Net: PLT Networking Libraries

Version 4.1.2

October 28, 2008

Contents

1 URLs and HTTP

```
(require net/url)
```

The net/url library provides utilities to parse and manipulate URIs, as specified in RFC 2396 [RFC2396], and to use the HTTP protocol.

To access the text of a document from the web, first obtain its URL as a string. Convert the address into a url structure using string->url. Then, open the document using get-pure-port or get-impure-port, depending on whether or not you wish to examine its MIME headers. At this point, you have a regular input port with which to process the document, as with any other file.

Currently the only supported protocols are "http" and sometimes "file".

1.1 URL Structure

```
(require net/url-structs)
```

The URL structure types are provided by the net/url-structs library, and re-exported by net/url.

```
(struct url (scheme
             user
             host
             port
             path-absolute?
             path
             query
             fragment))
  scheme : (or/c false/c string?)
  user : (or/c false/c string?)
  host : (or/c false/c string?)
  port : (or/c false/c exact-nonnegative-integer?)
  path-absolute? : boolean?
  path : (listof path/param?)
  query : (listof (cons/c symbol? (or/c false/c string?)))
  fragment : (or/c false/c string?)
```

The basic structure for all URLs, hich is explained in RFC 3986 [RFC3986]. The following diagram illustrates the parts:

```
  http://sky@www:801/cgi-bin/finger;xyz?name=shriram;host=nw#top
  {-1}   {2} {3} {4}{---5---------} {6} {----7-------------} {8}
```

```
1 = scheme, 2 = user, 3 = host, 4 = port,
5 = path (two elements),  6 = param (of second path element),
7 = query, 8 = fragment
```

The strings inside the `user`, `path`, `query`, and `fragment` fields are represented directly as Scheme strings, without URL-syntax-specific quoting. The procedures `string->url` and `url->string` translate encodings such as `%20` into spaces and back again.

By default, query associations are parsed with either `;` or `&` as a separator, and they are generated with `&` as a separator. The `current-alist-separator-mode` parameter adjusts the behavior.

An empty string at the end of the `path` list corresponds to a URL that ends in a slash. For example, the result of `(string->url "http://www.drscheme.org/a/")` has a `path` field with strings `"a"` and `""`, while the result of `(string->url "http://www.drscheme.org/a")` has a `path` field with only the string `"a"`.

When a `"file"` URL is represented by a `url` structure, the `path` field is mostly a list of path elements. For Unix paths, the root directory is not included in `path`; its presence or absence is implicit in the `path-absolute?` flag. For Windows paths, the first element typically represents a drive, but a UNC path is represented by a first element that is `""` and then successive elements complete the drive components that are separated by `/` or `\`.

```
(struct path/param (path param))
  path : (or/c string? (one-of/c 'up 'same))
  param : (listof string?)
```

A pair that joins a path segment with its params in a URL.

1.2 URL Functions

An HTTP connection is created as a *pure port* or a *impure port*. A pure port is one from which the MIME headers have been removed, so that what remains is purely the first content fragment. An impure port is one that still has its MIME headers.

```
(string->url str) → url?
  str : string?
```

Parses the URL specified by *str* into a `url` struct. The `string->url` procedure uses `form-urlencoded->alist` when parsing the query, so it is sensitive to the `current-alist-separator-mode` parameter for determining the association separator.

If *str* starts with `"file:"`, then the path is always parsed as an absolute path, and the parsing details depend on `file-url-path-convention-type`:

- `'unix` : If `"file:"` is followed by `//` and a non-`/`, then the first element after the `//` is parsed as a host (and maybe port); otherwise, the first element starts the path, and the host is `""`.
- `'windows` : If `"file:"` is followed by `//`, then the `//` is stripped; the remainder parsed as a Windows path. The host is always `""` and the port is always `#f`.

`(combine-url/relative base relative)` → `url?`
  base : `url?`
  relative : `string?`

Given a base URL and a relative path, combines the two and returns a new URL as per the URL combination specification. They are combined according to the rules in RFC 3986 [RFC3986].

This function does not raise any exceptions.

`(netscape/string->url str)` → `url?`
  str : `string?`

Turns a string into a URL, applying (what appear to be) Netscape's conventions on automatically specifying the scheme: a string starting with a slash gets the scheme `"file"`, while all others get the scheme `"http"`.

`(url->string URL)` → `string?`
  URL : `url?`

Generates a string corresponding to the contents of a `url` struct. For a `"file:"` URL, the URL must not be relative, the result always starts `file://`, and the interpretation of the path depends on the value of `file-url-path-convention-type`:

- `'unix` : Elements in *URL* are treated as path elements. Empty strings in the path list are treated like `'same`.
- `'windows` : If the first element is `""` then the next two elements define the UNC root, and the rest of the elements are treated as path elements. Empty strings in the path list are treated like `'same`.

The `url->string` procedure uses `alist->form-urlencoded` when formatting the query,

so it is sensitive to the current-alist-separator-mode parameter for determining the association separator. The default is to separate associations with a &.

```
(path->url path) → url?
  path : (or/c path-string? path-for-some-system?)
```

Converts a path to a url.

```
(url->path URL [kind]) → path-for-some-system?
  URL : url?
  kind : (one-of/c 'unix 'windows)
       = (system-path-convention-type)
```

Converts *URL*, which is assumed to be a "file" URL, to a path.

```
(file-url-path-convention-type) → (one-of/c 'unix 'windows)
(file-url-path-convention-type kind) → void?
  kind : (one-of/c 'unix 'windows)
```

Determines the default conversion to and from strings for "file" URLs. See string->url and url->string.

```
(get-pure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
(head-pure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
(delete-pure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
```

Initiates a GET/HEAD/DELETE request for *URL* and returns a pure port corresponding to the body of the response. The optional list of strings can be used to send header lines to the server.

The GET method is used to retrieve whatever information is identified by *URL*.

The HEAD method is identical to GET, except the server must not return a message body. The meta-information returned in a response to a HEAD request should be identical to the information in a response to a GET request.

The DELETE method is used to delete the entity identified by *URL*.

The "file" scheme for URLs is handled only by get-pure-port, which uses open-input-file, does not handle exceptions, and ignores the optional strings.

```
(get-impure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
(head-impure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
(delete-impure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
```

Like get-pure-port, etc., but the resulting impure port contains both the returned headers and the body. The "file" URL scheme is not handled by these functions.

```
(post-pure-port URL post [header]) → input-port?
  URL : url?
  post : bytes?
  header : (listof string?) = null
(put-pure-port URL post [header]) → input-port?
  URL : url?
  post : bytes?
  header : (listof string?) = null
```

Initiates a POST/PUT request for *URL* and sends the *post* byte string. The result is a pure port, which contains the body of the response is returned. The optional list of strings can be used to send header lines to the server.

```
(post-impure-port URL post [header]) → input-port?
  URL : url?
  post : bytes?
  header : (listof string?) = null
(put-impure-port URL post [header]) → input-port?
  URL : url?
  post : bytes?
  header : (listof string?) = null
```

Like post-pure-port and put-pure-port, but the resulting impure port contains both the returned headers and body.

```
(display-pure-port in) → void?
  in : input-port?
```

Writes the output of a pure port, which is useful for debugging purposes.

```
(purify-port in) → string?
  in : input-port?
```

Purifies a port, returning the MIME headers, plus a leading line for the form `HTTP/`⟨*vers*⟩ ⟨*code*⟩ ⟨*message*⟩, where ⟨*vers*⟩ is something like `1.0` or `1.1`, ⟨*code*⟩ is an exact integer for the response code, and ⟨*message*⟩ is arbitrary text without a return or newline.

The `net/head` library provides procedures, such as `extract-field` for manipulating the header.

Since web servers sometimes return mis-formatted replies, `purify-port` is liberal in what it accepts as a header. as a result, the result string may be ill formed, but it will either be the empty string, or it will be a string matching the following regexp:

```
#rx"^HTTP/.*?(\r\n\r\n|\n\n|\r\r)"
```

```
(call/input-url URL connect handle) → any
  URL : url?
  connect : (url? . -> . input-port?)
  handle : (input-port? . -> . any)
(call/input-url URL connect handle header) → any
  URL : url?
  connect : (url? (listof string?) . -> . input-port?)
  handle : (input-port? . -> . any)
  header : (listof string?)
```

Given a URL and a *connect* procedure like `get-pure-port` to convert the URL to an input port (either a pure port or impure port), calls the *handle* procedure on the port and closes the port on return. The result of the *handle* procedure is the result of `call/input-url`.

When a *header* argument is supplied, it is passed along to the *connect* procedure.

The connection is made in such a way that the port is closed before `call/input-url` returns, no matter how it returns. In particular, it is closed if *handle* raises an exception, or if the connection process is interruped by an asynchronous break exception.

```
(current-proxy-servers)
 → (listof (list/c string? string? (integer-in 0 65535)))
(current-proxy-servers mapping) → void?
  mapping : (listof (list/c string? string? (integer-in 0 65535)))
```

A parameter that determines a mapping of proxy servers used for connections. Each mapping is a list of three elements:

- the URL scheme, such as `"http"`;
- the proxy server address; and
- the proxy server port number.

Currently, the only proxiable scheme is `"http"`. The default mapping is the empty list (i.e., no proxies).

1.3 URL Unit

```
(require net/url-unit)
```

`url@` : `unit?`

Imports `tcp^`, exports `url^`.

1.4 URL Signature

```
(require net/url-sig)
```

`url^` : signature

Includes everything exported by the `net/url` module.

2 URI Codec: Encoding and Decoding URIs

```
(require net/uri-codec)
```

The net/uri-codec module provides utilities for encoding and decoding strings using the URI encoding rules given in RFC 2396 [RFC2396], and to encode and decode name/value pairs using the application/x-www-form-urlencoded mimetype given the in HTML 4.0 specification. There are minor differences between the two encodings.

The URI encoding uses allows a few characters to be represented as-is: a through z, A through Z, 0-9, -, _, ., !, ~, *, ', (and). The remaining characters are encoded as %⟨*xx*⟩, where ⟨*xx*⟩ is the two-character hex representation of the integer value of the character (where the mapping character–integer is determined by US-ASCII if the integer is less than 128).

The encoding, in line with RFC 2396's recommendation, represents a character as-is, if possible. The decoding allows any characters to be represented by their hex values, and allows characters to be incorrectly represented as-is.

