SSH

version 1.0

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8.4 Document System.

Contents

SSH Reference Manual

Short Summaries

ssh

The following functions are exported:

- `start(Type) -> ok | {error, Reason}`
  [page 7] Starts the Ssh application.
- `stop() -> ok`
  [page 7] Stops the Ssh application.
- `stop_daemon(SysRef) ->`
  [page 7] Stops the listener and all connections started by the listener.
- `stop_daemon(Address, Port) -> ok`
  [page 7] Stops the listener and all connections started by the listener.
- `stop_listener(SysRef) ->`
  [page 7] Stops the listener, but leaves existing connections started by the listener up and running.
- `stop_listener(Address, Port) -> ok`
  [page 7] Stops the listener, but leaves existing connections started by the listener up and running.

ssh_connection

The following functions are exported:

- `adjust_window(ConnectionRef, Channel, Bytes) -> ok`
  [page 10] Adjusts the ssh flowcontrol window.
- `close(ConnectionRef, ChannelId) -> ok`
  [page 10] Sends a close message on the channel `ChannelId`.
- `exec(ConnectionRef, ChannelId, Command, TimeOut) -> success | failiure`
  [page 10] Will request that the server start the execution of the given command.
- `reply_request(ConnectionRef, WantReply, Status, CannelId) -> ok`
  [page 10] Send status replies to requests that want such replies.
- `send(ConnectionRef, ChannelId, Data) ->`
  [page 10] Sends channel data
- `send(ConnectionRef, ChannelId, Data, Timeout) ->`
  [page 10] Sends channel data
- `send(ConnectionRef, ChannelId, Type, Data) ->`
  [page 10] Sends channel data
- `send(ConnectionRef, ChannelId, Type, Data, TimeOut) -> ok | {error, timeout}`
  [page 10] Sends channel data
- `send_eof(ConnectionRef, ChannelId) -> ok`
  [page 11] Sends eof on the channel `ChannelId`.
- `session_channel(ConnectionRef, Timeout) ->`
  [page 11] Opens a channel for a ssh session. A session is a remote execution of a program. The program may be a shell, an application, a system command, or some built-in subsystem.
- `session_channel(ConnectionRef, InitialWindowSize, MaxPacketSize, Timeout) -> {ok, ChannleId} | {error, Reason}`
  [page 11] Opens a channel for a ssh session. A session is a remote execution of a program. The program may be a shell, an application, a system command, or some built-in subsystem.

- `setenv(ConnectionRef, ChannelId, Var, Value, TimeOut) -> success | failure`
  [page 11] Environment variables may be passed to the shell/command to be started later.
- `shell(ConnectionRef, ChannelId) -> success | failure`
  [page 11] Will request that the user's default shell (typically defined in /etc/passwd in UNIX systems) be started at the other end.
- `subsystem(ConnectionRef, ChannelId, Subsystem, Timeout) -> success | failure`
  [page 11]

ssh_sftp

The following functions are exported:

- `connect(CM) ->`
  [page 13] Connect to an SFTP server
- `connect(Host, Options) ->`
  [page 13] Connect to an SFTP server
- `connect(Host, Port, Options) -> {ok, Pid} | {error, Reason}`
  [page 13] Connect to an SFTP server
- `read_file(Server, File) -> {ok, Data} | {error, Reason}`
  [page 13] Read a file
- `write_file(Server, File, Iolist) -> ok | {error, Reason}`
  [page 13] Write a file
- `list_dir(Server, Path) -> {ok, Filenames} | {error, Reason}`
  [page 14] List directory
- `open(Server, File, Mode) -> {ok, Handle} | {error, Reason}`
  [page 14] Open a file and return a handle
- `opendir(Server, Path) -> {ok, Handle} | {error, Reason}`
  [page 14] Open a directory and return a handle
- `close(Server, Handle) -> ok | {error, Reason}`
  [page 14] Close an open handle
- `read(Server, Handle, Len) -> {ok, Data} | eof | {error, Error}`
  [page 14] Read from an open file
- `pread(Server, Handle, Position, Length) -> {ok, Data} | eof | {error, Error}`
  [page 14] Read from an open file
- `aread(Server, Handle, Len) -> {async, N} | {error, Error}`
  [page 15] Read asynchronously from an open file
- `apread(Server, Handle, Position, Length) -> {async, N} | {error, Error}`
  [page 15] Read asynchronously from an open file
- `write(Server, Handle, Data) -> ok | {error, Error}`
  [page 15] Write to an open file
- `pwrite(Server, Handle, Position, Data) -> ok | {error, Error}`
  [page 15] Write to an open file

