

sql is mapped to a SQLite statement using the `statement-cache`. The statement columns are then retrieved using `osi_get_statement_columns`.

`(parse-sql` *x* [*symbol->sql*]`)` **procedure**
returns: two values: a query string and a list of syntax objects for the arguments

The `parse-sql` procedure is used by macro transformers to take syntax object *x* and produce a query string and associated arguments according to the patterns below. When one of these patterns is matched, the *symbol->sql* procedure is applied to the remaining symbols of the input before they are spliced into the query string, as if by `(format "~a" (symbol->sql sym))`. By default, *symbol->sql* is the identity function.

- `(insert` *table* `([`*column* e_1 e_2 `...] ...)`

  The `insert` form generates a SQL insert statement. The *table* and *column* patterns are SQL identifiers. Any *e* expression that is `(unquote` *exp*`)` is converted to `?` in the query, and *exp* is added to the list of arguments. All other expressions are spliced into the query string.

- `(update` *table* `([`*column* e_1 e_2 `...] ...)` *where* `...)`

  The `update` form generates a SQL update statement. The *table* and *column* patterns are SQL identifiers. Any *e* or *where* expression that is `(unquote` *exp*`)` is converted to `?` in the query, and *exp* is added to the list of arguments. All other expressions are spliced into the query string.

- `(delete` *table* *where* `...)`

  The `delete` form generates a SQL delete statement. The *table* pattern is a SQL identifier. Any *where* expression that is `(unquote` *exp*`)` is converted to `?` in the query, and *exp* is added to the list of arguments. All other expressions are spliced into the query string.

`(database?` *x*`)` **procedure**

returns: a boolean

The `database?` procedure determines whether or not the datum *x* is a database record instance.

`(database-create-time` *db*`)` **procedure**

returns: a clock time in milliseconds

The `database-create-time` procedure returns the clock time from `erlang:now` when database record instance *db* was created.

`(database-filename` *db*`)` **procedure**

returns: a string

The `database-filename` procedure returns the filename of database record instance *db*.

`(database-count)` **procedure**

returns: the number of open databases

The `database-count` procedure returns the number of open databases. This is the procedure returned by `(foreign-handle-count 'databases)`.

`(print-databases` [*op*]`)` **procedure**

returns: unspecified

The `print-databases` procedure prints information about all open databases to textual output port *op*, which defaults to the current output port. This is the procedure returned by `(foreign-handle-print 'databases)`.

`(statement?` *x*`)` **procedure**

returns: a boolean

The `statement?` procedure determines whether or not the datum x is a statement record instance.

`(statement-create-time` *stmt*`)` **procedure**
returns: a clock time in milliseconds

The `statement-create-time` procedure returns the clock time from `erlang:now` when statement record instance *stmt* was created.

`(statement-database` *stmt*`)` **procedure**
returns: a string

The `statement-database` procedure returns the database record instance of the statement record instance *stmt*.

`(statement-sql` *stmt*`)` **procedure**
returns: a string

The `statement-sql` procedure returns the SQL string of the statement record instance *stmt*.

`(statement-count)` **procedure**
returns: the number of unfinalized statements

The `statement-count` procedure returns the number of unfinalized statements. This is the procedure returned by `(foreign-handle-count 'statements)`.

`(print-statements` [*op*]`)` **procedure**
returns: unspecified

The `print-statements` procedure prints information about all unfinalized statements to textual output port *op*, which defaults to the current output port. This is the procedure returned by `(foreign-handle-print 'statements)`.

`(sqlite:bind` *stmt bindings*`)` **procedure**
returns: unspecified

The `sqlite:bind` procedure binds the variables in statement record instance *stmt* with the list of *bindings*. It resets the statement before binding the variables.

`(sqlite:clear-bindings` *stmt*`)` **procedure**
returns: unspecified

The `sqlite:clear-bindings` procedure clears the variable bindings in statement record instance *stmt*.

`(sqlite:close` *db*`)` **procedure**
returns: unspecified

The `sqlite:close` procedure closes the database associated with database record instance *db*.

`(sqlite:columns` *stmt*`)` **procedure**
returns: a vector of column names

The `sqlite:columns` procedure returns a vector of column names for the statement record instance *stmt*.

(`sqlite:execute` *stmt bindings*) **procedure**

returns: a list of rows where each row is a vector of data in column order

The `sqlite:execute` procedure calls (`sqlite:bind` *stmt bindings*) to bind any variables and then iteratively calls (`sqlite:step` *stmt*) to build the resulting list of rows. It resets the statement when the procedure exits.

(`sqlite:expanded-sql` *stmt*) **procedure**

returns: a string

The `sqlite:expanded-sql` procedure returns the SQL string expanded with the binding values for the statement record instance *stmt*.

(`sqlite:finalize` *stmt*) **procedure**

returns: unspecified

The `sqlite:finalize` procedure finalizes the statement record instance *stmt*.

(`sqlite:interrupt` *db*) **procedure**

returns: unspecified

The `sqlite:interrupt` procedure interrupts any pending operations on the database associated with database record instance *db*.

(`sqlite:last-insert-rowid` *db*) **procedure**

returns: unspecified

The `sqlite:last-insert-rowid` procedure returns the rowid of the most recent successful insert into a rowid table or virtual table on the database associated with database record instance *db*. It returns 0 if no such insert has occurred.

(`sqlite:open` *filename flags*) **procedure**

returns: a database record instance

The `sqlite:open` procedure opens the SQLite database in file *filename* with *flags* specified by `sqlite3_open_v2` [26]. The constants `SQLITE_OPEN_CREATE`, `SQLITE_OPEN_READONLY`, and `SQLITE_-OPEN_READWRITE` are exported from the `(swish db)` library. The `sqlite:open` procedure registers the database record with a foreign-handle guardian using the type name `databases`.

(`sqlite:prepare` *db sql*) **procedure**

returns: a statement record instance

The `sqlite:prepare` procedure returns a statement record instance for the *sql* statement in the database record instance *db*. The `sqlite:prepare` procedure registers the statement record with a foreign-handle guardian using the type name `statements`.

(`sqlite:sql` *stmt*) **procedure**

returns: a string

The `sqlite:sql` procedure returns the unexpanded SQL string for the statement record instance *stmt*.

`(sqlite:step` *stmt*`)` **procedure**

returns: a vector of data in column order or `#f`

The `sqlite:step` procedure steps the statement record instance *stmt* and returns the next row vector in column order or `#f` if there are no more rows.

Chapter 12

Log Database

12.1 Introduction

The log database is a single gen-server named `log-db` that uses the database interface (see Chapter 11) to log system events (see Chapter 7).

12.2 Theory of Operation

12.2.1 Initialization

The `log-db` gen-server handles startup and setup through two separate procedures. Startup uses the `db:start&link` procedure to connect to the the SQLite log database specified by the `(log-file)` parameter. It creates the file if it does not exist, but otherwise startup does not modify the database.

Setup makes sure the schema of the log database has been created and is up-to-date. A unique symbol identifying the schema version is stored in a table named version. This allows the software to upgrade between known schema versions and to exit with an error when it encounters an unsupported database version. These schema updates happen within a database transaction so that if there is an error, the changes are rolled back.

Setup calls `event-mgr:set-log-handler` after updating the schema. This registers the `log-db` to log system events. It also calls `event-mgr:flush-buffer`. This causes the event manager to stop buffering startup events and the `log-db` to log the events that were buffered.

Finally, setup sends a `<system-attributes>` event so that `log-db` receives and logs it.

Once the `log-db` gen-server has been setup, it continues to receive events from the system event manager. It converts events that it recognizes into insertions to the log database. Events that it does not recognize are ignored.

The tables are pruned using insert triggers to hold 90 days of information. To keep the insert operations fast, the timestamp columns are indexed, and the pruning deletes no more than 10 rows per insert.