The rules for the application/x-www-form-urlencoded mimetype given in the HTML 4.0 spec are:

- Control names and values are escaped. Space characters are replaced by +, and then reserved characters are escaped as described in RFC 1738, section 2.2: Non-alphanumeric characters are replaced by %⟨*xx*⟩ representing the ASCII code of the character. Line breaks are represented as CRLF pairs: %0D%0A. Note that RFC 2396 supersedes RFC 1738 [RFC1738].
- The control names/values are listed in the order they appear in the document. The name is separated from the value by = and name/value pairs are separated from each other by either ; or &. When encoding, ; is used as the separator by default. When decoding, both ; and & are parsed as separators by default.

These rules differs slightly from the straight encoding in RFC 2396 in that + is allowed, and it represents a space. The net/uri-codec library follows this convention, encoding a space as + and decoding + as a space. In addtion, since there appear to be some brain-dead decoders on the web, the library also encodes !, ~, ', (, and) using their hex representation, which is the same choice as made by the Java's URLEncoder.

2.1 Functions

(uri-encode *str*) → string?
  str : string?

Encode a string using the URI encoding rules.

```
(uri-decode str) → string?
  str : string?
```

Decode a string using the URI decoding rules.

```
(form-urlencoded-encode str) → string?
  str : string?
```

Encode a string using the `application/x-www-form-urlencoded` encoding rules. The result string contains no non-ASCII characters.

```
(form-urlencoded-deecode str) → string?
  str : string?
```

Decode a string encoded using the `application/x-www-form-urlencoded` encoding rules.

```
(alist->form-urlencoded alist) → string?
  alist : (listof (cons/c symbol? string?))
```

Encode an association list using the `application/x-www-form-urlencoded` encoding rules.

The `current-alist-separator-mode` parameter determines the separator used in the result.

```
(form-urlencoded->alist str)
 → (listof (cons/c symbol? string?))
  str : string
```

Decode a string encoded using the `application/x-www-form-urlencoded` encoding rules into an association list. All keys are case-folded for conversion to symbols.

The `current-alist-separator-mode` parameter determines the way that separators are parsed in the input.

```
(current-alist-separator-mode)
 → (one-of/c 'amp 'semi 'amp-or-semi 'semi-or-amp)
(current-alist-separator-mode mode) → void?
  mode : (one-of/c 'amp 'semi 'amp-or-semi 'semi-or-amp)
```

A parameter that determines the separator used/recognized between associations in `form-urlencoded->alist`, `alist->form-urlencoded`, `url->string`, and `string->url`.

The default value is `'amp-or-semi`, which means that both `&` and `;` are treated as separators when parsing, and `&` is used as a separator when encoding. The other modes use/recognize only of the separators.

Examples:

```
> (define ex '((x . "foo") (y . "bar") (z . "baz")))
> (current-alist-separator-mode 'amp) ; try 'amp...
> (form-urlencoded->alist "x=foo&y=bar&z=baz")
((x . "foo") (y . "bar") (z . "baz"))
> (form-urlencoded->alist "x=foo;y=bar;z=baz")
((x . "foo;y=bar;z=baz"))
> (alist->form-urlencoded ex)
"x=foo&y=bar&z=baz"
> (current-alist-separator-mode 'semi) ; try 'semi...
> (form-urlencoded->alist "x=foo;y=bar;z=baz")
((x . "foo") (y . "bar") (z . "baz"))
> (form-urlencoded->alist "x=foo&y=bar&z=baz")
((x . "foo&y=bar&z=baz"))
> (alist->form-urlencoded ex)
"x=foo;y=bar;z=baz"
> (current-alist-separator-mode 'amp-or-semi) ; try 'amp-or-semi...
> (form-urlencoded->alist "x=foo&y=bar&z=baz")
((x . "foo") (y . "bar") (z . "baz"))
> (form-urlencoded->alist "x=foo;y=bar;z=baz")
((x . "foo") (y . "bar") (z . "baz"))
> (alist->form-urlencoded ex)
"x=foo&y=bar&z=baz"
> (current-alist-separator-mode 'semi-or-amp) ; try 'semi-or-amp...
> (form-urlencoded->alist "x=foo&y=bar&z=baz")
((x . "foo") (y . "bar") (z . "baz"))
> (form-urlencoded->alist "x=foo;y=bar;z=baz")
((x . "foo") (y . "bar") (z . "baz"))
> (alist->form-urlencoded ex)
"x=foo;y=bar;z=baz"
```

3 FTP: Client Downloading

```
(require net/ftp)
```

The net/ftp library provides utilities for FTP client operations.

The library was written by Micah Flatt.

3.1 Functions

```
(ftp-connection? v) → boolean?
  v : any/c
```

Returns #t if *v* represents an FTP connection as returned by ftp-establish-connection, #f otherwise.

```
(ftp-establish-connection server
                          port-no
                          user
                          passwd) → ftp-connection?
  server : string?
  port-no : (integer-in 0 65535)
  user : string?
  passwd : string?
```

Establishes an FTP connection with the given server using the supplied username and password.

The username and password strings are encoded to bytes using the current locale's encoding.

```
(ftp-close-connection ftp-conn) → void?
  ftp-conn : ftp-connection?
```

Closes an FTP connection.

```
(ftp-cd ftp-conn new-dir) → void?
  ftp-conn : ftp-connection?
  new-dir : string?
```

Changes the current directory on the FTP server to *new-dir*. The *new-dir* argument is not interpreted at all, but simply passed on to the server (encoded using the current locale's

encoding); it must not contain a newline.

```
(ftp-directory-list ftp-conn)
 → (listof (list/c (one-of/c "-" "d" "l")
                   string?
                   string?))
  ftp-conn : ftp-connection?
```

Returns a list of files and directories in the current directory of the server, assuming that the server provides directory information in the quasi-standard Unix format.

Each file or directory is represented by a list of three strings. The first string is either `"-"`, `"d"`, or `"l"`, depending on whether the items is a file, directory, or link, respectively. The second item is the file's date; to convert this value to seconds consistent with `file-seconds`, pass the date string to `ftp-make-file-seconds`, below. The third string is the name of the file or directory.

All strings are decoded from bytes using the current locale's encoding.

```
(ftp-make-file-seconds ftp-date) → exact-integer?
  ftp-date : string?
```

Takes a date string produced by `ftp-directory-list` and converts it to seconds (which can be used with `seconds->date`).

```
(ftp-download-file ftp-conn local-dir file) → void?
  ftp-conn : ftp-connection?
  local-dir : path-string?
  file : string?
```

Downloads *file* from the server's current directory and puts it in *local-dir* using the same name. If the file already exists in the local directory, it is replaced, but only after the transfer succeeds (i.e., the file is first downloaded to a temporary file, then moved into place on success).

3.2 FTP Unit

```
(require net/ftp-unit)
```

```
ftp@ : unit?
```

Imports nothing, exports `ftp^`.

3.3 FTP Signature

```
(require net/ftp-sig)
```

`ftp^` : signature

Includes everything exported by the `net/ftp` module.

4 Send URL: Opening a Web Browser

```
(require net/sendurl)
```

Provides `send-url` for opening a URL in the user's chosen web browser.

See also `browser/external`, which requires `scheme/gui`, but can prompt the user for a browser if no browser preference is set.

```
(send-url  str
           [separate-window?
            #:escape escape?]) → void?
  str : string?
  separate-window? : any/c = #t
  escape? : any/c = #t
```

Opens *str*, which represents a URL, in a platform-specific manner. For some platforms and configurations, the *separate-window?* parameter determines if the browser creates a new window to display the URL or not.

Under Windows, `send-url` normally uses `shell-execute` to launch a browser. (If the URL appears to contain a fragment, it may use an intermediate redirecting file due to a bug in IE7.)

Under Mac OS X, `send-url` runs `osascript` to start the user's chosen browser.

Under Unix, `send-url` uses the value of the `external-browser` parameter to select a browser.

The `url` string is usually escaped to avoid dangerous shell characters (quotations, dollar signs, backslashes, and non-ASCII). Note that it is a good idea to encode URLs before passing them to this function. Also note that the encoding is meant to make the URL work in shell quotes: URLs can still hold characters like `#`, `?`, and `&`, so the `external-browser` should use quotations.

```
(send-url/file  path
                [separate-window?
                 #:fragment fragment
                 #:query query])      → void?
  path : path-string?
  separate-window? : any/c = #t
  fragment : (or/c string? false/c) = #f
  query : (or/c string? false/c) = #f
```

Similar to `send-url`, but accepts a path to a file to be displayed by the browser. Use this

function when you want to display a local file: it takes care of the peculiarities of constructing the correct `file://` URL, and uses `send-url` to display the file. If you need to use an anchor fragment or a query string, use the corresponding keyword arguments.

```
(send-url/contents  contents
                    [separate-window?
                     #:fragment fragment
                     #:query query
                     #:delete-at seconds]) → void?
  contents : string?
  separate-window? : any/c = #t
  fragment : (or/c string? false/c) = #f
  query : (or/c string? false/c) = #f
  seconds : (or/c number? false/c) = #f
```

Similar to `send-url/file`, but it consumes the contents of a page to show, and displayes it from a temporary file.

If `delete-at` is a number, the temporary file is removed after this many seconds. The deletion happens in a thread, so if mzscheme exits before that it will not happen — when this function is called it scans old generated files (this happens randomly, not on every call) and removes them to avoid cluttering the temporary directory. If `delete-at` is `#f`, no delayed deletion happens, but old temporary files are still deleted as described above.

```
(external-browser) → browser-preference?
(external-browser cmd) → void?
  cmd : browser-preference?
```

A parameter that, under Unix, determines the browser started `send-url`.

The parameter is initialized to the value of the `'external-browser` preference.

The parameter value can be any of the symbols in `unix-browser-list`, `#f` to indicate that the preference is unset, or a pair of strings. If the preference is unset, `send-url` uses the first of the browsers from `unix-browser-list` for which the executable is found. If the parameter is a pair of strings, then a command line is constructed by concatenating in order the first string, the URL string, and the second string.

If the preferred or default browser can't be launched, `send-url` fails. See `get-preference` and `put-preferences` for details on setting preferences.

```
(browser-preference? a) → boolean?
  a : any/c
```

Returns `#t` if `v` is a valid browser preference, `#f` otherwise. See `external-browser` for more information.