- `awrite(Server, Handle, Data) -> ok | {error, Error}`
  [page 15] Write asynchronously to an open file
- `apwrite(Server, Handle, Position, Data) -> ok | {error, Error}`
  [page 15] Write asynchronously to an open file
- `position(Server, Handle, Location) -> {ok, NewPosition | {error, Error}`
  [page 16] Seek position in open file
- `read_file_info(Server, Name) -> {ok, FileInfo} | {error, Reason}`
  [page 16] Get information about a file
- `get_file_info(Server, Handle) -> {ok, FileInfo} | {error, Reason}`
  [page 16] Get information about a file
- `read_link_info(Server, Name) -> {ok, FileInfo} | {error, Reason}`
  [page 16] Get information about a symbolic link
- `write_file_info(Server, Name, Info) -> ok | {error, Reason}`
  [page 17] Write information for a file
- `read_link(Server, Name) -> {ok, Target} | {error, Reason}`
  [page 17] Read symbolic link
- `make_symlink(Server, Name, Target) -> ok | {error, Reason}`
  [page 17] Create symbolic link
- `rename(Server, OldName, NewName) -> ok | {error, Reason}`
  [page 17] Rename a file
- `delete(Server, Name) -> ok | {error, Reason}`
  [page 17] Delete a file
- `make_dir(Server, Name) -> ok | {error, Reason}`
  [page 18] Create a directory
- `del_dir(Server, Name) -> ok | {error, Reason}`
  [page 18] Delete an empty directory
- `stop(Server) -> ok`
  [page 18] Stop sftp session

ssh_sftpd

The following functions are exported:

- `subsystem_spec(Options) -> {"sftp", child_spec()}`
  [page 19] Returns a child specification that allows an ssh daemon to handle an sftp subssystem

SSH

Erlang Module

Interface module for the SSH application

COMMON DATA TYPES

Type definitions that are used more than once in this module:

`boolean() = true | false`

`string() = list of ASCII characters`

`ssh_system_ref() - opaque to the user returned by connect/[1,2,3]`

`ssh_connection_ref() - opaque to the user returned by listner/[]`

`ip_address() - {N1,N2,N3,N4} % IPv4 | {K1,K2,K3,K4,K5,K6,K7,K8} % IPv6`

`child_spec() - A child process specification see` [supervisor(3)]

Exports

`close(ConnectionRef) ->`

Types:

- ConnectionRef = ssh_connection_ref()

Closed a ssh connection.

```
connect(Host) ->
connect(Host, Options) ->
connect(Host, Port, Options) -> {ok, ssh_connection_ref()} | {error, Reason}
```

Types:

- Host = string()
- Port = integer()
  The default is 22, the registered port for SSH.
- Options = [{Option, Value}]

Connects to an SSH server. No channel is started this is done by calling ssh_connect:session_cahnnel/2.

Options are:

`{user_dir, String}` Sets the user directory, normally `~/.ssh` (containing the files `known_hosts`, `id_rsa<c>, <c>id_dsa`, `authorized_keys`).

{`silently_accept_hosts, Boolean`} When true, (default is false), hosts are added to the file `known_hosts` without asking the user.

{`user_interaction, Boolean`} If true, which is the default, password questions and adding hosts to `known_hosts` will be asked interactively to the user, if they are not suppressed by other options. (This is done during connection to an SSH server.) If false, both these events will throw and exception and the server will not start.

{`public_key_alg, ssh_rsa | ssh_dsa`} Sets the preferred public key algorithm to use for user authentication. If the the preferred algorithm fails of some reason, the other algorithm is tried. The default is to try `ssh_rsa` first.

