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 [13].

Swish uses message-passing concurrency and fault isolation to provide fault-tolerant software [1, 16]. 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, 18] 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 2), 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.

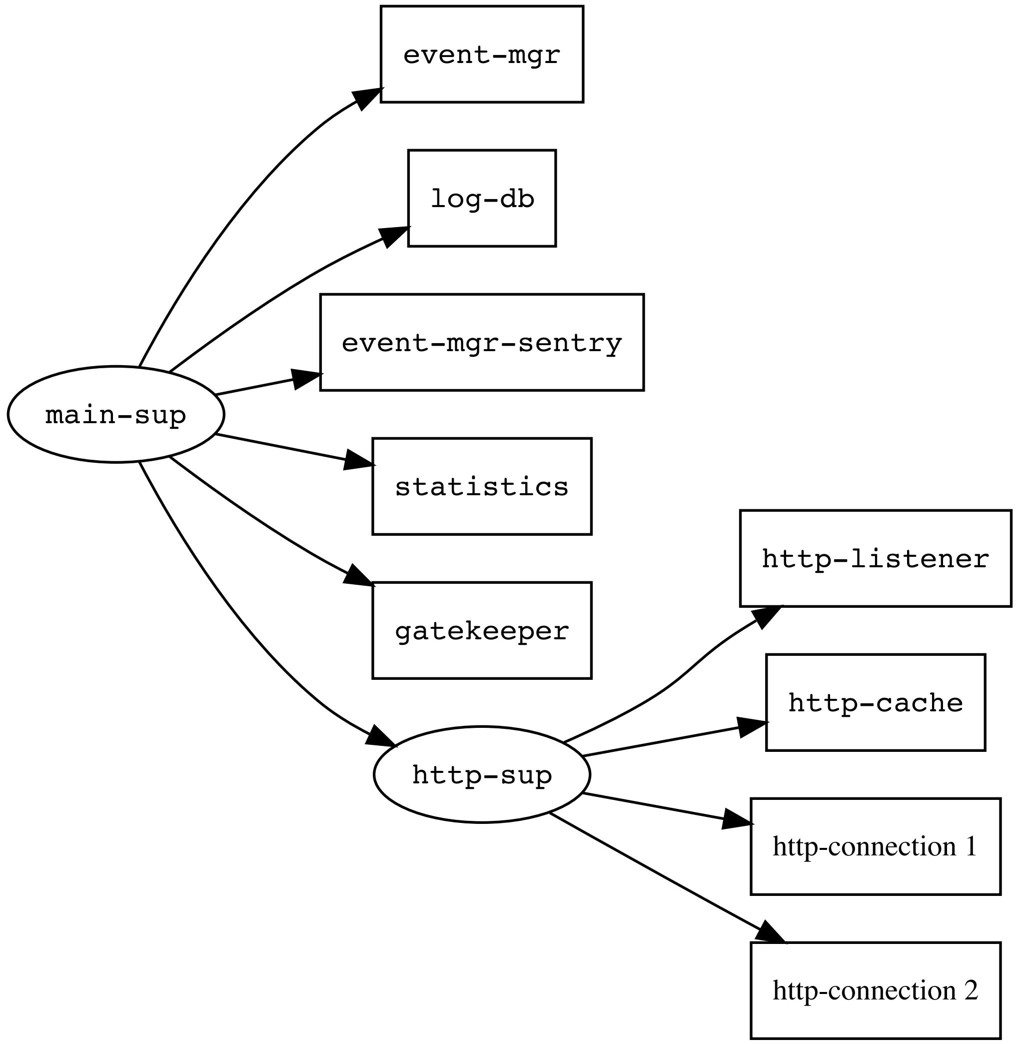

Figure 1.1: Supervision Tree

1.2 Supervision Tree

By default, Swish uses the supervision tree illustrated in Figure 1.1. 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 5). The log-db worker is a database gen-server (see Chapter 10) 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 11) that periodically posts a `<statistics>` event. The gatekeeper worker is the gen-server described in Chapter 6.

The `http-sup` supervisor is configured one-for-one with up to 10 restarts every 10 seconds. The `http-listener`, `http-cache`, and http-connection processes are described in Chapter 12. When the `http-port-number` parameter is set to its default value of `#f`, the `http-sup` supervisor is not used.