`unix-browser-list` : `(listof symbol?)`

A list of symbols representing Unix executable names that may be tried in order by `send-url`. The `send-url` function internally includes information on how to launch each executable with a URL.

5 SMTP: Sending E-Mail

```
(require net/smtp)
```

The net/smtp module provides tools for sending electronic mail messages using SMTP. The client must provide the address of an SMTP server; in contrast, the net/sendmail module uses a pre-configured sendmail on the local system.

The net/head library defines the format of a header string, which is used by send-smtp-message. The net/head module also provides utilities to verify the formatting of a mail address. The procedures of the net/smtp module assume that the given string arguments are well-formed.

5.1 SMTP Functions

```
(smtp-send-message  server-address
                    from
                    to
                    header
                    message
                   [#:port-no port-no/k
                    #:auth-user user
                    #:auth-passwd pw
                    #:tcp-connect connect
                    #:tls-encode encode
                    port-no])              → void?
  server-address : string?
  from : string?
  to : (listof string?)
  header : string?
  message : (listof (or/c string? bytes?))
  port-no/k : (integer-in 0 65535) = 25
  user : (or/c string? false/c) = #f
  pw : (or/c string? false/c) = #f
  connect : ((string? (integer-in 0 65535))      = tcp-connect
             . ->* . (input-port? output-port?))
  encode : (or/c false/c                                = #f
                 ((input-port? output-port?
                   #:mode (one-of/c 'connect)
                   #:encrypt (one-of/c 'tls)
                   #:close-original? (one-of/c #t))
                  . ->* . (input-port? output-port?)))
  port-no : (integer-in 0 65535) = port-no/k
```

Connects to the server at *server-address* and *port-no* to send a message. The *from* argument specifies the mail address of the sender, and *to* is a list of recipient addresses (including "To:", "CC", and "BCC" recipients).

The *header* argument is the complete message header, which should already include "From:", "To:", and "CC:" fields consistent with the given sender and recipients. See also the net/head library for header-creating utilities.

The *message* argument is the body of the message, where each string or byte string in the list corresponds to a single line of message text. No string in *message* should contain a carriage return or linefeed character.

The optional *port-no* argument—which can be specified either with the #:port-no keyword or, for backward compatibility, as an extra argument after keywords—specifies the IP port to use in contacting the SMTP server.

The optional #:auth-user and #:auth-passwd keyword argument supply a username and password for authenticated SMTP (using the AUTH PLAIN protocol).

The optional #:tcp-connect keyword argument supplies a connection procedure to be used in place of tcp-connect. For example, use ssl-connect to connect to the server via SSL.

If the optional #:tls-encode keyword argument supplies a procedure instead of #f, then the ESMTP STARTTLS protocol is used to initiate SSL communication with the server. The procedure given as the #:tls-encode argument should be like ports->ssl-ports; it will be called as

```
(encode r w #:mode 'connect #:encrypt 'tls #:close-original? #t)
```

and it should return two values: an input port and an export port. All further SMTP communication uses the returned ports.

For encrypted communication, normally either ssl-connect should be supplied for #:tcp-connect, or ports->ssl-ports should be supplied for #:tls-encode—one or the other (depending on what the server expects), rather than both.

```
(smtp-sending-end-of-message) → (-> any)
(smtp-sending-end-of-message proc) → void?
  proc : (-> any)
```

A parameter that determines a send-done procedure to be called after smtp-send-message has completely sent the message. Before the send-done procedure is called, breaking the thread that is executing smtp-send-message cancels the send. After the send-done procedure is called, breaking may or may not cancel the send (and probably will not).

5.2 SMTP Unit

```
(require net/smtp-unit)
```

---

smtp@ : unit?

Imports nothing, exports smtp^.

5.3 SMTP Signature

```
(require net/smtp-sig)
```

---

smtp^ : signature

Includes everything exported by the net/smtp module.

6 `sendmail`: Sending E-Mail

```
(require net/sendmail)
```

The `net/sendmail` module provides tools for sending electronic mail messages using a `sendmail` program on the local system. See also the `net/smtp` package, which sends mail via SMTP.

All strings used in mail messages are assumed to conform to their corresponding SMTP specifications, except as noted otherwise.

6.1 Sendmail Functions

```
(send-mail-message/port from
                        subject
                        to
                        cc
                        bcc
                        extra-header ...) → output-port?
  from : string?
  subject : string?
  to : (listof string?)
  cc : (listof string?)
  bcc : (listof string?)
  extra-header : string?
```

The first argument is the header for the sender, the second is the subject line, the third a list of "To:" recipients, the fourth a list of "CC:" recipients, and the fifth a list of "BCC:" recipients. Additional arguments argument supply other mail headers, which must be provided as lines (not terminated by a linefeed or carriage return) to include verbatim in the header.

The return value is an output port into which the client must write the message. Clients are urged to use `close-output-port` on the return value as soon as the necessary text has been written, so that the sendmail process can complete.

The *from* argument can be any value; of course, spoofing should be used with care.

```
(send-mail-message from
                   subject
                   to
                   cc
                   bcc
                   body
                   extra-header ...) → void?
  from : string?
  subject : string?
  to : (listof string?)
  cc : (listof string?)
  bcc : (listof string?)
  body : (listof string?)
  extra-header : string?
```

Like send-mail-message/port, but with *body* as a list of strings, each providing a line of the message body.

Lines that contain a single period do not need to be quoted.

```
(struct (no-mail-recipients exn) ())
```

Raised when no mail recipients were specified for send-mail-message/port.

6.2 Sendmail Unit

```
(require net/sendmail-unit)
```

---

sendmail@ : unit?

Imports nothing, exports `sendmail^`.

6.3 Sendmail Signature

```
(require net/sendmail-sig)
```

`sendmail^` : signature

Includes everything exported by the `net/sendmail` module.

7 Headers: Parsing and Constructing

```
(require net/head)
```

The net/head module provides utilities for parsing and constructing RFC 822 headers [RFC822], which are used in protocols such as HTTP, SMTP, and NNTP.

A *header* is represented as a string or byte string containing CRLF-delimited lines. Each field within the header spans one or more lines. In addition, the header ends with two CRLFs (because the first one terminates the last field, and the second terminates the header).

7.1 Functions

```
empty-header : string?
```

The string "\r\n\r\n", which corresponds to the empty header. This value is useful for building up headers with insert-field and append-headers.

```
(validate-header candidate) → void?
  candidate : (or string? bytes?)
```

Checks that *candidate* matches RFC 822. If it does not, an exception is raised.

```
(extract-field field header) → (or/c string? bytes? false/c)
  field : (or/c string? bytes?)
  header : (or/c string? bytes?)
```

Returns the header content for the specified field, or #f if the field is not in the header. The *field* string should not end with ":", and it is used case-insensitively. The returned string will not contain the field name, color separator, or CRLF terminator for the field; however, if the field spans multiple lines, the CRLFs separating the lines will be intact.

The *field* and *header* arguments must be both strings or both byte strings, and the result (if not #f) is of the same type.

Examples:

```
> (extract-field "TO" (insert-field "to" "me@localhost"
                                    empty-header))
"me@localhost"
```

```
(extract-all-fields header)
```

```
→ (listof (cons/c (or/c string? bytes?)
                  (or/c string? bytes?)))
  header : (or/c string? bytes?)
```

Returns an association-list version of the header; the case of the field names is preserved, as well as the order and duplicate uses of a field name.

The result provides strings if *header* is a string, byte strings if *header* is a byte string.

```
(remove-field field header) → (or/c string? bytes?)
  field : (or/c string? bytes?)
  header : (or/c string? bytes?)
```

Creates a new header by removing the specified field from *header* (or the first instance of the field, if it occurs multiple times). If the field is not in *header*, then the return value is *header*.

The *field* and *header* arguments must be both strings or both byte strings, and the result is of the same type.

```
(insert-field field value header) → (or/c string? bytes?)
  field : (or/c string? bytes?)
  value : (or/c string? bytes?)
  header : (or/c string? bytes?)
```

Creates a new header by prefixing the given *header* with the given *field-value* pair. The *value* string should not contain a terminating CRLF, but a multi-line value (perhaps created with `data-lines->data`) may contain separator CRLFs.

The *field*, *value*, and *header* arguments must be all strings or all byte strings, and the result is of the same type.

```
(replaces-field field value header) → (or/c string? bytes?)
  field : (or/c string? bytes?)
  value : (or/c string? bytes? false/c)
  header : (or/c string? bytes?)
```

Composes `remove-field` and (if *value* is not `#f`) `insert-field`.

```
(append-headers header1 header2) → (or/c string? bytes?)
  header1 : (or/c string? bytes?)
  header2 : (or/c string? bytes?)
```

Appends two headers.

The *header1* and *header2* arguments must be both strings or both byte strings, and the result is of the same type.

```
(standard-message-header from
                         to
                         cc
                         bcc
                         subject) → string?
  from : string?
  to : (listof -string?)
  cc : (listof strings?)
  bcc : (listof string?)
  subject : string?
```

Creates a standard mail header given the sender, various lists of recipients, a subject. A `"Date"` field is added to the header automatically, using the current time.

The BCC recipients do not actually appear in the header, but they're accepted anyway to complete the abstraction.

```
(data-lines->data listof) → string?
  listof : string?
```

Merges multiple lines for a single field value into one string, adding CRLF-TAB separators.

```
(extract-addresses line kind)
 → (or/c (listof string?)
         (listof (list/c string? string? string?)))
  line : string?
  kind : (one-of/c 'name 'address
                   'full 'all)
```

Parses `string` as a list of comma-delimited mail addresses, raising an exception if the list is ill-formed. This procedure can be used for single-address strings, in which case the returned list contains only one address.

The *kind* argument specifies which portion of an address should be returned:

- `'name` — the free-form name in the address, or the address itself if no name is available.

Examples:

```
> (extract-addresses "John Doe <doe@localhost>" 'name)
("John Doe")
> (extract-addresses "doe@localhost (Johnny Doe)" 'name)
("Johnny Doe")
> (extract-addresses "doe@localhost" 'name)
("doe@localhost")
> (extract-addresses " \"Doe, John\" <doe@localhost>, jane"
                     'name)
("\"Doe, John\"" "jane")
```

- `'address` — just the mailing address, without any free-form names.