12.2.2 Extensions

An application typically produces events beyond those that are part of Swish and may wish to log them in the same log database file where the Swish events are logged. The `log-db` design allows for this type of extension.

The `log-db:setup` procedure takes a list of `<event-logger>` tuples. Each logger represents an extension to the log database schema and contains two procedures, `setup` and `log`. The `log-db:setup` procedure calls the `setup` procedure of logger to make sure that its portion of the schema has been created and is up-to-date. Then, when `log-db` receives an event, it calls the `log` procedure of each logger. If the event is recognized by that portion of the schema, the `log` procedure inserts or updates data in the log database. Otherwise, the procedure ignores that event.

Additionally, the version table does not store a single schema version. Instead, it stores schema versions associated with names. The `setup` procedure of an `<event-logger>` uses an unique name for its portion of the schema and the `log-db:version` procedure to retrieve and set its version.

The schema and logging for Swish events is implemented as an `<event-logger>` defined by `swish-event-logger` and using the schema version name `swish`. An application that wishes to use this logging must provide `swish-event-logger` in the list to `log-db:setup`. If the application wishes to log Swish events in a different structure, it can omit the `swish-event-logger` and provide its own logger with its own schema. However, doing so makes the application more brittle with respect to changes in the Swish implementation.

12.3 Programming Interface

`<event-logger>` **tuple**

setup: procedure of no arguments that makes sure this portion of the schema is created and up-to-date

log: procedure of one argument, an event, that logs the event if it recognizes it and otherwise ignores it

`(log-db:start&link)` **procedure**

returns: `#(ok` *pid*`)` | `#(error` *error*`)`

The `log-db:start&link` procedure creates a new `db` gen-server named `log-db` using `db:start&link`. It uses the value of the `(log-file)` parameter as the path to the SQLite database and specifies `create` mode.

`(log-db:setup` *loggers*`)` **procedure**

returns: `ignore` | `#(error` *error*`)`

The argument *loggers* is a list of `<event-logger>` tuples. The `log-db:setup` makes sure the `log-db` is setup to run by doing the following in order.

1. Initialize or upgrade the database schema from within a `db:transaction` call. It does this by calling the `setup` procedure of each logger.

2. Register a procedure with `event-mgr:set-log-handler` to have the `log-db` gen-server log events it recognizes. When this procedure receives an event, it calls the `log` procedure of each logger.
3. Call `event-mgr:flush-buffer` to stop buffering system events and apply the log handler to the events already buffered.
4. Send a `<system-attributes>` event.

If everything succeeds, the procedure returns `ignore`. If either the `db:transaction` or `event-mgr:set-log-handler` indicate an error, the procedure returns that error.

`(log-db:version` *name* [*version*]`)` **procedure**

name: symbol identifying the schema
version: string specifying the version of the schema

When called with one argument, `log-db:version` retrieves the version associated with *name* from the database and returns it as a string. It returns `#f` if no version associated with *name* is stored in the database.

When called with two arguments, it stores *version* as the version associated with *name* in the database.

`(log-db:get-instance-id)` **procedure**
returns: a string

`log-db:setup` associates a globally unique identifier with the database file. The `log-db:get-instance-id` function caches and returns that identifier.

`swish-event-logger` **property**

The `swish-event-logger` is an `<event-logger>` tuple that defines the schema for Swish events. It uses the name `swish` to store its schema version.

`(create-table` *name* **syntax**
  `(`*field type* . *inline*`)`
  ...`)`
expands to: `(execute "create table if not exists ...")`

The `create-table` syntax describes the schema of a single table and expands into a call to `execute` to create the table if no table with that name already exists. The name of the table, *name*, and of each field, *field*, are converted from Scheme to SQL identifiers by replacing hyphen characters with underscores and eliminating any non-alphanumeric and non-underscore characters. The SQL definition of each field is produced by joining the converted field name, the *type* and any additional *inline* arguments into a space separated string.

`(define-simple-events` *create handle* **syntax**
  `(`*name clause* ...`)`
  ...`)`
expands to: A definition of the *create* and *handle* procedures

The `define-simple-events` syntax is used to log tuple types by inserting a row into a table with the same name and the same fields. Each *name* is a tuple type. Each *clause* is a valid `create-table` clause for one of the fields in that tuple type.

It defines *create* as a procedure of 0 arguments that consists of a `(create-table` *name clause* ...`)` for each tuple in the `define-simple-events`. This means that the name of the tuple type and each field are converted to SQL names by the `create-table` syntax.

It defines *handle* as a procedure of 1 argument, an event. If the event is one of the tuple types in the `define-simple-events`, it calls `db:log` with an insert statement applying `coerce` to each value. If the event is unrecognized, it returns `#f`.

`(coerce` *x*`)`	**procedure**

returns: a Scheme object

The argument *x* is a Scheme object mapped to a SQLite value.

type	transformation
string	*string*
bytevector	*bytevector*
number	*number, if it fits in 64 bits*
symbol	`symbol->string`
date	`format-rfc2822`
JSON object	`json:object->string`
process	`global-process-id`
condition	a string representing a JSON object with the following fields: `message` containing `(exit-reason->english` *x*`)` and `stacks` containing `(map stack->json (exit-reason->stacks` *x*`))`
continuation-condition	a string containing `#(error` *reason stack*`)` where the *stack* is obtained from `dump-stack`

`coerce` passes `#f` through unmodified which SQLite interprets as NULL. Other values are converted to string using `write`.

`(stack->json` *k* [*max-depth*]`)`	**procedure**

returns: a JSON object

The `stack->json` procedure renders the stack of continuation *k* as a JSON object by calling `walk-stack`. The return value may contain the following keys:

`type`	`"stack"`
`depth`	the depth of the stack
`truncated`	if present, the *max-depth* at which the stack dump was truncated
`frames`	if present, a list of JSON objects representing stack frames

A stack frame may contain the following keys:

`type`	`"stack-frame"`
`depth`	the depth of this frame
`source`	if present, a source object for the return point
`procedure-source`	if present, a source object for the procedure containing the return point
`free`	if present, a list of JSON objects representing free variables

A source object *x* with source file descriptor *sfd* is represented by a JSON object containing the following keys:

`bfp`	`(source-object-bfp` *x*`)`
`efp`	`(source-object-efp` *x*`)`
`path`	`(source-file-descriptor-path` *sfd*`)`
`checksum`	`(source-file-descriptor-checksum` *sfd*`)`

A free variable with value *val* is represented by a JSON object containing the following keys:

`name`	a string containing the variable name or its index
`value`	the result of `(format "~s"` *val*`)`

`(json-stack->string` [*op*] *x*`)` **procedure**

returns: see below

The two argument form of `json-stack->string` prints the stack represented by JSON object *x* to the textual output port *op*. The single argument form of `json-stack->string` prints the stack represented by JSON object *x* to a string output port and returns the resulting string. In either case, the printed form resembles that generated by `dump-stack` except that source locations are given as file offsets rather than line and character numbers.

12.4 Published Events

`<system-attributes>` **event**

timestamp: timestamp from `erlang:now`
date: date from `current-date`
software-info: JSON object from `software-info`
machine-type: `(symbol->string (machine-type))`
computer-name: computer name from `osi_get_hostname`

The `<system-attributes>` event is sent exactly once, when `log-db:setup` is called.

Chapter 13

System Statistics

13.1 Introduction

The system uses a single gen-server named `statistics` to periodically query statistics about the system, such as memory usage.

13.2 Theory of Operation

When the `statistics` gen-server starts, it posts a `<statistics>` event with `reason` = `startup`. Every five minutes thereafter, it posts a `<statistics>` event with `reason` = `update`. If the computer sleeps or hibernates, the gen-server posts a `<statistics>` event with `reason` = `suspend`. When the computer awakens, the gen-server posts a `<statistics>` event with `reason` = `resume`. When the gen-server terminates, it posts a `<statistics>` event with `reason` = `shutdown`.