{`connect_timeout, Milliseconds | infinity`} Sets the default timeout when trying to connect to.

{`user, String`} Provide a username. If this option is not given, ssh reads from the environment (`LOGNAME` or `USER` on unix, `USERNAME` on Windows).

{`password, String`} Provide a password for password authentication. If this option is not given, the user will be asked for a password if the password authentication method is attempted.

{`user_auth, Fun/3`} Provide a fun for password authentication. The fun will be called as `fun(User, Password, Opts)` and should return `true` or `false`.

{`key_cb, KeyCallbackModule`} Provide a special call-back module for key handling. The call-back module should be modeled after the `ssh_file` module. The function that must be exported are: `private_host_rsa_key/2`, `private_host_dsa_key/2`, `lookup_host_key/3` and `add_host_key/3`.

{`fd, FD`} Allow an existing file-descriptor to be used, given in FD. (Simply passed on to gen_tcp:connect.)

```
daemon(Port) ->
daemon(Port, Options) ->
daemon(HostAddress, Port, Options) -> ssh_system_ref()
```

Types:

- Port = integer()
- HostAddress = ip_address() | any
- Options = [{Option, Value}]
- Option = atom()
- Value = term()

Starts a server listening for SSH connections on the given port.

Options are:

{`subsystems, [{SubSysName, child_spec()}]`} Provides child specificatoions for handled subsystems. By default sftp is a handled subsystems.

{`shell, {Module, Function, Args} | fun()`} Defines what command line interface to use. Exampe use the erlang shell: {`shell, start, []`}`) which is the dafault behavior.`

{`system_dir, String`} Sets the system directory, containing the host files that identifies the host for ssh. The default is `/etc/ssh`, but note that SSH normally requires the host files there to be readable only by root.

{`user_passwords, [{User, Password}]`} Provide passwords for password authentication.They will be used when someone tries to connect to the server and public key user autentication fails. The option provides a list of valid user names and the corresponding password. `User` and `Password` are strings.

{`password, String`} Provide a global password that will authenticate any user (use with caution!).

{`pwdfun, fun/2`} Provide a function for password validation. This is called with user and password as strings, and should return `true` if the password is valid and `false` otherwise.

{`fd, FD`} Allow an existing file-descriptor to be used, given in FD. (Simply passed on to gen_tcp:listen.)

```
shell(Host) ->
shell(Host, Option) ->
shell(Host, Port, Option) -> _
```

Types:

- Host = string()
- Port = integer()
- Options - see ssh:connect/[1,2,3]

Starts an interactive shell to an SSH server on the given `Host`. The function waits for user input, and will not return until the remote shell is ended.(e.g. on exit from the shell)

```
start() ->
start(Type) -> ok | {error, Reason}
```

Types:

- Type = permanent | transient | temporary
- Reason = term()

Starts the Ssh application. Default type is temporary. See also [application(3)] Requiers that the crypto application has been started.

```
stop() -> ok
```

Stops the Ssh application. See also [application(3)]

```
stop_daemon(SysRef) ->
stop_daemon(Address, Port) -> ok
```

Types:

- SysRef = ssh_system_ref()
- Address = ip_address()
- Port = integer()

Stops the listener and all connections started by the listener.

```
stop_listener(SysRef) ->
stop_listener(Address, Port) -> ok
```

Types:

- SysRef = ssh_system_ref()
- Address = ip_address()
- Port = integer()

Stops the listener, but leaves existing connections started by the listener up and running.

SSH

Erlang Module

This module provides an API to the ssh connection protocol. Not all features of the connection protocol are officially supported yet. Only the ones supported are documented here.