Examples:

```
> (extract-addresses "John Doe <doe@localhost>" 'address)
("doe@localhost")
> (extract-addresses "doe@localhost (Johnny Doe)" 'address)
("doe@localhost")
> (extract-addresses "doe@localhost" 'address)
("doe@localhost")
> (extract-addresses " \"Doe, John\" <doe@localhost>, jane"
                     'address)
("doe@localhost" "jane")
```

- `'full` — the full address, essentially as it appears in the input, but normalized.

Examples:

```
> (extract-addresses "John Doe   < doe@localhost >" 'full)
("John Doe <doe@localhost>")
> (extract-addresses "  doe@localhost  (Johnny Doe)" 'full)
("doe@localhost (Johnny Doe)")
> (extract-addresses "doe@localhost" 'full)
("doe@localhost")
> (extract-addresses " \"Doe, John\" <doe@localhost>, jane"
                     'full)
("\"Doe, John\" <doe@localhost>" "jane")
```

- `'all` — a list containing each of the three possibilities: free-form name, address, and full address (in that order).

Examples:

```
> (extract-addresses "John Doe <doe@localhost>" 'all)
(("John Doe" "doe@localhost" "John Doe <doe@localhost>"))
> (extract-addresses "doe@localhost (Johnny Doe)" 'all)
(("Johnny Doe" "doe@localhost" "doe@localhost (Johnny Doe)"))
> (extract-addresses "doe@localhost" 'all)
(("doe@localhost" "doe@localhost" "doe@localhost"))
> (define r
```

```
    (extract-addresses " \"John\" <doe@localhost>, jane"
                       'all))
> (length r)
2
> (car r)
("\"John\"" "doe@localhost" "\"John\" <doe@localhost>")
> (cadr r)
("jane" "jane" "jane")
```

(assemble-address-field *addrs*) → string?
  addrs : (listof string?)

Creates a header field value from a list of addresses. The addresses are comma-separated, and possibly broken into multiple lines.

Examples:
```
> (assemble-address-field '("doe@localhost"
                            "Jane <jane@elsewhere>"))
"doe@localhost, Jane <jane@elsewhere>"
```

7.2 Header Unit

```
(require net/head-unit)
```

---

head@ : unit?

Imports nothing, exports `head^`.

7.3 Header Signature

```
(require net/head-sig)
```

`head^` : signature

Includes everything exported by the `net/head` module.

8 IMAP: Reading Mail

```
(require net/imap)
```

The net/imap module provides utilities for the client side of Internet Message Access Protocol version 4rev1 [RFC2060].

8.1 Connecting and Selecting Mailboxes

```
(imap-connection? v) → boolean?
  v : any/c
```

Return #t if v is a IMAP-connection value (which is opaque), #f otherwise.

```
(imap-connect server
              username
              password
              mailbox) → imap-connection?
                         exact-nonnegative-integer?
                         exact-nonnegative-integer?
  server : string?
  username : (or/c string? bytes?)
  password : (or/c string? bytes?)
  mailbox : (or/c string? bytes?)
```

Establishes an IMAP connection to the given server using the given username and password, and selects the specified mailbox. The first result value reprsents the connection.

The second and third return values indicate the total number of messages in the mailbox and the number of recent messages (i.e., messages received since the mailbox was last selected), respectively.

See also imap-port-number.

A user's primary mailbox is always called "INBOX". (Capitalization doesn't matter for that mailbox name.)

Updated message-count and recent-count values are available through imap-messages and imap-recent. See also imap-new? and imap-reset-new!.

```
(imap-port-number) → (integer-in 0 65535)
(imap-port-number k) → void?
```

```
  k : (integer-in 0 65535)
```

A parameter that determines the server port number. The initial value is 143.

```
(imap-connect* in
               out
               username
               password
               mailbox) → imap-connection?
                          exact-nonnegative-integer?
                          exact-nonnegative-integer?
  in : input-port?
  out : output-port?
  username : (or/c string? bytes?)
  password : (or/c string? bytes?)
  mailbox : (or/c string? bytes?)
```

Like imap-connect, but given input and output ports (e.g., ports for an SSL session) instead of a server address.

```
(imap-disconnect imap) → void?
  imap : imap-connection?
```

Closes an IMAP connection. The close may fail due to a communication error.

```
(imap-force-disconnect imap) → void?
  imap : imap-connection?
```

Closes an IMAP connection forcefully (i.e., without send a close message to the server). A forced disconnect never fails.

```
(imap-reselect imap mailbox) → exact-nonnegative-integer?
                               exact-nonnegative-integer?
  imap : imap-connection?
  mailbox : (or/c string? bytes?)
```

De-selects the mailbox currently selected by the connection and selects the specified mailbox, returning the total and recent message counts for the new mailbox. Expunge and message-state information is removed.

Do not use this procedure to poll a mailbox to see whether there are any new messages. Use imap-noop, imap-new?, and imap-reset-new! instead.

```
(imap-examine imap mailbox) → exact-nonnegative-integer?
                              exact-nonnegative-integer?
  imap : imap-connection?
  mailbox : (or/c string? bytes?)
```

Like `imap-reselect`, but the mailbox is selected as read-only.

8.2 Selected Mailbox State

```
(imap-noop imap) → exact-nonnegative-integer?
                   exact-nonnegative-integer?
  imap : imap-connection?
```

Sends a "no-op" message to the server, typically to keep the session alive. As for many commands, the server may report message-state updates or expunges, which are recorded in *imap*.

The return information is the same as for `imap-reselect`.

```
(imap-poll imap) → void?
  imap : imap-connection?
```

Does not send a request to the server, but checks for asynchronous messages from the server that update the message count, to report expunges, etc.

```
(imap-messages imap) → exact-nonnegative-integer?
  imap : imap-connection?
```

Returns the number of messages in the selected mailbox. The server can update this count during most any interaction.

This operation does not communicate with the server. It merely reports the result of previous communication.

```
(imap-recent imap) → exact-nonnegative-integer?
  imap : imap-connection?
```

Returns the number of "recent" messages in the currently selected mailbox, as most recently reported by the server. The server can update this count during most any interaction.

This operation does not communicate with the server. It merely reports the result of previous communication.

```
(imap-unseen imap) → (or/c exact-nonnegative-integer? false/c)
  imap : imap-connection?
```

Returns the number of "unseen" messages in the currently selected mailbox, as most recently reported by the server. The server can update this count during most any interaction. Old IMAP servers might not report this value, in which case the result is `#f`.

This operation does not communicate with the server. It merely reports the result of previous communication.

```
(imap-uidnext imap) → (or/c exact-nonnegative-integer? false/c)
  imap : imap-connection?
```

Returns the predicted next uid for a message in the currently selected mailbox, as most recently reported by the server. The server can update this count during most any interaction. Old IMAP servers might not report this value, in which case the result is `#f`.

This operation does not communicate with the server. It merely reports the result of previous communication.

```
(imap-uidvalidity imap)
 → (or/c exact-nonnegative-integer? false/c)
  imap : imap-connection?
```

Returns an id number that changes when all uids become invalid. The server *cannot* update this number during a session. Old IMAP servers might not report this value, in which case the result is `#f`.

This operation does not communicate with the server. It merely reports the result of previous communication.

```
(imap-new? imap) → boolean?
  imap : imap-connection?
```

Returns `#t` if the server has reported an increase in the message count for the currently mailbox since the last call to `imap-reset-new!`. Selecting a mailbox implicitly calls `imap-reset-new!`.

This operation does not communicate with the server. It merely reports the result of previous communication.

```
(imap-reset-new! imap) → void?
  imap : imap-connection?
```

Resets the new flag for the session; see `imap-new?`. This operation does not communicate with the server.

```
(imap-get-expunges imap) → (listof exact-nonnegative-integer?)
  imap : imap-connection?
```

Returns pending expunge notifications from the server for the selected mailbox in terms of message positions (not uids), and clears the pending notifications. The result list is sorted, ascending.

This operation does not communicate with the server. It merely reports the result of previous communication.

The server can notify the client of newly deleted messages during most other commands, but not asynchronously between commands. Furthermore, the server cannot report new deletions during `imap-get-messages` or `imap-store` operations.

Before calling any IMAP operation that works in terms of message numbers, pending expunge notifications must be handled by calling `imap-get-expunges`.

```
(imap-pending-expunges? imap) → boolean?
  imap : imap-connection?
```

Returns `#f` if `imap-get-expunges` would return an empty list, `#t` otherwise.

```
(imap-get-updates imap)
 → (listof (cons/c exact-nonnegative-integer?
                   (listof pair?)))
  imap : imap-connection?
```

Returns information must like `imap-get-messages`, but includes information reported asynchronously by the server (e.g., to notify a client with some other client changes a message attribute). Instead of reporting specific requested information for specific messages, the result is associates message positions to field-value association lists. The result list is sorted by message position, ascending.

This operation does not communicate with the server. It merely reports the result of previous communication. It also clears the update information from the connection after reporting it.

When a server reports information that supersedes old reported information for a message,

or if the server reports that a message has been deleted, then old information for the message is dropped. Similarly, if `imap-get-messages` is used to explicitly obtain information, any redundant (or out-of-date) information is dropped.

A client need not use `imap-get-updates` ever, but accumulated information for the connection consumes space.

```
(imap-pending-updates? imap) → boolean?
  imap : imap-connection?
```

Returns `#f` if `imap-get-updates` would return an list, `#t` otherwise.

8.3 Manipulating Messages

```
(imap-get-messages imap msg-nums fields) → (listof list?)
  imap : imap-connection?
  msg-nums : (listof exact-nonnegative-integer?)
  fields : (listof (one-of/c 'uid
                             'header
                             'body
                             'flags))
```

Downloads information for a set of messages. The *msg-nums* argument specifies a set of messages by their message positions (not their uids). The *fields* argument specifies the type of information to download for each message. The available fields are:

- `'uid` — the value is an integer
- `'header` — the value is a header (a string, but see `net/head`)
- `'body` — the value is a byte string, with CRLF-separated lines
- `'flags` — the value is a list of symbols that correspond to IMAP flags; see `imap-flag->symbol`

The return value is a list of entry items in parallel to *msg-nums*. Each entry is itself a list containing value items in parallel to *fields*.