The `<statistics>` event is handled by the `log-db` gen-server (see Chapter 12), which adds the data to the statistics table in the log database.

13.3 Programming Interface

`(statistics:start&link)` **procedure**

returns: `#(ok` *pid*`)` | `#(error` *error*`)`

The `statistics:start&link` procedure creates a new gen-server named `statistics` using `gen-server:start&link`. It then posts a `<statistics>` event with `reason` = `startup`.

`(statistics:resume)` **procedure**

The `statistics:resume` procedure casts a message to the `statistics` gen-server that causes it to publish a `<statistics>` event with `reason` = `resume`. This procedure is called from the operating system interface via the top-level-value `$resume`.

`(statistics:suspend)`	**procedure**

The `statistics:suspend` procedure casts a message to the `statistics` gen-server that causes it to publish a `<statistics>` event with `reason` = `suspend`. This procedure is called from the operating system interface via the top-level-value `$suspend`.

13.4 Published Events

`<statistics>`	**event**

timestamp: timestamp from `erlang:now`
date: date from `current-date`
reason: `startup`, `update`, `suspend`, `resume`, or `shutdown`
bytes-allocated: Scheme heap size from `bytes-allocated`
osi-bytes-used: C heap size from `osi_get_bytes_used`
sqlite-memory: SQLite memory used from `osi_get_sqlite_status`
sqlite-memory-highwater: SQLite memory highwater since last event from `osi_get_sqlite_status`
foreign-handles: JSON object with types and counts reported by `count-foreign-handles`
cpu: CPU time in seconds since last event
real: elapsed time in seconds since last event
bytes: Scheme heap bytes allocated since last event
gc-count: number of garbage collections since last event
gc-cpu: CPU time in seconds of garbage collections since last event
gc-real: elapsed time in seconds of garbage collections since last event
gc-bytes: Scheme heap bytes reclaimed since last event

The JSON object in the `foreign-handles` field contains at least the following entries:

`databases`	number of open SQLite databases
`osi-ports`	number of open osi-ports
`path-watchers`	number of open path watchers
`statements`	number of allocated SQLite statements
`tcp-listeners`	number of open TCP/IP listeners

This event is sent every five minutes while the `statistics` gen-server is running.

Chapter 14

HTTP Interface

14.1 Introduction

The HTTP interface provides a basic implementation of the Hypertext Transfer Protocol [14] and the WebSocket Protocol [13]. The programming interface includes procedures for the HyperText Markup Language (HTML) version 5 [19] and JavaScript Object Notation (JSON) [3].

14.2 Theory of Operation

The HTTP interface provides `http:add-file-server` to listen for connections on a TCP port and serve files from a directory and `http:add-server` to serve content via a user-defined procedure. These procedures configure the `app-sup-spec` to include one or more web servers in the supervision hierarchy that is started by `app:start`.

Both `http:add-file-server` and `http:add-server` are implemented in terms of `http:configure-server`, which defines a supervisor that manages the *listener* gen-server, *dispatcher* processes, *cache* gen-servers, and *evaluator* processes. Internally, the listener starts a cache manager process to support evaluating and serving files. The dispatcher starts a connection process to manage protocol details. This structure is illustrated in Figure 14.1.

The HTTP supervisor provided via `http:configure-server` is configured one-for-one with up to 10 restarts within 10 seconds.

The *listener* gen-server awaits TCP connections using `listen-tcp` and accepts them using `accept-tcp`. For each TCP connection, the listener uses its supervisor to spawn and link a *dispatcher* process that reads each request, calls the *URL handler* configured for that listener, and repeats until the connection closes. The listener closes the associated output port when the dispatcher exits.

Each *dispacher* process spawns a *connection* gen-server. The connection is responsible for reading and parsing requests in a timely manner. A connection server reads from its input port until a CR LF occurs. Well-formed input is converted to a `<request>` tuple. The HTTP request header is then read and associated with the request. If the connection takes more than `request-timeout` milliseconds to parse a request, the dispatcher fires an exception, and the TCP connection is closed.

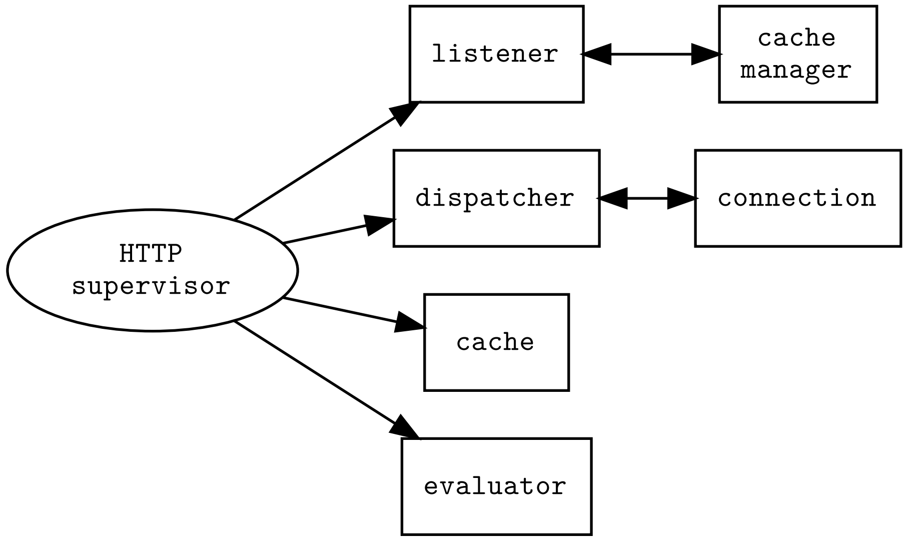

Figure 14.1: HTTP supervision tree

The dispatcher logs the specific request, validates that the requested path does not include "..", and invokes the URL handler provided by the listener.

For forms, the URL handler is responsible for checking the request method and reading form content with `http:call-with-form`. Any unread content as determined by `Content-Length` is ignored.

After a request is processed, the current process and connection can be reused. Unless the `Connection` header specifies `close`, the system reads another request from the input port after skipping any unread content from the current request.

14.2.1 URL handler and Media Type handler

Using `http:add-server` or `http:configure-server` configures a web server to serve content via a URL handler that may be compiled into the Swish application.

A *URL handler* is a procedure that should return `#t` when it processes a request, and `#f` otherwise.

To allow future extension without change to application code, the HTTP interface provides `http:url-handler` to define a handler which exposes the following implicit variables:

conn: a connection process
request: a `<request>` tuple
header: a JSON object
params: a JSON object

`http:call-url-handler` can be used to invoke a handler, and `http:compose-url-handlers` chains a sequence of handlers together.

A *media type handler* is a procedure that takes a file extension as input and returns a media type or `#f`. The default media type handler always returns `#f`.

14.2.2 Default file handling

Using `http:add-file-server` or `http:configure-file-server` configures a web server to serve files from a directory. These procedures generate a URL handler by calling `http:make-default-file-url-handler`. This handler asks the *cache manager* gen-server for a cache responsible for the given directory.

The *cache manager* gen-server maintains the mapping of the top-level web directory to a *cache* gen-server. If a cache does not exist for the given path, a new *cache* gen-server is created.

A *cache* gen-server stores URL handlers and provides a mapping from file extension to media type. It creates a directory watcher using `watch-directory` to invalidate the cache when anything in the directory tree changes. It terminates after 5 minutes of inactivity.

The optional `media-type-handler` is configured via `http:make-default-media-type-handler` to match the extension of *filename* case insensitively against an extension in the `mime-types` file of the configured directory. Each line of `mime-types` has the form `("`*extension*`" . "`*Content-Type*`")`.