Chapter 2

Operating System Interface

2.1 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 [19] and database support via SQLite [20].

2.2 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.

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 §3.3).

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 2.3 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*`)`.

2.3 Programming Interface

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

2.3.1 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 on the main thread only.

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 on the main thread only.

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 on the main thread only.

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 on the main thread only.

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.

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.

2.3.2 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*.

2.3.3 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.

2.3.4 Process Functions

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

The `osi_exit` function 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_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.

2.3.5 File System Functions

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.

uptr `osi_get_stdin`(void); **function**

The `osi_get_stdin` function returns a handle to a static port that reads from the standard input of the current process. This port cannot be closed.

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`.

2.3.6 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`.

2.3.7 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.

Chapter 3

Erlang Embedding

3.1 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 2) 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 3.2 introduces the main data structures, Section 3.3 describes how the concurrency model works, and Section 3.4 gives the programming interface.

3.2 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`, 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, and bit 2 set when the process is blocked for I/O
- `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.

3.3 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 6. 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 [18] 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 2) 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*, the message #(DOWN *m* *p* *r*) is sent to each of its monitor *m*'s origin processes. The message #(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 register objects that wrap operating system interface handles with a guardian and send messages

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

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 only by a writer process. Keeping the input and output sides separate prevents concurrent use.

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

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* and associated finalizer are used to identify and close inaccessible osi-ports using `osi_close_port`. A *path-watcher guardian* and associated finalizer are used to identify and close inaccessible path watchers. A *listener guardian* and associated finalizer are 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.

3.4 Programming Interface

3.4.1 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.

3.4.2 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* → *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.

3.4.3 Process Termination, Links, and Monitors

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

expands to:

```
(call/1cc
 (lambda (return)
   (with-exception-handler
    (lambda (reason) (return `#(EXIT ,reason)))
    (lambda () e1 e2 ...))))
```

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.

`(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, message `#(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, message `#(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, the message `#(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.

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 $p_1 \ldots\ p_n$
`,_`	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* $\mid$ `[`*field pattern*`]`} ...`)`	an instance of the tuple *type*, each *field* of which is bound to fresh variable *field* or matches the corresponding *pattern*

Figure 3.1: Pattern Grammar

3.4.4 Messages and Pattern Matching

The pattern matching syntax of Figure 3.1 provides a concise and expressive way to match data structures and bind variables to parts. The `receive`, `match`, and `match-let*` macros use this pattern language. The implementation uses two structurally recursive passes over the pattern. The first pass checks the pattern syntax including tuple types and field names and accumulates the list of pattern variables. This list enables it to check for duplicate pattern variables. The second pass 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 3.1 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 3.1 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 3.1 for the pattern grammar.

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

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.

3.4.5 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.

`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.

3.4.6 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.

`(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. 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*`)`
`(dbg` *base proc*`)`
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."

`(dbg` *base proc*`)` folds over the processes that exited with a continuation condition and calls *proc* with the process id, process exception state, and the accumulator value (initially *base*). It returns the final accumulator value.

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

The `dump-stack` procedure prints information about the stack up to *max-depth* 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. The default max-depth is 10.

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

`(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 using a weak eq-hashtable mapping processes to values. 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`: `custom-port-buffer-size`, `exit-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`.

`(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.

`(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.

3.4.7 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*.

3.4.8 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 do not track or report 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 binary input port and *op* is the binary output port.

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-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-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-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
- `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-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) #o777 '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) #o777 'output)`.

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

The `open-file-to-write` procedure calls
`(open-file` *name* `(+ O_WRONLY O_CREAT O_EXCL) #o777 '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-count)` **procedure**

returns: the number of open osi-ports

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

`(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-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-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.

`(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 do not track or report 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.

`(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.

3.4.9 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 a queue, `#f` otherwise

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

3.4.10 Hash Tables

The implementation of functional hash tables is based on the Erlang dict module [9, 14].

(`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:

$$F_0 = \mathit{init}$$
$$F_i = (f\ \ \mathit{key}_i\ \ \mathit{val}_i\ \ F_{i-1}) \text{ for } 1 \le i \le n$$

(`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*

3.4.11 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.