Pending expunges must be handled before calling this function; see `imap-get-expunges`.

Examples:

```
> (imap-get-message imap '(1 3 5) '(uid header))
((107 #"From: larry@stooges.com ...") (110 #"From: moe@stooges.com ...") (112 #"From: curly@
```

```
(imap-flag->symbol flag) → symbol?
  flag : symbol?
(symbol->imap-flag sym) → symbol?
  sym : symbol?
```

An IMAP flag is a symbol, but it is generally not a convenient one to use within a Scheme program, because it usually starts with a backslash. The `imap-flag->symbol` and `symbol->imap-flag` procedures convert IMAP flags to convenient symbols and vice-versa:

	symbol	*IMAP flag*
message flags:	`'seen`	`'|\Seen|`
	`'answered`	`'|\Answered|`
	`'flagged`	`'|\Flagged|`
	`'deleted`	`'|\Deleted|`
	`'draft`	`'|\Draft|`
	`'recent`	`'|\Recent|`
mailbox flags:	`'noinferiors`	`'|\Noinferiors|`
	`'noselect`	`'|\Noselect|`
	`'marked`	`'|\Marked|`
	`'unmarked`	`'|\Unmarked|`
	`'hasnochildren`	`'|\HasNoChildren|`
	`'haschildren`	`'|\HasChildren|`

The `imap-flag->symbol` and `symbol->imap-flag` functions act like the identity function when any other symbol is provided.

```
(imap-store imap mode msg-nums imap-flags) → void?
  imap : imap-connection?
  mode : (one-of/c '+ '- '!)
  msg-nums : (listof exact-nonnegative-integer?)
  imap-flags : (listof symbol?)
```

Sets flags for a set of messages. The mode argument specifies how flags are set:

- `'+` — add the given flags to each message
- `'-` — remove the given flags from each message
- `'!` — set each message's flags to the given set

The *msg-nums* argument specifies a set of messages by their message positions (not their uids). The `flags` argument specifies the imap flags to add/remove/install.

Pending expunges must be handled before calling this function; see `imap-get-expunges`.

The server will not report back message-state changes (so they will not show up through `imap-get-updates`).

Examples:

```
> (imap-store imap '+ '(1 2 3) (list (symbol->imap-flag 'deleted)))
; marks the first three messages to be deleted
> (imap-expunge imap)
; permanently removes the first three messages (and possibly
; others) from the currently-selected mailbox
```

```
(imap-expunge imap) → void?
  imap : imap-connection?
```

Purges every message currently marked with the `'|\Deleted|` flag from the mailbox.

8.4 Querying and Changing (Other) Mailboxes

```
(imap-copy imap msg-nums dest-mailbox) → void?
  imap : imap-connection?
  msg-nums : (listof exact-nonnegative-integer?)
  dest-mailbox : (or/c string? bytes?)
```

Copies the specified messages from the currently selected mailbox to the specified mailbox.

Pending expunges must be handled before calling this function; see `imap-get-expunges`.

```
(imap-append imap mailbox message) → void?
  imap : imap-connection?
  mailbox : string?
  message : (or/c string? bytes?)
```

Adds a new message (containing *message*) to the given mailbox.

```
(imap-status imap mailbox statuses) → list?
  imap : imap-connection?
  mailbox : (or/c string? bytes?)
  statuses : (listof symbol?)
```

Requests information about a mailbox from the server, typically *not* the currently selected mailbox.

The *statuses* list specifies the request, and the return value includes one value for each symbol in *statuses*. The allowed status symbols are:

- 'messages — number of messages
- 'recent — number of recent messages
- 'unseen — number of unseen messages
- 'uidnext — uid for next received message
- 'uidvalidity — id that changes when all uids are changed

Use imap-messages to get the message count for the currently selected mailbox, etc. Use imap-new? and imap-reset-new! to detect when new messages are available in the currently selected mailbox.

```
(imap-mailbox-exists? imap mailbox) → boolean?
  imap : imap-connection?
  mailbox : (or/c string? bytes?)
```

Returns #t if *mailbox* exists, #f otherwise.

```
(imap-create-mailbox imap mailbox) → void?
  imap : imap-connection?
  mailbox : (or/c string? bytes?)
```

Creates *mailbox*. (It must not exist already.)

```
(imap-list-child-mailboxes  imap
                            mailbox
                           [delimiter])
 → (listof (list/c (listof symbol?) bytes?))
  imap : imap-connection?
  mailbox : (or/c string? bytes? false/c)
  delimiter : (or/c string? bytes?)
            = (imap-get-hierarchy-delimiter)
```

Returns information about sub-mailboxes of *mailbox*; if *mailbox* is #f, information about all top-level mailboxes is returned. The *delimiter* is used to parse mailbox names from the server to detect hierarchy.

The return value is a list of mailbox-information lists. Each mailbox-information list contains two items:

- a list of imap flags for the mailbox
- the mailbox's name

```
(imap-get-hierarchy-delimiter imap) → bytes?
  imap : imap-connection?
```

Returns the server-specific string that is used as a separator in mailbox path names.

```
(imap-mailbox-flags imap mailbox) → (listof symbol?)
  imap : imap-connection?
  mailbox : (or/c string? bytes?)
```

Returns a list of IMAP flags for the given mailbox. See also `imap-flag->symbol`.

8.5 IMAP Unit

```
(require net/imap-unit)
```

`imap@` : unit?

Imports nothing, exports `imap^`.

8.6 IMAP Signature

```
(require net/imap-sig)
```

`imap^` : signature

Includes everything exported by the `net/imap` module.

9 POP3: Reading Mail

```
(require net/pop3)
```

The net/pop3 module provides tools for the Post Office Protocol version 3 [RFC977].

```
(struct communicator (sender receiver server port state))
  sender : output-port?
  receiver : input-port?
  server : string?
  port : (integer-in 0 65535)
  state : (one-of/c 'disconnected 'authorization 'transaction)
```

Once a connection to a POP-3 server has been established, its state is stored in a communicator instance, and other procedures take communicator instances as an argument.

```
(connect-to-server server [port-number]) → communicator?
  server : string?
  port-number : (integer-in 0 65535) = 110
```

Connects to *server* at *port-number*.

```
(disconnect-from-server communicator) → void?
  communicator : communicator?
```

Disconnects *communicator* from the server, and sets *communicator*'s state to 'disconnected.

```
(authenticate/plain-text user
                         passwd
                         communicator) → void?
  user : string?
  passwd : string?
  communicator : communicator?
```

Authenticates using *user* and *passwd*. If authentication is successful, *communicator*'s state is set to 'transaction.

```
(get-mailbox-status communicator) → exact-nonnegative-integer?
                                    exact-nonnegative-integer?
  communicator : communicator?
```

Returns the number of messages and the number of octets in the mailbox.

```
(get-message/complete communicator
                      message-number)
 → (listof string?) (listof string?)
  communicator : communicator?
  message-number : exact-integer?
```

Given a message number, returns a list of message-header lines and list of message-body lines.

```
(get-message/headers communicator
                     message-number)
 → (listof string?) (listof string?)
  communicator : communicator?
  message-number : exact-integer?
```

Given a message number, returns a list of message-header lines.

```
(get-message/body communicator
                  message-number)
 → (listof string?) (listof string?)
  communicator : communicator?
  message-number : exact-integer?
```

Given a message number, returns a list of message-body lines.

```
(delete-message communicator
                message-number) → void?
  communicator : communicator?
  message-number : exact-integer?
```

Deletes the specified message.

```
(get-unique-id/single communicator
                      message-number) → string?
  communicator : communicator?
  message-number : exact-integer?
```

Gets the server's unique id for a particular message.

```
(get-unique-id/all communicator)
 → (listof (cons/c exact-integer? string?))
  communicator : communicator?
```

Gets a list of unique id's from the server for all the messages in the mailbox. The `car` of each item in the result list is the message number, and the `cdr` of each item is the message's id.

```
(make-desired-header tag-string) → regexp?
  tag-string : string?
```

Takes a header field's tag and returns a regexp to match the field

---

```
(extract-desired-headers header desireds) → (listof string?)
  header : (listof string?)
  desireds : (listof regexp?)
```

Given a list of header lines and of desired regexps, returns the header lines that match any of the *desireds*.

9.1 Exceptions

```
(struct (pop3 exn) ())
```

The supertype of all POP3 exceptions.

```
(struct (cannot-connect pop3) ())
```

Raised when a connection to a server cannot be established.

```
(struct (username-rejected pop3) ())
```

Raised if the username is rejected.

```
(struct (password-rejected pop3) ())
```

Raised if the password is rejected.

```
(struct (not-ready-for-transaction pop3) (communicator))
```

```
  communicator : communicator?
```

Raised when the communicator is not in transaction mode.

```
(struct (not-given-headers pop3) (communicator message))
  communicator : communicator?
  message : exact-integer?
```

Raised when the server does not respond with headers for a message as requested.

```
(struct (illegal-message-number pop3) (communicator message))
  communicator : communicator?
  message : exact-integer?
```

Raised when the client specifies an illegal message number.

```
(struct (cannot-delete-message exn) (communicator message))
  communicator : communicator?
  message : exact-integer?
```

Raised when the server is unable to delete a message.

```
(struct (disconnect-not-quiet pop3) (communicator))
  communicator : communicator?
```

Raised when the server does not gracefully disconnect.

```
(struct (malformed-server-response pop3) (communicator))
  communicator : communicator?
```

Raised when the server produces a mal-formed response.

9.2 Example Session

```
> (require net/pop3)
> (define c (connect-to-server "cs.rice.edu"))
> (authenticate/plain-text "scheme" "********" c)
> (get-mailbox-status c)
196
816400
```

```
> (get-message/headers c 100)
("Date: Thu, 6 Nov 1997 12:34:18 -0600 (CST)"
 "Message-Id: <199711061834.MAA11961@new-world.cs.rice.edu>"
 "From: Shriram Krishnamurthi <shriram@cs.rice.edu>"
 ....
 "Status: RO")
> (get-message/complete  c 100)
("Date: Thu, 6 Nov 1997 12:34:18 -0600 (CST)"
 "Message-Id: <199711061834.MAA11961@new-world.cs.rice.edu>"
 "From: Shriram Krishnamurthi <shriram@cs.rice.edu>"
 ....
 "Status: RO")
("some body" "text" "goes" "." "here" "." "")
> (get-unique-id/single c 205)
no message numbered 205 available for unique id
> (list-tail (get-unique-id/all c) 194)
((195 . "e24d13c7ef050000") (196 . "3ad2767070050000"))
> (get-unique-id/single c 196)
"3ad2767070050000"
> (disconnect-from-server c)
```

9.3 POP3 Unit

```
(require net/pop3-unit)
```

---

pop3@ : unit?