14.2.3 Dynamic Pages

A *dynamic page* is any file that is transformed before its content is sent to the browser. For example, by setting the option `file-transform` to `http:enable-dynamic-pages`, a path that ends in ".ss" containing a sequence of definitions followed by a sequence of expressions is wrapped in a URL handler and evaluated by the `eval` system procedure. The page explicitly sends a response via `http:respond` or `http:respond-file`.

In general, a cache gen-server resolves a URL to a file path and checks its internal state. If a URL handler is unavailable, the gen-server calls the optional `file-transform` procedure passing the absolute file path.

The `file-transform` procedure returns `#f` or a procedure. When it returns `#f`, the URL is considered static and is sent directly over the connection using `http:respond-file`.

When it returns a procedure, the cache starts an *evaluator* process. The evaluator process calls the procedure with the cache's root directory and the absolute file path. The evaluator procedure returns `#f` or a URL handler. The cache stores the URL handler.

14.2.4 WebSocket Protocol

The WebSocket Protocol enables bidirectional communication and is well supported by web browsers and software platforms. The protocol utilizes message fragmentation and control frame messages. The `(swish websocket)` library represents a WebSocket connection as a gen-server referred to as a *websocket*.

The *websocket* gen-server manages message fragmentation, ping and pong control messages, and sends messages to a separate process. The following messages are sent:

- `#(ws:init` *ws*`)`: Indicates that the HTTP protocol upgrade to websocket is complete for process *ws* and messages can be exchanged.

- `#(ws:message` *ws message*`)`: Indicates that a *message* bytevector or string was received by process *ws*.

- `#(ws:closed` *ws code reason*`)`: Indicates that the websocket and underlying output port is closed. The cause is indicated by the numeric status *code*. Status codes 1000 and 1001 are used to indicate normal exit reasons. The *reason* string may be useful for logging and debugging but is not necessarily presentable to a user.

As a server, a URL handler can call `ws:upgrade` in tail position to switch to the WebSocket Protocol.

A client establishes a connection to a server using `ws:connect` to connect and request usage of the WebSocket Protocol.

14.3 Security

The HTTP interface is written in Scheme, and therefore buffer overrun exploits cannot be used against the system.

User input should be carefully checked before calling `eval` or invoking a database query.

A URL which directs the system away from a cache's root path using ".." could allow access to system files. Therefore, the dispatcher explicitly rejects relative paths.

The HTTP interface limits incoming data to protect against large memory allocation that may crash the application.

The HTTP interface allows individual form URL handlers to specify incoming data and file upload limits to protect against large memory allocation and running out of disk space.

14.4 Programming Interface

`<request>` **tuple**

host: a string
method: a symbol
path: a decoded string
original-path: same as *path*
header: a decoded JSON object
params: a decoded JSON object

The *original-path* is used internally when redirecting the browser to a new location. If user code copies the `<request>` tuple, it should not modify *original-path*.

`(http:configure-server` *name port url-handler* [*options*]`)` **procedure**
returns: a list of child specifications (see page 105)

name: a symbol for the listener gen-server or `#f` for an anonymous server
port: a fixnum $0 \leq port \leq 65535$
url-handler: a procedure
options: see below

The `http:configure-server` procedure returns a list of supervisor child specifications that define an HTTP server.

The specification defines a supervisor configured as one-for-one with up to 10 restarts every 10 seconds. The supervisor starts a listener gen-server. This supervisor is used to create dispatcher, cache, and evaluator processes.

The listener is a gen-server registered as *name*. It accepts connections on the given TCP *port* and processes them with *url-handler*.

The *options* can be defined using `(http:options (`*option value*`) ...)`. The following options may be used:

option	default	description
`request-limit`	8,192	a positive fixnum; the maximum length in bytes of the first line of an HTTP request
`request-timeout`	30,000	a positive fixnum; the maximum number of milliseconds to wait to receive an HTTP request after initial connection
`response-timeout`	60,000	a positive fixnum; the maximum number of milliseconds to allow the server to send a response
`header-limit`	1,048,576	a positive fixnum; the maximum length in bytes of the HTTP request headers
`media-type-handler`	`(lambda (ext) #f)`	a procedure; given a file extension returns the media type of the file or `#f`
`file-search-extensions`	`'(".html")`	a list of strings; a cache process uses the list to disambiguate URL paths to file system paths.
`file-transform`	`(lambda (path) #f)`	a procedure; a cache process calls this procedure with a file path and expects an evaluator procedure or `#f` (see §14.2.3)

`(http:add-server` *arg* `...)` **syntax**

The `http:add-server` macro expands to code that appends the result of `(http:configure-server` *arg* `...)` to the `app-sup-spec` parameter.

`(http:configure-file-server` *name port dir* [*options*]`)` **procedure**
returns: a list of child specifications (see page 105)

The `http:configure-file-server` procedure defines an HTTP server that provides content from the file-system rooted at *dir*.

The `http:configure-file-server` procedure calls `http:configure-server` with `(http:make-default-file-url-handler` *dir*`)` as the URL handler. If the `media-type-handler` option is not specified, it defaults to one constructed via `(http:make-default-media-type-handler` *dir*`)`.

`(http:add-file-server` *arg* `...)` **syntax**

The `http:add-file-server` macro expands to code that appends the result of `(http:configure-file-server` *arg* `...)` to the `app-sup-spec` parameter.

`(http:make-default-file-url-handler` *dir*`)` **procedure**
returns: a URL handler

The `http:make-default-file-url-handler` defines a URL handler that retrieves the cache responsible for the directory *dir*. It then looks up a URL handler from the cache for the given request. If found, it calls the handler and flushes the output port. Otherwise, it returns `#f`.

`(http:make-default-media-type-handler` *dir*`)` **procedure**
returns: a media type handler

The `http:make-default-media-type-handler` defines a media type handler that retrieves the cache responsible for the directory *dir*. It then requests the media type for the given file extension. If found, the type is returned. Otherwise, it returns `#f`.

`(http:url-handler` $body_1$ $body_2$ `...)` **syntax**

The `http:url-handler` macro defines a URL handler procedure described in Section 14.2.1.

`(http:call-url-handler` *handler*`)` **syntax**

The `http:call-url-handler` macro invokes a URL handler, implicitly capturing variables in scope as described in Section 14.2.1.

`(http:compose-url-handlers` *handlers*`)` **procedure**
returns: a URL handler

The `http:compose-url-handlers` procedure defines a URL handler that takes a list of *handlers* and invokes each handler, in order, until one returns `#t`. Otherwise, it returns `#f`.

If a URL handler returns `#f` but sends output to the output-port, exception `http-side-effecting-handler` is raised.

`(http:get-port-number` *listener*`)` **procedure**
returns: the listener gen-server TCP port number

The `http:get-port-number` procedure calls `(gen-server:call` *listener* `'get-port-number)`. If the listener's port was configured to be zero, the operating system will choose an available port number. `http:get-port-number` uses `listener-port-number` to retrieve the actual port number that the server is listening on.

`(http:find-header` *name* *header*`)` **procedure**
returns: a string | `#f`

The `http:find-header` procedure returns the value associated with *name* in JSON object *header* if present and `#f` otherwise. If *name* is a symbol, the lookup uses `eq?`. If *name* is a string, it is first mapped to a lower-case symbol before performing the lookup. Otherwise, the exception `#(bad-arg http:find-header` *name*`)` is raised.

(`http:get-header` *name header*) **procedure**
returns: a string

The `http:get-header` procedure returns the value associated with *name* in JSON object *header* if present and raises exception `#(http-invalid-header` *name*`)` otherwise. If *name* is a symbol, the lookup uses `eq?`. If *name* is a string, it is first mapped to a lower-case symbol before performing the lookup. Otherwise, the exception `#(bad-arg http:get-header` *name*`)` is raised.