3.4.12 String Utilities

The string utilities below are found in the `(swish string-utils)` library. For regular expression support, see the `(swish pregexp)` library described in [22].

`(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 [21].

`(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.

`(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.

Chapter 4

Generic Server

> *The generic server provides an "empty" server, that is, a framework from which instances of servers can be built.* —Joe Armstrong [1]

4.1 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.

4.2 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`.

Unless the termination reason is either `normal` or `shutdown`, 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.

4.3 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: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.

`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.

4.4 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
last-message: the last message received by the server
state: the last state passed into `terminate`
reason: the reason for termination

This event is fired after a successful call to `terminate` unless the reason for termination is `normal` or `shutdown`. 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`

4.5 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 `#(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. Unless the termination reason is either `normal` or `shutdown`, 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 5

Event Manager

5.1 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.

5.2 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 5.1.

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. 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.

state

`<event-mgr-state>` **tuple**

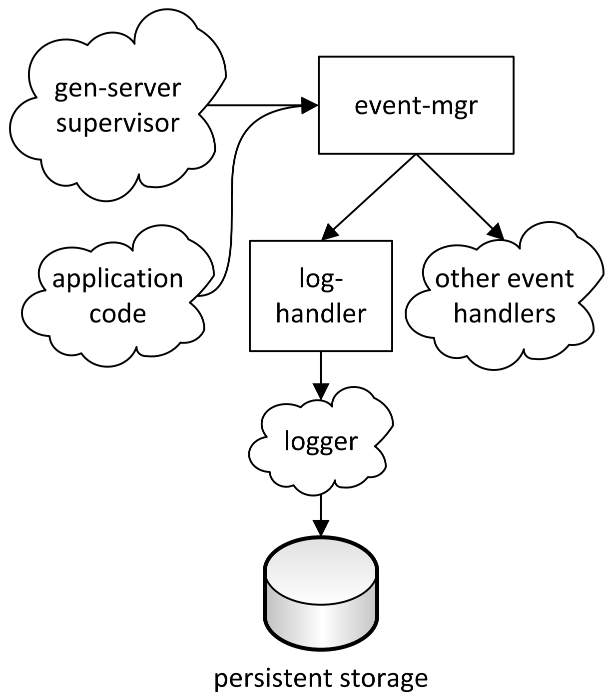

Figure 5.1: Event flow

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 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 the following messages:

- `#(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 using `console-event-handler`, and remove the log handler from the state.

- Otherwise, buffer the event.

5.3 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.

Chapter 6

Gatekeeper

6.1 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 3.3). The `$cp0` procedure uses resource `$cp0`, the `$np-compile` procedure uses resource `$np-compile`, and so forth.

6.2 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* $\in$ *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* $\in$ *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 the following message:

- `#(DOWN` *monitor* `_ _)`: Find *mutex* $\in$ *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* $\in$ *state* where `#(`*owner* `_)` $\in$ *waiting.waiters*. If no such *waiting* exists, return `#f`. Otherwise, return `(deadlock?` *process waiting state*`)`.

6.3 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 7

Supervisor

7.1 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 [15] is a good reference for understanding the mindset of creating fault tolerant systems.

7.2 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 the following message:

- `#(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.

7.3 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.

7.4 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)`.

7.5 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
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

This event is fired after the supervisor terminates a child process, and after the supervisor detects a failure in a child.

7.6 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: `ok`

The `watcher:shutdown-children` procedure terminates and deletes each `watch-only` child in *watcher*.

Chapter 8

Application

8.1 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.

8.2 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 the following message:

- `#(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 port, ignoring any exceptions, and then calls `(osi_exit` *exit-code*`)`.

8.3 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-process 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 9

Database Interface

9.1 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 2).

Other SQLite resources are available online [23] or in The Definitive Guide to SQLite [20].

9.2 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. 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.

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 (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, `PrepareStatement` 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 handle) (immutable database)))`
  The `statement` record is not registered with a guardian. The statement is finalized using `FinalizeStatement` when it is removed from the cache. `CloseDatabase` 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 and mode symbol and attempts to open that database, setting `journal_mode` to "wal" if *mode* is `create`. The handle returned from `OpenDatabase` is wrapped in a *database* record that is registered with a guardian. The garbage collector is hooked 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 the following messages:

- `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*.