Imports nothing, exports pop3^.

9.4 POP3 Signature

```
(require net/pop3-sig)
```

---

pop3^ : signature

Includes everything exported by the net/pop3 module.

10 MIME: Decoding Internet Data

```
(require net/mime)
```

The net/mime library provides utilities for parsing and creating MIME encodings as described in RFC 2045 through RFC 2049.

The library was written by Francisco Solsona.

10.1 Message Decoding

```
(mime-analyze message-in part?) → message?
  message-in : (or/c bytes? input-port)
  part? : any/c
```

Parses *message-in* and returns the parsed result as a message instance.

```
(struct message (version entity fields))
  version : real?
  entity : entity
  fields : (listof string?)
```

A decoded MIME message. The version is 1.0 by default. The entity field represents the message data. The fields field contains one string for each field in the message header.

```
(struct entity (type
                subtype
                charset
                encoding
                disposition
                params
                id
                description
                other
                fields
                parts
                body))
  type : symbol?
  subtype : symbol?
  charset : symbol?
  encoding : symbol?
```

```
disposition : disposition?
params : (listof (cons/c symbol? string?))
id : string?
description : string?
other : (listof string?)
fields : (listof string?)
parts : (listof message?)
body : (output-port? . -> . void?)
```

Represents the content of a message or a sub-part.

Standard values for the `type` field include `'text`, `'image`, `'audio`, `'video`, `'application`, `'message`, and `'multipart`.

Standard values for the `subtype` field depend on the `type` field, and include the following:

`'text`	`'plain`	[RFC1521, NSB]
	`'richtext`	[RFC1521, NSB]
	`'tab-separated-values`	[Lindner]
`'multipart`	`'mixed`	[RFC1521, NSB]
	`'alternative`	[RFC1521, NSB]
	`'digest`	[RFC1521, NSB]
	`'parallel`	[RFC1521, NSB]
	`'appledouble`	[MacMime, Faltstrom]
	`'header-set`	[Crocker]
`'message`	`'rfc822`	[RFC1521, NSB]
	`'partial`	[RFC1521, NSB]
	`'external-body`	[RFC1521, NSB]
	`'news`	[RFC 1036, Spencer]
`'application`	`'octet-stream`	[RFC1521, NSB]
	`'postscript`	[RFC1521, NSB]
	`'oda`	[RFC1521, NSB]
	`'atomicmail`	[atomicmail, NSB]
	`'andrew-inset`	[andrew-inset, NSB]
	`'slate`	[slate, Crowley]
	`'wita`	[Wang Info Transfer, Campbell]
	`'dec-dx`	[Digital Doc Trans, Campbell]
	`'dca-rft`	[IBM Doc Content Arch, Campbell]
	`'activemessage`	[Shapiro]
	`'rtf`	[Lindner]
	`'applefile`	[MacMime, Faltstrom]
	`'mac-binhex40`	[MacMime, Faltstrom]
	`'news-message-id`	[RFC1036, Spencer]
	`'news-transmission`	[RFC1036, Spencer]
	`'wordperfect5.1`	[Lindner]
	`'pdf`	[Lindner]

	'zip	[Lindner]
	'macwriteii	[Lindner]
	'msword	[Lindner]
	'remote-printing	[RFC1486,MTR]
'image	'jpeg	[RFC1521, NSB]
	'gif	[RFC1521, NSB]
	'ief	[RFC1314]
	'tiff	[MTR]
'audio	'basic	[RFC1521, NSB]
'video	'mpeg	[RFC1521, NSB]
	'quicktime	[Lindner]

Standard values for the `charset` field include `'us-ascii`, which is the default.

Standard values for the `encoding` field are `'7bit`, `'8bit`, `'binary`, `'quoted-printable`, and `'base64`. The default is `'7bit`.

The `params` field contains a list of parameters from other MIME headers.

The `id` field is taken from the `"Content-Id"` header field.

The `description` field is taken from the `"Content-description"` header field.

The `other` field contains additional (non-standard) field headers whose field names start with `"Content-"`.

The `fields` field contains additional field headers whose field names *do not* start with `"Content-"`.

The `parts` contains sub-parts from multipart MIME messages. This list is non-empty only when `type` is `'multipart` or `'message`.

The `body` field represents the body as a function that consumes an output out and writes the decoded message to the port. No bytes are written if `type` is `'multipart` or `'message`. All of the standard values of `encoding` are supported. The procedure only works once (since the encoded body is pulled from a stream).

```
(struct disposition (type
                     filename
                     creation
                     modification
                     read
                     size
                     params))
  type : symbol?
  filename : (or/c string? false/c)
  creation : (or/c string? false/c)
```

```
  modification : (or/c string? false/c)
  read : (or/c string? false/c)
  size : (or/c exact-nonnegative-integer? false/c)
  params : (listof (cons/c symbol? string?))
```

Represents a "Content-Disposition" header as defined in RFC 2183.

Standard values for the type field include 'inline and 'attachment.

The filename field is drawn from the "filename" parameter of the "Content-Disposition" header, if included in the message.

The creation, modification, and read fields represent file timestamps as drawn from the "creation-date", "modification-date", and "read-date" attributes of the "Content-Disposition" header, if included in the message.

The size field is drawn from the "size" parameter of the "Content-Disposition" header, if included in the message.

The params field stores any additional attribute bindings of the "Content-Disposition" header, if included in the message.

10.2 Exceptions

```
(struct mime-error ())
```

The supertype of all MIME exceptions.

```
(struct (unexpected-termination mime-error) (msg))
  msg : string?
```

Raised when an end-of-file is reached while parsing the headers of a MIME entity. It usually means that the message does not conform to RFC 2045 and friends.

```
(struct (missing-multipart-boundary-parameter mime-error) ())
```

Raised when a multipart type is specified, but no "Boundary" parameter is given or an end-of-file is encountered before the boundary.

```
(struct (malformed-multipart-entity mime-error) (msg))
  msg : string?
```

Similar to unexpected-termination, but used only while scanning parts of a multipart message.

```
(struct (empty-mechanism mime-error) ())
```

Raised when no transport encoding mechanism was provided with the "Content-Transfer-Encoding" field.

```
(struct (empty-type mime-error) ())
```

Raised when no type is specified for "Content-Type", or when the specification is incorrectly formatted.

```
(struct (empty-subtype mime-error) ())
```

Raised when no sub-type is specified for "Content-Type", or when the specification is incorrectly formatted.

```
(struct (empty-disposition-type mime-error) ())
```

Raised when type specified for the "Content-Disposition" field, or when the specification is incorrectly formatted.

10.3 MIME Unit

```
(require net/mime-unit)
```

---

mime@ : unit?

Imports nothing, exports mime^.

10.4 MIME Signature

```
(require net/mime-sig)
```

---

mime^ : signature

Includes everything exported by the `net/mime` module.

11 Base 64: Encoding and Decoding

```
(require net/base64)
```

The net/base64 library provides utilities for Base 64 (mime-standard) encoding and decoding.

11.1 Functions

```
(base64-encode bstr) → bytes?
  bstr : bytes?
```

Consumes a byte string and returns its Base 64 encoding as a new byte string. The returned string is broken into 72-byte lines separated by CRLF combinations, and always ends with a CRLF combination unless the input is empty.

```
(base64-decode bstr) → bytes?
  bstr : bytes?
```

Consumes a byte string and returns its Base 64 decoding as a new byte string.

```
(base64-encode-stream in out [newline-bstr]) → void?
  in : input-port?
  out : output-port?
  newline-bstr : bytes? = #"\n"
```

Reads bytes from *in* and writes the encoded result to *out*, breaking the output into 72-character lines separated by *newline-bstr*, and ending with *newline-bstr* unless the input stream is empty. Note that the default *newline-bstr* is just #"\n", not #"\r\n". The procedure returns when it encounters an end-of-file from *in*.

```
(base64-decode-stream in out) → void?
  in : input-port?
  out : output-port?
```

Reads a Base 64 encoding from *in* and writes the decoded result to *out*. The procedure returns when it encounters an end-of-file or Base 64 terminator = from *in*.

11.2 Base64 Unit

```
(require net/base64-unit)
```

---

base64@ : unit?

Imports nothing, exports base64^.

11.3 Base64 Signature

```
(require net/base64-sig)
```

---

base64^ : signature

Includes everything exported by the net/base64 module.

12 Quoted-Printable: Encoding and Decoding

```
(require net/qp)
```

The net/qp library provides utilities for quoted-printable (mime-standard) encoding and decoding from RFC 2045 section 6.7.

The library was written by Francisco Solsona.

12.1 Functions

```
(qp-encode bstr) → bytes?
  bstr : bytes?
```

Consumes a byte string and returns its quoted printable representation as a new string. The encoded string uses #"\r\n" where necessary to create shorter lines.

```
(qp-decode bstr) → bytes?
  bstr : bytes?
```

Consumes a byte string and returns its un-quoted printable representation as a new string. Non-soft line breaks are preserved in whatever form they exist (CR, LR, or CRLF) in the input string.

```
(qp-encode-stream in out [newline-bstr]) → void?
  in : input-port?
  out : output-port?
  newline-bstr : bytes? = #"\n"
```

Reads characters from *in* and writes the quoted printable encoded result to *out*.

The *newline-bstr* argument is used for soft line-breaks (after =). Note that the default *newline-bstr* is just #"\n", not #"\r\n".

Other line breaks are preserved in whatever form they exist (CR, LR, or CRLF) in the input stream.

```
(qp-decode-stream in out) → void?
  in : input-port?
  out : output-port?
```

Reads characters from *in* and writes de-quoted-printable result to *out*. Non-soft line breaks are preserved in whatever form they exist (CR, LR, or CRLF) in the input stream.