(`http:get-content-length` *header*) **procedure**
returns: an unsigned integer | `#f`

The `http:get-content-length` procedure returns the unsigned integer value associated with `content-length` in JSON object *header* if present and `#f` otherwise. It raises exception `#(http-invalid-content-length` *value*`)` if the *value* string does not represent an unsigned integer.

(`http:find-param` *name params*) **procedure**
returns: a string | `#f`

The `http:find-param` procedure returns the value associated with *name* in JSON object *params* if present and `#f` otherwise. If *name* is a symbol, the lookup uses `eq?`. If *name* is a string, it is converted to a symbol before performing the lookup. Otherwise, the exception `#(bad-arg http:find-param` *name*`)` is raised.

(`http:get-param` *name params*) **procedure**
returns: a string

The `http:get-param` procedure returns the value associated with *name* in JSON object *params* if present and raises exception `#(http-invalid-param` *name*`)` otherwise. If *name* is a symbol, the lookup uses `eq?`. If *name* is a string, it is converted to a symbol before performing the lookup. Otherwise, the exception `#(bad-arg http:get-param` *name*`)` is raised.

(`http:read-header` *ip limit*) **procedure**
returns: a JSON object

The `http:read-header` procedure reads from the binary input port *ip* until it reads a blank line. It creates a JSON object where the key is a symbol formed from the down-cased characters before the first colon and the value is a string formed from the characters that remain after skipping white space. Duplicate message-header fields result in a single entry where the value is a comma-separated list.

Reading beyond *limit* raises exception `http-input-limit-exceeded`.

Failure to find a colon on any given line raises exception `http-invalid-header`.

(`http:read-status` *ip limit*) **procedure**
returns: number | `#f`

The `http:read-status` procedure reads the HTTP response status line from the binary input port *ip* and returns the status code as a number if well formed and `#f` otherwise. Reading beyond *limit* raises exception `http-input-limit-exceeded`.

(`http:write-status` *op* *status*) **procedure**
returns: unspecified

The `http:write-status` procedure writes the HTTP response status line to the binary output port *op*.

Unless *status* is a fixnum and $100 \leq status \leq 599$, the exception `#(bad-arg http:write-status` *status*`)` is raised.

According to HTTP [14] the status line includes a human-readable reason phrase. The grammar shows that it can in fact be zero characters long; therefore, the reason phrase is not included in this implementation.

(`http:write-header` *op* *header-alist*) **procedure**
returns: unspecified

The `http:write-header` procedure writes the HTTP *header-alist* and trailing CR LF to the binary output port *op*.

header-alist is an association list whose keys and values are strings. If any of its keys are not strings, exception `#(bad-arg http:write-header` *header-alist*`)` is raised.

(`http:respond` *conn* *status* *header-alist* [*content* [*timeout*]]) **procedure**
returns: `#t`

The `http:respond` procedure writes the HTTP *status* and *header-alist* to the connection *conn* using `http:write-status` and `http:write-header`, adding `Content-Length` to *header-alist* in situations described below. When `Cache-Control` is not present in *header-alist*, it is added with value `no-cache`. When *content* is a bytevector, it is written. Then the output port is flushed.

The default value of *content* is `#f`, which specifies that the `Content-Length` header is not added. When *status* is 100–199, 204, or 304, the `Content-Length` header is not added, and if *content* is a bytevector, it must be empty. Otherwise, the `Content-Length` header is added with the length of *content*.

The default value of *timeout* is `(response-timeout)`.

(`http:respond-file` *conn* *status* *header-alist* *filename* [*timeout*]) **procedure**
returns: `#t`

The `http:respond-file` procedure writes the HTTP *status* and *header-alist* to the connection *conn* using `http:write-status` and `http:write-header`, adding `Content-Length` to *header-alist*. The `Cache-Control` header is added, if it is not already present, with value `max-age=3600`. The `Content-Type` header is added, if it is not already present, by calling the `media-type-handler`. The content of the file is streamed to the output port so that the file does not need to be loaded into memory. The output port is flushed.

The default value of *timeout* is `(response-timeout)`.

`(http:call-with-form` *conn header content-limit file-limit files f* [*timeout*]`)` **procedure**
returns: see below

The `http:call-with-form` procedure checks JSON object *header* for a `content-type` of `multipart/form-data` or `application/x-www-form-urlencoded`. It parses form name/value pairs into a JSON object. After parsing the form data, the function *f* is called with the JSON object. After *f* returns, all uploaded files are deleted, and the return value is passed to the caller.

Because `http:call-with-form` is used inside a URL handler after the `<http-request>` event has been fired, the specific form data is not logged. Applications should consider the sensitivity of submitted data before logging.

When files are sent and the form variable name appears in the *files* list, the file is stored in `(tmp-dir)`, and the JSON value is a JSON object with `type="file"` and `filename=`*filename*.

For example, a form sending the variables `name`, `rank`, and `image` might look like:

```
{
  "name": "Steve R",
  "rank": "Captain",
  "image":
  {
    "type": "file",
    "filename": filename
  }
}
```

Name/value pairs count against *content-limit* while file data counts against *file-limit*. An exception is raised when either limit is exceeded.

The default value of *timeout* is `(request-timeout)`.

`(http:call-with-ports` *conn f* [*timeout*]`)` **procedure**
returns: see below

The `http:call-with-ports` procedure ignores any current connection state and calls *f* in the connection process with the connection's input and output ports. Running *f* in the connection process allows the caller to timeout when input is unavailable. The return value of *f* is returned to the caller.

The default value of *timeout* is 5000 milliseconds.

`(http:switch-protocol` *proc*`)` **procedure**
returns: see below

When used in a URL handler, `http:switch-protocol` returns a special value to control the dispatcher loop. The dispatcher detaches from the connection process and calls *proc* with the input and output ports.

`(http:percent-encode` *s*`)` **procedure**
returns: an encoded string

The `http:percent-encode` procedure writes the characters `A`–`Z`, `a`–`z`, `0`–`9`, hyphen, underscore, period, and `~`. Other characters are converted to a `%` prefix and two digit hexadecimal representation.

(`http:enable-dynamic-pages` *path*) **procedure**
returns: a procedure | `#f`

If the file extension of *path* is ".ss", the `http:enable-dynamic-pages` procedure returns the `http:eval-dynamic-page` procedure. Otherwise, it returns `#f`.

(`http:eval-dynamic-page` *root-dir abs-path*) **procedure**
returns: a URL handler

The `http:eval-dynamic-page` procedure wraps a `http:url-handler` around the contents of a file, extends the syntax with the constructs described in Section 14.4.1, and calls the `eval` system procedure.

14.4.1 Dynamic Page Constructs

A dynamic page exposes the same implicit variables as a URL handler described in Section 14.2.1 as well as the following additional syntax.

(`find-param` *key*) **syntax**

Implementation: The `find-param` macro expands to (`http:find-param` *key params*).

(`get-param` *key*) **syntax**

Implementation: The `get-param` macro expands to (`http:get-param` *key params*).

(`http:include "`*filename*`"`) **syntax**

The `http:include` construct includes the definitions from *filename*, a path relative to the root path of the cache if *filename* begins with a forward slash, else relative to the directory of the current file.

Implementation: The `http:include` macro calls `read-file` and `read-bytevector` to retrieve a list of expressions that are spliced in at the same scope as the use of `http:include`. The splicing is done with `let-syntax` so that any nested `http:include` expressions are processed relative to the directory of *filename*.

14.4.2 WebSocket Protocol

(`ws:upgrade` *conn request process* [*options*]) **procedure**
returns: see below

The `ws:upgrade` procedure checks the header of the *request* tuple and upgrades the calling process to a gen-server that speaks the WebSocket protocol. The connection *conn* is used during the opening handshake to report errors. As it runs, the gen-server sends the messages defined in Section 14.2.4 to the given *process*. It raises an exception upon termination (see `gen-server:enter-loop`).