9.3 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.

9.4 Programming Interface

(`db:start&link` *name filename mode*) **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 `OpenDatabase`:

- `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)`.

The SQLite constants can be found in `sqlite3.h` or online [23].

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) (raise 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 `BindStatement`. The statement is then executed using `StepStatement`. The results are accumulated as a list, and the statement is reset using `ResetStatement` 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 `PrepareStatement` 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 `BindStatement`. A thunk is returned which, when called, executes the statement using `StepStatement`. 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 `GetStatementColumns`.

`(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-count)` **procedure**

returns: the number of open databases

The `database-count` procedure returns the number of open 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.

(statement-count) **procedure**

returns: the number of unfinalized statements

The `statement-count` procedure returns the number of unfinalized 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.

(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` [23]. The constants `SQLITE_OPEN_CREATE`, `SQLITE_OPEN_READONLY`, and `SQLITE_OPEN_READWRITE` are exported from the `(swish db)` library.

`(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*.

`(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 10

Log Database

10.1 Introduction

The log database is a single gen-server named `log-db` that uses the database interface (see Chapter 9) to log system events (see Chapter 5).

10.2 Theory of Operation

10.2.1 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.

10.2.2 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.

10.3 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`
process	integer key for *process*, unique in this database instance
condition	a string containing `#(error` *reason*`)` where the *reason* is obtained from `display-condition`
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`.

10.4 Published Events

`<system-attributes>`	**event**

timestamp: timestamp from `erlang:now`
date: date from `current-date`
software-version: software version string
computer-name: computer name from `osi_get_hostname`

The `<system-attributes>` event is sent exactly once, when `log-db:setup` is called.

Chapter 11

System Statistics

11.1 Introduction

The system uses a single gen-server named `statistics` to periodically query statistics about the system, such as memory usage.

11.2 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 10), which adds the data to the statistics table in the log database.

11.3 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 aliased to `app:resume` and called from the operating system interface.

`(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 aliased to `app:suspend` and called from the operating system interface.

11.4 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`
databases: number of open SQLite databases
statements: number of allocated SQLite statements
listeners: number of open TCP/IP listeners
ports: number of open osi-ports
watchers: number of open path watchers
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

This event is sent every five minutes while the `statistics` gen-server is running.

Chapter 12

HTTP Interface

12.1 Introduction

The HTTP interface provides a basic implementation of the Hypertext Transfer Protocol [13]. The programming interface includes procedures for the HyperText Markup Language (HTML) version 5 [17] and JavaScript Object Notation (JSON) [3].

12.2 Theory of Operation

The HTTP interface provides a supervisor, `http-sup`, to manage the `http-listener` gen-server, the `http-cache` gen-server, and new connection processes. This structure is illustrated in Figure 12.1.

The `http-listener` is a gen-server that creates a TCP listener using `listen-tcp` and accepts new connections using `accept-tcp`. For each connection, the http-listener uses its supervisor to spawn and link a handler process.

Each handler reads from its input port until a CR LF occurs. Well-formed input is converted to a `<request>` tuple, and the HTTP request header and any content parameters are read.

When `Content-Length` appears in the header, the content bytes are read. If the `Content-Type` is `multipart/form-data` or `application/x-www-form-urlencoded`, the content is converted to

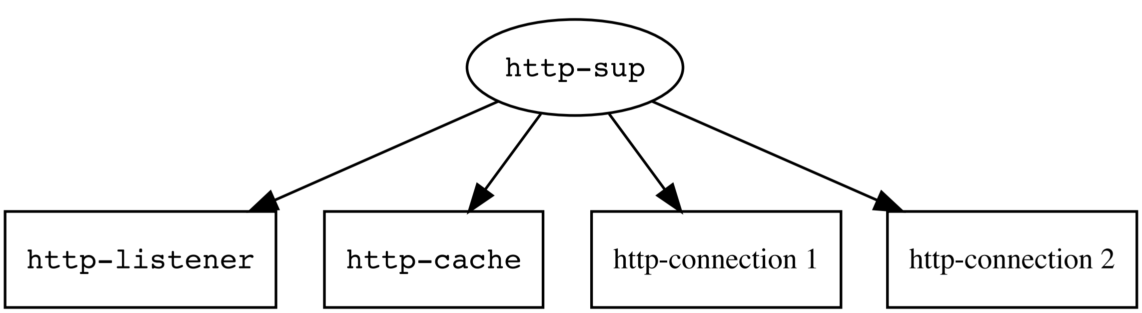

Figure 12.1: HTTP tree

an association list. Otherwise, parameter `"unhandled-content"` is included with the value of the raw bytevector of data. Each uploaded file is stored in `(tmp-dir)`, and the association list value is `#(<file>` *filename*`)`.