COMMON DATA TYPES

Type definitions that are used more than once in this module:

```
boolean() = true | false
string() = list of ASCII characters
ssh_connection_ref() - opaque to the user returned by ssh:connect/[1,2,3]
or sent to a ssh channel processes
ssh_system_ref() - opaque to the user returned by ssh:daemon/[1,2,3]
```

MESSAGES SENT TO CHANNEL PROCESSES

As a result of the connection protocol the following messages may be sent to a channel process.

```
{ssh_cm, ConnectionRef, {open, ChannelId, RemoteChannelId, {session}}}
{ssh_cm, ConnectionRef, {subsystem, ChannelId, WantReply, Name}}
{ssh_cm, ConnectionRef, {closed, ChannelId}}
{ssh_cm, ConnectionRef, {data, ChannelId, Type, Data}}
{ssh_cm, ConnectionRef, {eof, ChannelId}}
{ssh_cm, ConnectionRef, {exit_signal,ChannelId, Signal, Err, Lang}}
{ssh_cm, ConnectionRef, {exit_status,ChannelId, Status}}
{ssh_cm, ConnectionRef, {shell, WantReply}}
{ssh_cm, ConnectionRef, {pty, ChannelId, WantReply, Pty}}
 {ssh_cm, ConnectionRef, {window_change, ChannelId, Width, Height, PixWidth, PixHeight
```

Exports

`adjust_window(ConnectionRef, Channel, Bytes) -> ok`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()
- Bytes = integer()

Adjusts the ssh flowcontrol window. Should be called after handling a {ssh_cm, ConnectionRef, {data, ChannelId, Type, Data}} message in the following way: `ssh_connection:adjust_window(ConnectionRef, ChannelId, size(Data)),`

`close(ConnectionRef, ChannelId) -> ok`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()

Sends a close message on the channel `ChannelId`

`exec(ConnectionRef, ChannelId, Command, TimeOut) -> success | failiure`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()
- Command = string()
- Timeout = infinity | integer()

Will request that the server start the execution of the given command. Result will be received as N x {ssh_cm, ConnectionRef, {data, ChannelId, Type, Data}} messages followed by {ssh_cm, ConnectionRef, {eof, ChannelId}} {ssh_cm, ConnectionRef, {exit_status, ChannelId, Status}}, {ssh_cm, ConnectionRef, {closed, ChannelId}}

`reply_request(ConnectionRef, WantReply, Status, CannelId) -> ok`

Types:

- ConnectionRef = ssh_connection_ref()
- WantReply = boolean()
- Status = success | failure
- ChannelId = integer()

Send status replies to requests that want such replies. Should be called after handling an ssh_cm-message containing a WantReply.

`send(ConnectionRef, ChannelId, Data) ->`
`send(ConnectionRef, ChannelId, Data, Timeout) ->`
`send(ConnectionRef, ChannelId, Type, Data) ->`
`send(ConnectionRef, ChannelId, Type, Data, TimeOut) -> ok | {error, timeout}`

Types:

- ConnectionRef = ssh_connection_ref()

- ChannelId = integer()
- Data = binary()
- Type = 0 | 1 see RFC 4254
- Timeout = infinity | integer()

Sends channel data.

`send_eof(ConnectionRef, ChannelId) -> ok`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()

Sends eof on the channel `ChannelId`.

`session_channel(ConnectionRef, Timeout) ->`
`session_channel(ConnectionRef, InitialWindowSize, MaxPacketSize, Timeout) -> {ok, ChannleId} | {error, Reason}`

Types:

- ConnectionRef = ssh_connection_ref()
- InitialWindowSize = integer()
- MaxPacketSize = integer()
- Timeout = infinity | integer()

Opens a channel for a ssh session. A session is a remote execution of a program. The program may be a shell, an application, a system command, or some built-in subsystem.

`setenv(ConnectionRef, ChannelId, Var, Value, TimeOut) -> success | failure`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()
- Var = string()
- Value = string()
- Timeout = infinity | integer()

Environment variables may be passed to the shell/command to be started later.

`shell(ConnectionRef, ChannelId) -> success | failure`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()

Will request that the user's default shell (typically defined in /etc/passwd in UNIX systems) be started at the other end. Messages for the caller to handle:

`subsystem(ConnectionRef, ChannelId, Subsystem, Timeout) -> success | failure`

Types:

- ConnectionRef = ssh_connection_ref()
- ChannelId = integer()
- Subsystem = string()

- Timeout = infinity | integer()

Sends a request to execute a predefined subsystem

ssh_sftp

Erlang Module

This module implements an SFTP (SSH FTP) client. SFTP is a secure, encrypted file transfer service available for SSH.