The *options* are defined with `(ws:options (`*option value*`) ...)`. The following options may be used:

option	default	description
`fragmentation-size`	1,048,576	a positive fixnum or `#f` to disable; the maximum length in bytes of a single message payload
`maximum-message-size`	16,777,216	a positive fixnum; the maximum length in bytes of the total payload for a single message that can be read from input
`ping-frequency`	30,000	a positive fixnum; the maximum number of milliseconds since last receiving a message before issuing a ping request
`pong-timeout`	5,000	a positive fixnum; the maximum number of milliseconds to wait for a reply to a ping before failing

`(ws:connect` *hostname port request process* [*options*]`)` **procedure**
returns: a websocket gen-server process

The `ws:connect` procedure initiates a TCP connection to *hostname* on the given *port* number and issues an upgrade for the *request* uniform resource identifier string. It then spawns and returns a gen-server process that sends the messages defined in Section 14.2.4 to the given *process*.

The *options* are the same as defined for `ws:upgrade`.

`(ws:send` *server message*`)` **procedure**
returns: `ok`

The `ws:send` procedure uses `gen-server:cast` to transmit the bytevector or string *message* to the websocket process or registered name *server*.

`(ws:close` *server*`)` **procedure**
returns: `ok`

The `ws:close` procedure uses `gen-server:cast` to close the websocket process or registered name *server*.

14.4.3 HyperText Markup Language

`(html:encode` *s*`)` **procedure**
`(html:encode` *op s*`)`
returns: see below

The `html:encode` procedure converts special character entities in string *s*.

input	output
"	"
&	&
<	<
>	>

The single argument form of `html:encode` returns an encoded string.

The two argument form of `html:encode` sends the encoded string to the textual output port *op*.

(html->string *x*)	**procedure**
(html->string *op* *x*)	

returns: see below

The `html->string` procedure transforms an object into HTML. The transformation, *H*, is described below:

x	*H*(*x*)
()	nothing
#!void	nothing
string	*E*(*string*)
number	*number*
(begin *pattern* ...)	*H*(*pattern*)...
(cdata *string* ...)	[!CDATA[*string*...]]
(html5 [(@ *attr* ...)] *pattern* ...)	<!DOCTYPE html><html *A*(*attr*) ...>*H*(*pattern*)...</html>
(raw *string* ...)	*string*...
(script [(@ *attr* ...)] *string* ...)	<script *A*(*attr*) ...>*string*...</script>
(style [(@ *attr* ...)] *string* ...)	<style *A*(*attr*) ...>*string*...</style>
(*tag* [(@ *attr* ...)] *pattern* ...)	<*tag* *A*(*attr*) ...>*H*(*pattern*)...</*tag*>
(*void-tag* [(@ *attr* ...)])	<*void-tag* *A*(*attr*) ...>

E denotes the `html:encode` function.

A *void-tag* is one of `area`, `base`, `br`, `col`, `embed`, `hr`, `img`, `input`, `keygen`, `link`, `menuitem`, `meta`, `param`, `source`, `track`, or `wbr`. A *tag* is any other symbol.

The attribute transformation, *A*, is described below, where *key* is a symbol:

attr	*A*(*attr*)
#!void	nothing
(*key*)	*key*
(*key* *string*)	*key*="*E*(*string*)"
(*key* *number*)	*key*="*number*"

The single argument form of `html->string` returns an encoded HTML string.

The two argument form of `html->string` sends the encoded HTML string to the textual output port *op*.

Input that does not match the specification causes a `#(bad-arg html->string` *x*`)` exception to be raised.

(html->bytevector *x*)	**procedure**

returns: a bytevector

The `html->bytevector` procedure calls `html->string` on *x* using a bytevector output port transcoded using `(make-utf8-transcoder)` and returns the resulting bytevector.

14.4.4 JavaScript Object Notation

This implementation translates JavaScript types into the following Scheme types:

JavaScript	Scheme
`true`	`#t`
`false`	`#f`
`null`	`#\nul`
string	*string*
number	*number*
array	*list*
object	hashtable mapping symbols to values

This implementation does not range check values to ensure that a JavaScript implementation can interpret the data.

`(json:extend-object` *ht* `[`*key* *value*`] ...)` **syntax**

The `json:extend-object` construct adds the *key* / *value* pairs to the hashtable *ht* using `hashtable-set!`. Each *key* is a literal identifier or an unquoted expression `,`*e* that evaluates to a symbol. The resulting expression returns *ht*.

`(json:make-object [`*key* *value*`] ...)` **syntax**

The `json:make-object` construct expands into a call to `json:extend-object` with a new hashtable.

`(json:object?` *x* **procedure**

returns: a boolean

The `json:object?` procedure determines whether or not the datum *x* is an object created by `json:make-object`.

`(json:cells` *ht*`)` **procedure**

returns: a vector

The `json:cells` procedure returns a vector containing the cells of the underlying hashtable.

`(json:size` *ht*`)` **procedure**

returns: an integer

The `json:size` procedure returns the number of cells in the underlying hashtable.

`(json:delete!` *ht* *path*`)` **procedure**

returns: unspecified

The `json:delete!` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:delete!` is equivalent to `hashtable-delete!`. Otherwise, `json:delete!` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:delete!` reaches the final key in *path*, it calls `hashtable-delete!` to remove the association for that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, `json:delete!` has no effect.

(`json:ref` *ht path default*) **procedure**
returns: the value found by traversing *path* in *ht*, *default* if none

The `json:ref` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:ref` is equivalent to `hashtable-ref`. Otherwise, `json:ref` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:ref` reaches the final key in *path*, it calls `hashtable-ref` to retrieve the value of that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, or if the final hashtable does not contain the final key, `json:ref` returns *default*.

(`json:set!` *ht path value*) **procedure**
returns: unspecified

The `json:set!` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:set!` is equivalent to `hashtable-set!`. Otherwise, `json:set!` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:set!` reaches the final key in *path*, it calls `hashtable-set!` to set that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, `json:set!` installs an empty hashtable at that key before proceding. If *path* is malformed at some point, `json:set!` may still mutate hashtables along the valid portion of the path before reporting an error.

(`json:update!` *ht path procedure default*) **procedure**
returns: unspecified

The `json:update!` procedure expects *path* to be a symbol or a non-empty list of symbols. If *path* is a symbol, then `json:update!` is equivalent to `hashtable-update!`. Otherwise, `json:update!` follows *path* as it descends into the nested hashtable *ht*, treating each element as a key into the hashtable reached by traversing the preceding elements. When `json:update!` reaches the final key in *path*, it calls `hashtable-update!` to update that key in the hashtable reached at that point. If any key along the way does not map to a hashtable, `json:update!` installs an empty hashtable at that key before proceding. If *path* is malformed at some point, `json:update!` may still mutate hashtables along the valid portion of the path before reporting an error.

(`json:read` *ip* [*custom-inflate*]) **procedure**
returns: a Scheme object or the eof object

The `json:read` procedure reads characters from the textual input port *ip* and returns an appropriate Scheme object. When `json:read` encounters a JSON object, it builds the corresponding hashtable and calls *custom-inflate* to perform application-specific conversion. By default, *custom-inflate* is the identity function.

The following exceptions may be raised:

- `invalid-surrogate-pair`

- `unexpected-eof`
- `#(unexpected-input` *data* *input-position*`)`

`(json:write` *op* *x* [*indent*] [*custom-write*]`)` **procedure**
returns: unspecified

The `json:write` procedure writes the object *x* to the textual output port *op* in JSON format. JSON objects are sorted by key using `string<?` on the string values of the symbols to provide stable output. Scheme fixnums, bignums, and finite flonums may be used as numbers.