12.2 Exceptions

```
(struct qp-error ())
(struct (qp-wrong-input qp-error) ())
(struct (qp-wrong-line-size qp-error) ())
```

None of these are used anymore, but the bindings are preserved for backward compatibility.

12.3 Quoted-Printable Unit

```
(require net/qp-unit)
```

---

qp@ : unit?

Imports nothing, exports qp^.

12.4 -Printable Signature

```
(require net/qp-sig)
```

---

qp^ : signature

Includes everything exported by the net/qp module.

13 DNS: Domain Name Service Queries

```
(require net/dns)
```

The net/dns library provides utilities for looking up hostnames.

Thanks to Eduardo Cavazos and Jason Crowe for repairs and improvements.

13.1 Functions

```
(dns-get-address nameserver address) → string?
  nameserver : string?
  address : string?
```

Consults the specified nameserver (normally a numerical address like "128.42.1.30") to obtain a numerical address for the given Internet address.

The query record sent to the DNS server includes the "recursive" bit, but dns-get-address also implements a recursive search itself in case the server does not provide this optional feature.

```
(dns-get-name nameserver address) → string?
  nameserver : string?
  address : string?
```

Consults the specified nameserver (normally a numerical address like "128.42.1.30") to obtain a name for the given numerical address.

```
(dns-get-mail-exchanger nameserver address) → string?
  nameserver : string?
  address : string?
```

Consults the specified nameserver to obtain the address for a mail exchanger the given mail host address. For example, the mail exchanger for "ollie.cs.rice.edu" might be "cs.rice.edu".

```
(dns-find-nameserver) → (or/c string? false/c)
```

Attempts to find the address of a nameserver on the present system. Under Unix, this procedure parses "/etc/resolv.conf" to extract the first nameserver address. Under Windows, it runs nslookup.exe.

13.2 DNS Unit

```
(require net/dns-unit)
```

---

dns@ : unit?

Imports nothing, exports dns^.

13.3 DNS Signature

```
(require net/dns-sig)
```

---

dns^ : signature

Includes everything exported by the net/dns module.

14 NNTP: Newsgroup Protocol

```
(require net/nntp)
```

The net/nntp module provides tools to access Usenet group via NNTP [RFC977].

14.1 Connection and Operations

```
(struct communicator (sender receiver server port))
  sender : output-port?
  receiver : input-port?
  server : string?
  port : (integer-in 0 65535)
```

Once a connection to a Usenet server has been established, its state is stored in a communicator, and other procedures take communicators as an argument.

```
(connect-to-server server [port-number]) → communicator?
  server : string?
  port-number : (integer-in 0 65535) = 119
```

Connects to *server* at *port-number*.

```
(disconnect-from-server communicator) → void?
  communicator : communicator?
```

Disconnects an NNTP communicator.

```
(open-news-group communicator newsgroup)
 → exact-nonnegative-integer?
   exact-nonnegative-integer?
   exact-nonnegative-integer?
  communicator : communicator?
  newsgroup : string?
```

Selects the newsgroup of an NNTP connection. The returned values are the total number of articles in the group, the first available article, and the last available article.

```
(authenticate-user communicator
                   username
                   password)    → void?
  communicator : communicator?
  username : string?
  password : string?
```

Tries to authenticate a user with the original authinfo command (uses cleartext). The *password* argument is ignored if the server does not ask for it.

```
(head-of-message communicator
                 message-index) → (listof string?)
  communicator : communicator?
  message-index : exact-nonnegative-integer?
```

Given a message number, returns its header lines.

```
(body-of-message communicator
                 message-index) → (listof string?)
  communicator : communicator?
  message-index : exact-nonnegative-integer?
```

Given a message number, returns the body of the message.

```
(newnews-since communicator message-index) → (listof string?)
  communicator : communicator?
  message-index : exact-nonnegative-integer?
```

Implements the NEWNEWS command (often disabled on servers).

```
((generic-message-command command
                          ok-code)
                          communicator
                          message-index) → (listof string?)
  command : string?
  ok-code : exact-integer?
  communicator : communicator?
  message-index : exact-nonnegative-integer?
```

Useful primitive for implementing head-of-message, body-of-message and other similar commands.

```
(make-desired-header tag-string) → regexp?
  tag-string : string?
```

Takes a header field's tag and returns a regexp to match the field

---

```
(extract-desired-headers header desireds) → (listof string?)
  header : (listof string?)
  desireds : (listof regexp?)
```

Given a list of header lines and of desired regexps, returns the header lines that match any of the *desireds*.

14.2 Exceptions

```
(struct (nntp exn) ())
```

The supertype of all NNTP exceptions.

```
(struct (unexpected-response nntp) (code text))
  code : exact-integer?
  text : string?
```

Raised whenever an unexpected response code is received. The `text` field holds the response text sent by the server.

```
(struct (bad-status-line nntp) (line))
  line : string?
```

Raised for mal-formed status lines.

```
(struct (premature-close nntp) (communicator))
  communicator : communicator?
```

Raised when a remote server closes its connection unexpectedly.

```
(struct (bad-newsgroup-line nntp) (line))
  line : string?
```

Raised when the newsgroup line is improperly formatted.

```
(struct (non-existent-group nntp) (group))
  group : string?
```

Raised when the server does not recognize the name of the requested group.

```
(struct (article-not-in-group nntp) (article))
  article : exact-integer?
```

Raised when an article is outside the server's range for that group.

```
(struct (no-group-selected nntp) ())
```

Raised when an article operation is used before a group has been selected.

```
(struct (article-not-found nntp) (article))
  article : exact-integer?
```

Raised when the server is unable to locate the article.

```
(struct (authentication-rejected nntp) ())
```

Raised when the server reject an authentication attempt.

14.3 NNTP Unit

```
(require net/nntp-unit)
```

```
nntp@ : unit?
```

Imports nothing, exports `nntp^`.

14.4 NNTP Signature

```
(require net/nntp-sig)
```

`nntp^` : signature

Includes everything exported by the `net/nntp` module.

15 TCP: Unit and Signature

The net/tcp-sig and net/tcp-unit libraries define a tcp^ signature and tcp@ implementation, where the implementation uses scheme/tcp.

Some units in the "net" collection import tcp^, so that they can be used with transports other than plain TCP. For example, url@ imports tcp^.

See also tcp-redirect and make-ssl-tcp@.

15.1 TCP Signature

```
(require net/tcp-sig)
```

---

tcp^ : signature

```
(tcp-listen  port-no
             [max-allow-wait
              reuse?
              hostname])      → tcp-listener?
  port-no : (and/c exact-nonnegative-integer?
                   (integer-in 1 65535))
  max-allow-wait : exact-nonnegative-integer? = 4
  reuse? : any/c = #f
  hostname : (or/c string? false/c) = #f
```

Like tcp-listen from scheme/tcp.

```
(tcp-connect  hostname
              port-no
              [local-hostname
               local-port-no]) → input-port? output-port?
  hostname : string?
  port-no : (and/c exact-nonnegative-integer?
                   (integer-in 1 65535))
  local-hostname : (or/c string? false/c) = #f
  local-port-no : (or/c (and/c exact-nonnegative-integer? = #f
                               (integer-in 1 65535))
                        false/c)
```

Like tcp-connect from scheme/tcp.

```
(tcp-connect/enable-break  hostname
                           port-no
                          [local-hostname]
                           local-port-no)
 → input-port? output-port?
  hostname : string?
  port-no : (and/c exact-nonnegative-integer?
                   (integer-in 1 65535))
  local-hostname : (or/c string? false/c) = #f
  local-port-no : (or/c (and/c exact-nonnegative-integer?
                               (integer-in 1 65535))
                        false/c)
```

Like tcp-connect/enable-break from scheme/tcp.

```
(tcp-accept listener) → input-port? output-port?
  listener : tcp-listener?
```

Like tcp-accept from scheme/tcp.

```
(tcp-accept/enable-break listener) → input-port? output-port?
  listener : tcp-listener?
```

Like tcp-accept/enable-break from scheme/tcp.

```
(tcp-accept-ready? listener) → boolean?
  listener : tcp-listener?
```

Like tcp-accept-ready? from scheme/tcp.

```
(tcp-close listener) → void?
  listener : tcp-listener?
```

Like tcp-close from scheme/tcp.

```
(tcp-listener? v) → boolean?
  v : any/c
```

Like tcp-listener? from scheme/tcp.

```
(tcp-abandon-port tcp-port) → void?
  tcp-port : port?
```

Like tcp-abandon-port from scheme/tcp.

```
(tcp-addresses tcp-port [port-numbers?])
 → (or/c (values string? string?)
         (values string? (integer-in 1 65535)
                 string? (integer-in 1 65535)))
  tcp-port : port?
  port-numbers? : any/c = #f
```

Like `tcp-addresses` from `scheme/tcp`.

15.2 TCP Unit

```
(require net/tcp-unit)
```

`tcp@ : unit?`

Imports nothing and exports `tcp^`, implemented using `scheme/tcp`.

16 TCP Redirect: `tcp^` via Channels

```
(require net/tcp-redirect)
```

The `net/tcp-redirect` library provides a function for directing some TCP port numbers to use buffered channels instead of the TCP support from `scheme/tcp`.

```
(tcp-redirect port-numbers) → unit?
  port-numbers : (listof (integer-in 0 65535))
```

Returns a unit that implements `tcp^`. For port numbers not listed in *port-numbers*, the unit's implementations are the `scheme/tcp` implementations.

For the port numbers listed in *port-numbers* and for connections to `"127.0.0.1"`, the unit's implementation does not use TCP connections, but instead uses internal buffered channels. Such channels behave exactly as TCP listeners and ports.