`http:file-handler` is then called combining the `<request>` query parameters and content parameters. `http:file-handler` logs the specific request, validates that the requested path does not include "..", retrieves a page handler from the `http-cache`, and invokes it. A *page handler* is a procedure which responds to a particular HTTP request. The output port is flushed after the page handler returns.

After a request is processed, all uploaded files are deleted, and the current process and connection can be reused. The system reads another request from the input port.

The `http-cache` is a gen-server that stores page handlers and provides a mapping from file extension to content type. It creates a directory watcher using `watch-directory` to invalidate the cache when anything in the `(web-dir)` tree changes.

The `http-cache` considers a path that ends in ".ss" a dynamic page loaded from `(web-dir)`. Other paths are considered static and are sent directly over the connection using `http:respond-file`.

12.3 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 `(web-dir)` using ".." could allow access to system files. `http:file-handler` explicitly checks for relative paths.

The HTTP interface limits incoming data to protect against large memory allocation which may crash the system. URL requests are limited to 4,096 bytes. Headers are limited to 1,048,576 bytes. Posted content is limited to 4,194,304 bytes, not including uploaded files.

The HTTP interface does not limit incoming file uploads. If the disk runs out of space, the handler process will exit with an I/O error.

12.4 Dynamic Pages

A dynamic page is a sequence of definitions followed by a sequence of expressions stored in ".ss" files in `(web-dir)`. The definitions and expressions are placed in a `lambda` expression that is evaluated by the `interpret` system procedure. The page is responsible for sending the HTTP response. The output port is flushed after the page handler returns.

12.5 Dynamic Page Constructs

The evaluated `lambda` expression exposes the following variables to a dynamic page:

ip: binary input port
op: binary output port
request: a `<request>` tuple
header: an association list
params: an association list

`(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 `(web-dir)` 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*.

12.6 Programming Interface

`<request>` **tuple**

method: a symbol
path: a decoded string
query: a decoded association list

`http-port-number` **parameter**

value: `#f` or a fixnum $0 \le port \le 65535$

The `http-port-number` parameter specifies whether or not `http-sup:start&link` should start the HTTP server and, if started, on what port that server should listen for connections.

`(http-sup:start&link)` **procedure**

returns: `#(ok` *pid*`)` | `#(error` *error*`)`

If `(http-port-number)` is not `#f`, the `http-sup:start&link` procedure creates a supervisor named `http-sup` using `supervisor:start&link` configured one-for-one with up to 10 restarts every 10 seconds. The supervisor starts the `http-cache` and `http-listener` gen-servers.

`(http:get-port-number)` **procedure**

returns: see below

If the `(http-port-number)` is 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 *header*. Header comparisons are case-insensitive. If *name* is not a string, 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 assocated with *name* in *header* or exits with reason `#(invalid-header` *name header*`)`. Header comparisons are case-insensitive. If *name* is not a string, exception `#(bad-arg http:get-header` *name*`)` is raised.

(`http:find-param` *name params*) **procedure**
returns: a string | `#f`

The `http:find-param` procedure returns the value associated with *name* in *params*. Parameter comparisons are case-sensitive. If *name* is not a string, 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 *params* or exits with reason `#(invalid-param` *name params*`)`. Parameter comparisons are case-sensitive. If *name* is not a string, exception `#(bad-arg http:get-param` *name*`)` is raised.

(`http:read-header` *ip limit*) **procedure**
returns: an association list

The `http:read-header` procedure reads from the binary input port *ip* until a blank line is read.

An association list is created by making a string from the characters before the first colon as the key. Non-linear white space is skipped, and the remaining characters are converted to a string value.

Reading beyond *limit* will result in exiting with reason `input-limit-exceeded`.