When *indent* is a non-negative fixnum, the output is more readable by a human. List items and key/value pairs are indented on individual lines by the specified number of spaces. When *indent* is 0, a newline is added to the end of the output. The default indent of `#f` produces compact output.

The optional *custom-write* procedure may intervene to handle lists and hashtables differently or to handle objects that have no direct JSON counterpart. If *custom-write* does not handle a given object, it should return false to let `json:write` proceed normally. The *custom-write* procedure is called with four arguments: the textual output port *op*, the Scheme object *x*, the current *indent* level, and a writer procedure *wr* that should be used to write the values of arbitrary Scheme objects. The *wr* procedure is equivalent to `(lambda (`*op* *x* *indent*`) (json:write` *op* *x* *indent* *custom-write*`))`.

If an object cannot be formatted, `#(invalid-datum` *x*`)` is raised.

`(json:write-object` *op* *indent* *wr* `[`*key* *value*`] ...)` **syntax**
returns: `#t`

Given a textual output port *op*, an *indent* level, and a writer procedure *wr*, the `json:write-object` construct writes a JSON object with the given *key* / *value* pairs to *op*, sorted by key using `string<?` on the string values of the symbols. Each *key* must be a distinct symbol. The *wr* procedure takes *op*, an object *x*, and an *indent* level just like the *wr* procedure that is passed to `json:write`'s *custom-write* procedure.

The following are equivalent, provided the keys are symbols.

`(begin (json:write` *op* `(json:make-object [`*key* *value*`] ...)` *indent*`) #t)`
`(json:write-object` *op* *indent* `json:write [`*key* *value*`] ...)`

The latter trades code size and compile time for run-time efficiency. At compile time, `json:write-object` sorts the keys and preformats the strings that will separate values.

`(json:object->bytevector` *x* [*indent*] [*custom-write*]`)` **procedure**
returns: a bytevector

The `json:object->bytevector` procedure calls `json:write` on *x* with the optional *indent* and *custom-write*, if any, using a bytevector output port transcoded using `(make-utf8-transcoder)` and returns the resulting bytevector.

`(json:bytevector->object` *x* [*custom-inflate*]`)` **procedure**
returns: a Scheme object

The `json:bytevector->object` procedure creates a bytevector input port on x, calls `json:read` with the optional *custom-inflate*, if any, and returns the resulting Scheme object after making sure the rest of the bytevector is only whitespace.

(`json:object->string` x [*indent*] [*custom-write*]) **procedure**
returns: a JSON formatted string

The `json:object->string` procedure creates a string output port, calls `json:write` on x with the optional *indent* and *custom-write*, if any, and returns the resulting string.

(`json:string->object` x [*custom-inflate*]) **procedure**
returns: a Scheme object

The `json:string->object` procedure creates a string input port on x, calls `json:read` with the optional *custom-inflate*, if any, and returns the resulting Scheme object after making sure the rest of the string is only whitespace.

(`json:write-structural-char` x *indent* *op*) **procedure**
returns: the new indent level

The `json:write-structural-char` procedure writes the character x at an appropriate *indent* level to the textual output port *op*. The character should be one of the following JSON structural characters: `[ ] { } : ,`

This procedure is intended for use within custom writers passed in to `json:write` and, for performance, it does not check its input arguments.

14.5 Published Events

`<http-request>` **event**

timestamp: timestamp from `erlang:now`
pid: handler process
host: the IP address of the client
method: `<request>` method
path: `<request>` path
header: a JSON object
params: a JSON object

Chapter 15

Command Line Interface

15.1 Introduction

The command-line interface (`cli`) provides parsing of command-line arguments as well as consistent usage of common options and display of help.

15.2 Theory of Operation

Many programs parse command-line arguments and perform actions based on them. The `cli` library helps to make programs that process arguments and display help simple and consistent. Command-line arguments are parsed left to right in a single pass. Command-line interface specifications, or `cli-specs`, are used for parsing and error checking a command line, displaying one-line usage, and displaying a full help summary.

Arguments may be preceded by a single dash (`-`), a double dash (`--`), or no dash at all. A single dash precedes short, single character arguments. The API does not allow numbers as they could be mistaken as a negative numerical value supplied to another argument. A double dash precedes longer, more descriptive arguments, `--repl` for example. Positional arguments are not preceded by any dashes. As arguments with dashes are consumed, the remaining arguments are matched against the positional specifications in order.

Argument specifications include a type such as: `bool`, `count`, `string`, and `list`. A set of `bool` and `count` arguments can be specified together (`-abc` is equivalent to `-a -b -c`). Arguments of type `list` collect values in left to right order.

The API does not directly support sub-commands and alternate usage help text. These can be implemented using the primitives provided. The implementations of `swish-build` and `swish-test` provide examples of advanced command-line handling.

In the following REPL transcript, we define `example-cli` using `cli-specs`. We then set the `command-line-arguments` parameter as they would be for an application. Calling `parse-command-line-arguments` returns a procedure, `opt`, which we can use to access the parsed command-line values. Finally, we use `display-help` to display the automatically generated help.

```
> (define example-cli
    (cli-specs
     default-help
     [verbose -v count "indicates verbosity level"]
     [output -o (string "<output>") "print output to an <output> file"]
     [repl --repl bool "start a repl"]
     [files (list "<file>" ...) "a list of input files"]))
> (command-line-arguments '("-vvv" "-o" "file.out" "file.in"))
> (define opt (parse-command-line-arguments example-cli))
> (opt 'verbose)
3
> (opt 'output)
"file.out"
> (opt 'files)
("file.in")
> (display-help "sample" example-cli)
Usage: sample [-hv] [-o <output>] [--repl] <file> ...

  -h, --help        display this help and exit
  -v                indicates verbosity level
  -o <output>       print output to an <output> file
  --repl            start a repl
  <file> ...        a list of input files
```

Putting the parts together into `sample.ss`, we have a working example albeit incomplete.

```
#!/usr/bin/env swish

(define example-cli
  (cli-specs
   default-help
   [verbose -v count "indicates verbosity level"]
   [output -o (string "<output>") "print output to an <output> file"]
   [repl --repl bool "start a repl"]
   [files (list "<file>" ...) "a list of input files"]))

(let ([opt (parse-command-line-arguments example-cli)])
  (when (opt 'help)
    (display-help "sample" example-cli)
    (exit 0))
  (let ([verbosity (or (opt 'verbose) 0)])
    (when (> verbosity 0)
      (printf "showing verbosity level: ~a~n" verbosity)))
  (when (opt 'repl)
    (new-cafe)))
```

15.3 Programming Interface

```
(cli-specs                                                    syntax
  [default-help]
  (name [short] [long] type help
    [(conflicts conflicts)]
    [(requires requires)]
    [(usage [visibility] [how])])
  ...)
```

expands to:

```
a list of <arg-spec> tuples
```

The `cli-specs` macro simplifies the creation of the `<arg-spec>` tuples. The `<arg-spec> name` field uniquely identifies a specification, and is used to retrieve parsed argument values and check constraints.

name: a symbol to identify the argument
short: a symbol of the form `-x`, where x is a single character, see below
long: a symbol of the form `--x`, where x is a string
type: see Figure 15.1
help: a string or list of strings that describes the argument
conflicts: a list of `<arg-spec>` names
requires: a list of `<arg-spec>` names

To specify `-i` or `-I` for *short*, use |`-i`| and |`-I`| respectively to prevent Chez Scheme from reading them as the complex number $0 - 1i$.

Type	Result
`bool`	`#t`
`count`	a positive integer
`(string` *x*`)`	a string
`(list` *x*`)`	a list of one item
`(list` *x* `...)`	a list of one or more items up to the next argument
`(list .` *x*`)`	a list of the rest of the arguments

For each type where x is specified, x is a string that is used in the help display.