17 SSL Unit: tcp^ via SSL

```
(require net/ssl-tcp-unit)
```

The net/ssl-tcp-unit library provides a function for creating a tcp^ implementation with openssl functionality.

```
(make-ssl-tcp@ server-cert-file
               server-key-file
               server-root-cert-files
               server-suggest-auth-file
               client-cert-file
               client-key-file
               client-root-cert-files)  → unit?
  server-cert-file : (or/c path-string? false/c)
  server-key-file : (or/c path-string? false/c)
  server-root-cert-files : (or/c (listof path-string?) false/c)
  server-suggest-auth-file : path-string?
  client-cert-file : (or/c path-string? false/c)
  client-key-file : (or/c path-string? false/c)
  client-root-cert-files : (listof path-string?)
```

Returns a unit that implements tcp^ using the SSL functions from openssl. The arguments to make-ssl-tcp@ control the certificates and keys uses by server and client connections:

- *server-cert-file* — a PEM file for a server's certificate; #f means no certificate (which is unlikely to work with any SSL client)
- *server-key-file* — a private key PEM to go with *server-cert-file*; #f means no key (which is likely renders a certificate useless)
- *server-root-cert-files* — a list of PEM files for trusted root certificates; #f disables verification of peer client certificates
- *server-suggest-auth-file* — PEM file for root certificates to be suggested to peer clients that must supply certificates
- *client-cert-file* — a PEM file for a client's certificate; #f means no certificate (which is usually fine)
- *client-key-file* — a private key PEM to go with *client-cert-file*; #f means no key (which is likely renders a certificate useless)
- *client-root-cert-files* — a list of PEM files for trusted root certificates; #f disables verification of peer server certificates

18 CGI Scripts

```
(require net/cgi)
```

The net/cgi module provides tools for scripts that follow the Common Gateway Interface [CGI].

The net/cgi library expects to be run in a certain context as defined by the CGI standard. This means, for instance, that certain environment variables will be bound.

Unfortunately, not all CGI environments provide this. For instance, the FastCGI library, despite its name, does not bind the environment variables required of the standard. Users of FastCGI will need to bind REQUEST_METHOD and possibly also QUERY_STRING to successfully employ the CGI library. The FastCGI library ought to provide a way to extract the values bound to these variables; the user can then put these into the CGI program's environment using the putenv function.

A CGI *binding* is an association of a form item with its value. Some form items, such as checkboxes, may correspond to multiple bindings. A binding is a tag-string pair, where a tag is a symbol or a string.

18.1 CGI Functions

```
(get-bindings)
 → (listof (cons/c (or/c symbol? string?) string?))
(get-bindings/post)
 → (listof (cons/c (or/c symbol? string?) string?))
(get-bindings/get)
 → (listof (cons/c (or/c symbol? string?) string?))
```

Returns the bindings that corresponding to the options specified by the user. The get-bindings/post and get-bindings/get variants work only when POST and GET forms are used, respectively, while get-bindings determines the kind of form that was used and invokes the appropriate function.

```
(extract-bindings key? bindings) → (listof string?)
  key? : (or/c symbol? string?)
  bindings : (listof (cons/c (or/c symbol? string?) string?))
```

Given a key and a set of bindings, determines which ones correspond to a given key. There may be zero, one, or many associations for a given key.

```
(extract-binding/single key? bindings) → string?
  key? : (or/c symbol? string?)
  bindings : (listof (cons/c (or/c symbol? string?) string?))
```

Like extract-bindings, but for a key that has exactly one association.

```
(output-http-headers) → void?
```

Outputs all the HTTP headers needed for a normal response. Only call this function if you are not using generate-html-output or generate-error-output.

```
(generate-html-output  title
                       body
                      [text-color
                       bg-color
                       link-color
                       vlink-color
                       alink-color]) → void?
  title : string?
  body : (listof string?)
  text-color : string? = "#000000"
  bg-color : string? = "#ffffff"
  link-color : string? = "#cc2200"
  vlink-color : string? = "#882200"
  alink-color : string? = "#444444"
```

Outputs an response: a title and a list of strings for the body.

The last five arguments are each strings representing a HTML color; in order, they represent the color of the text, the background, un-visited links, visited links, and a link being selected.

```
(string->html str) → string?
  str : string?
```

Converts a string into an HTML string by applying the appropriate HTML quoting conventions.

```
(generate-link-text str html-str) → string?
  str : string?
  html-str : string?
```

Takes a string representing a URL, a HTML string for the anchor text, and generates HTML corresponding to an anchor.

```
(generate-error-output strs) → any
  strs : (listof string?)
```

The procedure takes a list of HTML strings representing the body, prints them with the subject line ”Internal error”, and exits via `exit`.

```
(get-cgi-method) → (one-of/c "GET" "POST")
```

Returns either `"GET"` or `"POST"` when invoked inside a CGI script, unpredictable otherwise.

```
(bindings-as-html listof) → (listof string?)
  listof : (cons/c (or/c symbol? string?) string?)
```

Converts a set of bindings into a list of HTML strings, which is useful for debugging.

```
(struct cgi-error ())
```

A supertype for all exceptions thrown by the `net/cgi` library.

```
(struct (incomplete-%-suffix cgi-error) (chars))
  chars : (listof char?)
```

Raised when a `%` in a query is followed by an incomplete suffix. The characters of the suffix—excluding the `%`—are provided by the exception.

```
(struct (invalid-%-suffix cgi-error) (char))
  char : char?
```

Raised when the character immediately following a `%` in a query is invalid.

18.2 CGI Unit

```
(require net/cgi-unit)
```

```
cgi@ : unit?
```

Imports nothing, exports `cgi^`.

18.3 CGI Signature

```
(require net/cgi-sig)
```

`cgi^` : signature

Includes everything exported by the `net/cgi` module.

19 Cookie: HTTP Client Storage

```
(require net/cookie)
```

The net/cookie library provides utilities for using cookies as specified in RFC 2109 [RFC2109].

19.1 Functions

```
(cookie? v) → boolean?
  v : any/c
```

Returns #t if *v* represents a cookie, #f otherwise.

```
(set-cookie name value) → cookie?
  name : string?
  value : string?
```

Creates a new cookie, with default values for required fields.

```
(cookie:add-comment cookie comment) → cookie?
  cookie : cookie?
  comment : string?
```

Modifies *cookie* with a comment, and also returns *cookie*.

```
(cookie:add-domain cookie domain) → cookie?
  cookie : cookie?
  domain : string?
```

Modifies *cookie* with a domain, and also returns *cookie*. The *domain* must match a prefix of the request URI.

```
(cookie:add-max-age cookie seconds) → cookie?
  cookie : cookie?
  seconds : exact-nonnegative-integer?
```

Modifies *cookie* with a maximum age, and also returns *cookie*. The *seconds* argument is number of seconds that a client should retain the cookie.

```
(cookie:add-path cookie path) → cookie?
  cookie : cookie?
  path : string?
```

Modifies *cookie* with a path, and also returns *cookie*.

```
(cookie:secure cookie secure) → cookie?
  cookie : cookie?
  secure : boolean?
```

Modifies *cookie* with a security flag, and also returns *cookie*.

```
(cookie:version cookie version) → cookie?
  cookie : cookie?
  version : exact-nonnegative-integer?
```

Modifies *cookie* with a version, and also returns *cookie*. The default is the only known incarnation of HTTP cookies: 1.

```
(print-cookie cookie) → string?
  cookie : cookie?
```

Prints *cookie* to a string. Empty fields do not appear in the output except when there is a required default.

```
(get-cookie name cookies) → (listof string?)
  name : string?
  cookies : string?
```

Returns a list with all the values (strings) associated with *name*.

The method used to obtain the "Cookie" header depends on the web server. It may be an environment variable (CGI), or you may have to read it from the input port (FastCGI), or maybe it comes in an initial-request structure, etc. The get-cookie and get-cookie/single procedure can be used to extract fields from a "Cookie" field value.

```
(get-cookie/single name cookies) → (or/c string? false/c)
  name : string?
  cookies : string?
```

Like get-cookie, but returns the just first value string associated to *name*, or #f if no

association is found.

```
(struct (cookie-error exn:fail) ())
```

Raised for errors when handling cookies.

19.2 Examples

19.2.1 Creating a cookie

```
(let ((c (cookie:add-max-age
          (cookie:add-path
           (set-cookie "foo" "bar")
           "/servlets")
          3600)))
  (print-cookie c))
```

Produces

```
"foo=bar; Max-Age=3600; Path=/servlets; Version=1"
```

To use this output in a “regular” CGI, instead of the last line use:

```
(display (format "Set-Cookie: ~a" (print-cookie c)))
```

and to use with the PLT Web Server, use:

```
(make-response/full code message (current-seconds) mime
                    `((Set-Cookie . ,(print-cookie c)))
                    body)
```

19.2.2 Parsing a cookie

Imagine your Cookie header looks like this:

```
> (define cookies
    "test2=2; test3=3; xfcTheme=theme6; xfcTheme=theme2")
```

Then, to get the values of the xfcTheme cookie, use

```
> (get-cookie "xfcTheme" cookies)
("theme6" "theme2")
> (get-cookie/single "xfcTheme" cookies)
```

```
"theme6"
```

If you try to get a cookie that simply is not there:

```
> (get-cookie/single "foo" cookies)
#f
> (get-cookie "foo" cookies)
()
```

Note that not having a cookie is normally not an error. Most clients won't have a cookie set then first arrive at your site.

19.3 Cookie Unit

```
(require net/cookie-unit)
```

`cookie@ : unit?`

Imports nothing, exports `cookie^`.

19.4 Cookie Signature

```
(require net/cookie-sig)
```

`cookie^ : signature`

Includes everything exported by the `net/cookie` module.

Bibliography

[CGI] "Common Gateway Interface (CGI/1.1)." http://hoohoo.ncsa.uiuc.edu/cgi/

[RFC822] David Crocker, "Standard for the Format of ARPA Internet Text Messages," RFC, 1982. http://www.ietf.org/rfc/rfc0822.txt

[RFC977] Brian Kantor and Phil Lapsley, "Network News Transfer Protocol," RFC, 1986. http://www.ietf.org/rfc/rfc0977.txt

[RFC1738] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," RFC, 1994. http://www.ietf.org/rfc/rfc1738.txt

[RFC1939] J. Myers and M. Rose, "Post Office Protocol - Version 3," RFC, 1996. http://www.ietf.org/rfc/rfc1939.txt

[RFC2060] M. Crispin, "Internet Message Access Protocol - Version 4rev1," RFC, 1996. http://www.ietf.org/rfc/rfc2060.txt

[RFC2109] D. Kristol and L. Montulli, "HTTP State Management Mechanism," RFC, 1997. http://www.ietf.org/rfc/rfc2109.txt

[RFC2396] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," RFC, 1998. http://www.ietf.org/rfc/rfc2396.txt

[RFC3986] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC, 2005. http://www.ietf.org/rfc/rfc3986.txt

Index