Failure to find a colon on any given line will result in exiting with reason `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 *op* and returns the number if well formed and `#f` otherwise. Reading beyond *limit* will result in exiting with reason `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 [13] the status line includes a human readable reason phrase. The grammar shows that it can in fact be 0 characters long; therefore, the reason phrase is not included in this implementation.

`(http:write-header` *op header*`)` **procedure**
returns: unspecified

The `http:write-header` procedure writes the HTTP *header*, and trailing CR LF to the binary output port *op*.

header is an association list. If *header*'s keys are not strings, exception `#(bad-arg http:write-header` *header*`)` is raised.

`(http:respond` *op status header content*`)` **procedure**
returns: unspecified

The `http:respond` procedure writes the HTTP *status* and *header* to binary output port *op* using `http:write-status` and `http:write-header`, adding `Content-Length` to the *header*. When `Cache-Control` is not present in *header*, it is added with value `no-cache`. The *content* is then written, and the output port is flushed.

content is a bytevector.

`(http:respond-file` *op status header filename*`)` **procedure**
returns: unspecified

The `http:respond-file` procedure writes the HTTP *status* and *header* to binary output port *op* using `http:write-status` and `http:write-header`, adding `Content-Length` to *header*. 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 and the extension of *filename* matches (case insensitively) an extension in the `mime-types` file of `(web-dir)`. Each line of `mime-types` has the form `("`*extension*`" . "`*Content-Type*`")`. 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.

`(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.

`(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* ...)	$H(pattern)\ldots$
(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.

12.6.1 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 case-sensitive strings 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!`. 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: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<?` 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<?`. Each *key* must be a distinct string literal. 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 string literals.

```
(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: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.

12.7 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: an association list
params: an association list

Chapter 13

Command Line Interface

13.1 Introduction

The command-line interface (`cli`) provides parsing of command-line arguments as well as consistent usage of common options and display of help.

13.2 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)))
```

13.3 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 string 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 13.1
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 13.1: 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 13.2.

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 13.2: `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><code>short</code></td><td><code>"-x"</code> if spec short is <code>x</code>, else <code>#f</code></td></tr>
<tr><td><code>long</code></td><td><code>"--x"</code> if spec long is <code>x</code>, else <code>#f</code></td></tr>
<tr><td><code>args</code></td><td>The spec type is evaluated as follows:
<table>
<tr><th>type</th><th>Return value:</th></tr>
<tr><td><code>bool</code></td><td><code>#f</code></td></tr>
<tr><td><code>count</code></td><td><code>#f</code></td></tr>
<tr><td><code>(string</code> x<code>)</code></td><td><code>"x"</code></td></tr>
<tr><td><code>(list</code> x<code>)</code></td><td><code>"x"</code></td></tr>
<tr><td><code>(list</code> x <code>...)</code></td><td><code>"x ..."</code></td></tr>
<tr><td><code>(list .</code> x<code>)</code></td><td><code>"x ..."</code></td></tr>
</table></td></tr>
<tr><td><code>(or</code> how ...<code>)</code></td><td>Recur and use the first non-<code>#f</code></td></tr>
<tr><td><code>(and</code> how ...<code>)</code></td><td>Recur and concatenate all non-<code>#f</code> values</td></tr>
<tr><td><code>(opt</code> how<code>)</code></td><td>Recur and surround the result with square brackets <code>[]</code> if non-<code>#f</code></td></tr>
<tr><td><code>(req</code> how<code>)</code></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 string to use as the key of the output hash table
type: see Figure 13.1
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] 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`.

[14] 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.

[15] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[16] 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.

[17] Ian Hickson. HTML 5, October 2014. `http://www.w3.org/TR/2014/REC-html5-20141028/`.

[18] 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.

[19] libuv. `http://libuv.org/`.

[20] Michael Owens. *The Definitive Guide to SQLite.* Apress, 2006.

[21] E. Resnick. Internet Message Format, April 2001. `http://www.ietf.org/rfc/rfc2822.txt`.

[22] Dorai Sitaram. pregexp: Portable Regular Expressions for Scheme and Common Lisp, 2005. `http://ds26gte.github.io/pregexp/index.html`.

[23] SQLite. `http://www.sqlite.org/`.

List of Figures

Index