Figure 15.1: Command-line argument types

The `list` types can support multiple x arguments, for instance `(list "i1" "i2")` would specify a list of two arguments.

visibility → `show`
| `hide`
| `fit`

When printing the help usage line, a *visiblility* of `show` means the argument must be displayed. `hide` forces the argument to be hidden. `fit` displays the argument if it fits on the line.

how → `short`
  | `long`
  | `opt`
  | `req`

The *how* expands into input of the `format-spec` procedure according to Figure 15.2.

Keyword	Expands into:
`short`	`(opt (and short args))`
`long`	`(opt (and long args))`
`opt`	`(opt (and (or short long) args))`
`req`	`(req (and (or short long) args))`

Figure 15.2: `cli-specs` *how* field

For options with *short* or *long* specified, `fit` and `opt` are the defaults. For other options, `show` and `req` are the defaults.

conflicts is a list of specification names that prevent this argument from processing correctly. When multiple command-line arguments are specified that are in conflict, an exception is raised.

requires is a list of other specification names that are necessary for this argument to be processed correctly. Unless all the required command-line arguments are specified, an exception is raised.

The `conflicts`, `requires`, and `usage` clauses may be specified in any order.

`(display-help` *exe-name specs* [*args*] [*op*]`)` **procedure**
returns: unspecified

The `display-help` procedure is equivalent to calling `display-usage` with a prefix of `"Usage:"` followed by `display-options`.

`(display-options` *specs* [*args*] [*op*]`)` **procedure**
returns: unspecified

For each specification in *specs*, the `display-options` procedure renders two columns of output to *op*, which defaults to the current output port. The first column renders the short and long form of the argument with its additional inputs. The second column renders the `<arg-spec> help` field and will automatically wrap if the `help-wrap-width` is exceeded.

If an *args* hash table is specified, the specified value appended to the second column. This is useful for displaying default or current values.

`(display-usage` *prefix exe-name specs* [*width*] [*op*]`)` **procedure**
returns: unspecified

The `display-usage` procedure displays the first line of help output to *op*, which defaults to the current output port. It starts with *prefix* and *exe-name* then attempts to fit *specs* onto the line using `format-spec`. When the line will exceed *width* characters, some arguments may collapse to `[options]`.

A *width* of #f defaults the line width to help-wrap-width.

(format-spec *spec* [*how*]) — **procedure**

returns: a string

The format-spec procedure is responsible rendering *spec* as a string as specified by *how*. format-spec can display dashes in front of arguments, ellipses on list types, and brackets around optional arguments. A *how* of #f defaults to the <arg-spec> usage field.

<table>
<tr><th>how</th><th>Return value:</th></tr>
<tr><td>short</td><td>"-x" if spec short is x, else #f</td></tr>
<tr><td>long</td><td>"--x" if spec long is x, else #f</td></tr>
<tr><td>args</td><td>The spec type is evaluated as follows:
<table>
<tr><th>type</th><th>Return value:</th></tr>
<tr><td>bool</td><td>#f</td></tr>
<tr><td>count</td><td>#f</td></tr>
<tr><td>(string x)</td><td>"x"</td></tr>
<tr><td>(list x)</td><td>"x"</td></tr>
<tr><td>(list x ...)</td><td>"x ..."</td></tr>
<tr><td>(list . x)</td><td>"x ..."</td></tr>
</table></td></tr>
<tr><td>(or how ...)</td><td>Recur and use the first non-#f</td></tr>
<tr><td>(and how ...)</td><td>Recur and concatenate all non-#f values</td></tr>
<tr><td>(opt how)</td><td>Recur and surround the result with square brackets [] if non-#f</td></tr>
<tr><td>(req how)</td><td>Recur and use the result</td></tr>
</table>

help-wrap-width — **parameter**

value: a positive fixnum

The help-wrap-width parameter specifies the default width for display-usage and display-options.

(parse-command-line-arguments *specs* [*ls*] [*fail*]) — **procedure**

returns: a procedure

The parse-command-line-arguments procedure processes the elements of *ls* from left to right in a single pass. As it scans each *x* in *ls*, the parser must find a suitable *s* within *specs*. If a suitable *s* cannot be found, the parser reports an error by calling *fail*. Based on the type of *s*, the parser may consume additional elements following *x*. The type of *s* determines what data the parser records for that argument. When *s* is satisfied, the parser continues scanning the remaining elements of *ls*.

The parser returns a procedure *p* that accepts zero or one argument. When called with no arguments, *p* returns a hash table that maps the name of each *s* found while processing *ls* to the data recorded for that argument. When called with the name of an element *s* in *specs*, *p* returns the data, if any, recorded for that name in the hash table or else #f. If a particular *s* was not found while processing *ls*, the internal hash table has no entry for the name of *s* and *p* returns #f when given that name. If called with a name that is not in *specs*, *p* raises an exception.

The following table summarizes the parser's behavior.

`<arg-spec> type`	extra arguments consumed / recorded	return value of `(`p $name$`)`
`bool`	none	`#t`
`count`	none	an exact positive integer
`(string` x`)`	one	a string
`(list` x_0 `...` x_n `...)`	n or more, up to the next option	a list of strings
`(list` x_0 `...` x_n `.` *rest*`)`	at least n and all remaining	a list of strings

By default *ls* is the value of `(command-line-arguments)` and *fail* is a procedure that applies `errorf` to its arguments. Providing a *fail* procedure allows a developer to accumulate parsing errors without necessarily generating exceptions.

`<arg-spec>` **tuple**

name: a symbol to use as the key of the output hash table
type: see Figure 15.1
short: `#f` | a character
long: `#f` | a string
help: a string or list of strings describing argument
conflicts: a list of `<arg-spec>` names
requires: a list of `<arg-spec>` names
usage: a list containing one *visibility* symbol and a `format-spec` *how* expression

Bibliography

[1] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[2] Joe Armstrong. *Programming Erlang—Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[3] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014. `http://www.ietf.org/rfc/rfc7159.txt`.

[4] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[5] C99 — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/w/index.php?title=C99&oldid=813613099`.

[6] R. Kent Dybvig. *Chez Scheme Version 9 User's Guide*. Cadence Research Systems, 2017. `https://cisco.github.io/ChezScheme/csug9.5/csug.html`.

[7] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation*, 5:83–110, 1992.

[8] Erlang. `http://www.erlang.org/`.

[9] Erlang dict module. `http://www.erlang.org/doc/man/dict.html`.

[10] Erlang gen_server module. `http://www.erlang.org/doc/man/gen_server.html`.

[11] Erlang queue module. `http://www.erlang.org/doc/man/queue.html`.

[12] Erlang supervisor module. `http://www.erlang.org/doc/man/supervisor.html`.

[13] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, December 2011. `https://rfc-editor.org/rfc/rfc6455.txt`.

[14] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014. `http://www.ietf.org/rfc/rfc7230.txt`.

[15] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, second edition, 2002.

[16] William G. Griswold and Gregg M. Townsend. The design and implementation of dynamic hashing for sets and tables in icon. *Software Practice and Experience*, 23(4):351–367, April 1993.

[17] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[18] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the Third IJCAI*, pages 235–245, Stanford, MA, 1973.

[19] Ian Hickson. HTML 5, October 2014. `http://www.w3.org/TR/2014/REC-html5-20141028/`.

[20] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[21] libuv. `http://libuv.org/`.

[22] Michael Owens. *The Definitive Guide to SQLite*. Apress, 2006.

[23] E. Resnick. Internet Message Format, April 2001. `http://www.ietf.org/rfc/rfc2822.txt`.

[24] The Base16, Base32, and Base64 data encodings, 2006. `https://tools.ietf.org/html/rfc4648`.

[25] Dorai Sitaram. pregexp: Portable Regular Expressions for Scheme and Common Lisp, 2005. `https://ds26gte.github.io/pregexp/`.

[26] SQLite. `http://www.sqlite.org/`.

[27] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

List of Figures

Index