The errors returned are from the SFTP server, and are often not posix error codes.

Exports

`connect(CM) ->`
`connect(Host, Options) ->`
`connect(Host, Port, Options) -> {ok, Pid} | {error, Reason}`

Types:

- Host = string()
- CM = pid()
- Port = integer()
- Options = [{Option, Value}]
- Option = atom()
- Value = term()
- Reason = term()

Connects to an SFTP server. A `gen_server` is started and returned if connection is successful. This server is used to perform SFTP commands on the server.

For options, see `ssh:connect`.

`read_file(Server, File) -> {ok, Data} | {error, Reason}`

Types:

- Server = pid()
- File = string()
- Data = binary()
- Reason = term()

Reads a file from the server, and returns the data in a binary, like `file:read_file/1`.

`write_file(Server, File, Iolist) -> ok | {error, Reason}`

Types:

- Server = pid()
- File = string()
- Data = binary()

- Reason = term()

Writes a file to the server, like `file:write_file/2`. The file is created if it's not there.

`list_dir(Server, Path) -> {ok, Filenames} | {error, Reason}`

Types:

- Server = pid()
- Path = string()
- Filenames = [Filename]
- Filename = string()
- Reason = term()

Lists the given directory on the server, returning the filenames as a list of strings.

`open(Server, File, Mode) -> {ok, Handle} | {error, Reason}`

Types:

- Server = pid()
- File = string()
- Mode = [Modeflag]
- Modeflag = read | write | creat | trunc | append | binary
- Handle = term()
- Reason = term()

Opens a file on the server, and returns a handle that is used for reading or writing.

`opendir(Server, Path) -> {ok, Handle} | {error, Reason}`

Types:

- Server = pid()
- Path = string()
- Reason = term()

Opens a handle to a directory on the server, the handle is used for reading directory contents.

`close(Server, Handle) -> ok | {error, Reason}`

Types:

- Server = pid()
- Handle = term()
- Reason = term()

Closes a handle to an open file or directory on the server.

`read(Server, Handle, Len) -> {ok, Data} | eof | {error, Error}`
`pread(Server, Handle, Position, Length) -> {ok, Data} | eof | {error, Error}`

Types:

- Server = pid()
- Handle = term()
- Position = integer()

- Len = integer()
- Data = string() | binary()
- Reason = term()

Reads `Len` bytes from the file referenced by `Handle`. Returns {`ok, Data`}, or `eof`, or {`error, Reason`}. If the file is opened with `binary`, `Data` is a binary, otherwise it is a string.

If the file is read past eof, only the remaining bytes will be read and returned. If no bytes are read, `eof` is returned.

The `pread` function reads from a specified position, combining the `position` and `read` functions.

`aread(Server, Handle, Len) -> {async, N} | {error, Error}`
`apread(Server, Handle, Position, Length) -> {async, N} | {error, Error}`

Types:

- Server = pid()
- Handle = term()
- Position = integer()
- Len = integer()
- N = term()
- Reason = term()

Reads from an open file, without waiting for the result. If the handle is valid, the function returns {`async, N`}, where N is a term guaranteed to be unique between calls of `aread`. The actual data is sent as a message to the calling process. This message has the form {`async_reply, N, Result`}, where `Result` is the result from the read, either {`ok, Data`}, or `eof`, or {`error, Error`}.

The `apread` function reads from a specified position, combining the `position` and `aread` functions.

`write(Server, Handle, Data) -> ok | {error, Error}`
`pwrite(Server, Handle, Position, Data) -> ok | {error, Error}`

Types:

- Server = pid()
- Handle = term()
- Position = integer()
- Data = iolist()
- Reason = term()

Write `data` to the file referenced by `Handle`. The file should be opened with `write` or append flag. Returns `ok` if successful and {`error, Reason`} otherwise.

Typical error reasons are:

`ebadf` The file is not opened for writing.

`enospc` There is a no space left on the device.

`awrite(Server, Handle, Data) -> ok | {error, Error}`
`apwrite(Server, Handle, Position, Data) -> ok | {error, Error}`

Types:

- Server = pid()
- Handle = term()
- Position = integer()
- Len = integer()
- Data = binary()
- Reason = term()

Writes to an open file, without waiting for the result. If the handle is valid, the function returns `{async, N}`, where N is a term guaranteed to be unique between calls of `awrite`. The result of the `write` operation is sent as a message to the calling process. This message has the form `{async_reply, N, Result}`, where `Result` is the result from the write, either `ok`, or `{error, Error}`.

The `apwrite` writes on a specified position, combining the `position` and `awrite` operations.

`position(Server, Handle, Location) -> {ok, NewPosition | {error, Error}`

Types:

- Server = pid()
- Handle = term()
- Location = Offset | {bof, Offset} | {cur, Offset} | {eof, Offset} | bof | cur | eof
- Offset = int()
- NewPosition = integer()
- Reason = term()

Sets the file position of the file referenced by `Handle`. Returns `{ok, NewPosition` (as an absolute offset) if successful, otherwise `{error, Reason}`. `Location` is one of the following:

`Offset` The same as `{bof, Offset}`.

`{bof, Offset}` Absolute offset.

`{cur, Offset}` Offset from the current position.

`{eof, Offset}` Offset from the end of file.

`bof | cur | eof` The same as above with `Offset` 0.

`read_file_info(Server, Name) -> {ok, FileInfo} | {error, Reason}`
`get_file_info(Server, Handle) -> {ok, FileInfo} | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Handle = term()
- FileInfo = record()
- Reason = term()

Returns a `file_info` record from the file specified by `Name` or `Handle`, like `file:read_file_info/2`.

`read_link_info(Server, Name) -> {ok, FileInfo} | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Handle = term()
- FileInfo = record()
- Reason = term()

Returns a `file_info` record from the symbolic link specified by `Name` or `Handle`, like `file:read_link_info/2`.

`write_file_info(Server, Name, Info) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Info = record()
- Reason = term()

Writes file information from a `file_info` record to the file specified by `Name`, like `file:write_file_info`.

`read_link(Server, Name) -> {ok, Target} | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Target = string()
- Reason = term()

Read the link target from the symbolic link specified by `name`, like `file:read_link/1`.

`make_symlink(Server, Name, Target) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Target = string()
- Reason = term()

Creates a symbolic link pointing to `Target` with the name `Name`, like `file:make_symlink/2`.

`rename(Server, OldName, NewName) -> ok | {error, Reason}`

Types:

- Server = pid()
- OldName = string()
- NewName = string()
- Reason = term()

Renames a file named `OldName`, and gives it the name `NewName`, like `file:rename/2`

`delete(Server, Name) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Reason = term()

Deletes the file specified by `Name`, like `file:delete/1`

`make_dir(Server, Name) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Reason = term()

Creates a directory specified by `Name`. `Name` should be a full path to a new directory. The directory can only be created in an existing directory.

`del_dir(Server, Name) -> ok | {error, Reason}`

Types:

- Server = pid()
- Name = string()
- Reason = term()

Deletes a directory specified by `Name`. The directory should be empty, and

`stop(Server) -> ok`

Types:

- Server = pid()

Stops the `sftp` session, closing the connection. Any open files on the server will be closed.

ssh_sftpd

Erlang Module

Specifies a channel process to handle a sftp subsystem.

COMMON DATA TYPES

`child_spec() - A child process specification see` [supervisor(3)]

Exports

`subsystem_spec(Options) -> {"sftp", child_spec()}`

Types:

- Options = [{Option, Value}]

Should be used together with ssh:daemon/[1,2,3]

Options are:

`{cwd, String}` Sets the initial current working directory for the server.

`{file_handler, CallbackModule}` Determines which module to call for communicating with the file server. Default value is ssh_sftpd_file that uses the file and filelib API:s to access the standard OTP file server. This option may be used to plug in the use of other file servers.

`{root, String}` Sets the sftp root directory. The user will then not be able to see any files above this root. If for instance the root is set to `/tmp` the user will see this directory as `/` and if the user does cd `/etc` the user will end up in `/tmp/etc`.

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.