Net: Networking Libraries

Version 6.9

April 27, 2017

Contents

Index **96**

1 HTTP Client

```
(require net/http-client)
```
package: base

The net/http-client library provides utilities to use the HTTP protocol.

```
(http-conn? x) → boolean?
  x : any/c
```

Identifies an HTTP connection.

```
(http-conn-live? x) → boolean?
  x : any/c
```

Identifies an HTTP connection that is "live", i.e. one that is still connected to the server.

```
(http-conn-liveable? x) → boolean?
  x : any/c
```

Identifies an HTTP connection that can be made "live", i.e. one for which http-conn-send! is valid. Either the HTTP connection is already http-conn-live?, or it can auto-reconnect.

```
(http-conn) → http-conn?
```

Returns a fresh HTTP connection.

```
(http-conn-open! hc
                 host
                [#:ssl? ssl?
                 #:port port
                 #:auto-reconnect? auto-reconnect?]) → void?
  hc : http-conn?
  host : (or/c bytes? string?)
  ssl? : base-ssl?-tnl/c = #f
  port : (between/c 1 65535) = (if ssl? 443 80)
  auto-reconnect? : boolean? = #f
```

Uses *hc* to connect to *host* on port *port* using SSL if *ssl?* is not #f (using *ssl?* as an argument to ssl-connect to, for example, check certificates.) If *auto-reconnect?* is #t, then the HTTP connection is going to try to *auto-reconnect* for subsequent requests. I.e., if the connection is closed when performing http-conn-send! or http-conn-recv!, then http-conn-enliven! is going to be called on it.

If *hc* is live, the connection is closed.

```
(http-conn-open  host
                [#:ssl? ssl?
                 #:port port
                 #:auto-reconnect? auto-reconnect?]) → http-conn?
  host : (or/c bytes? string?)
  ssl? : base-ssl?-tnl/c = #f
  port : (between/c 1 65535) = (if ssl? 443 80)
  auto-reconnect? : boolean? = #f
```

Calls `http-conn-open!` with a fresh connection, which is returned.

```
(http-conn-close! hc) → void?
  hc : http-conn?
```

Closes *hc* if it is live.

```
(http-conn-abandon! hc) → void?
  hc : http-conn?
```

Closes the output side of *hc*, if it is live.

```
(http-conn-enliven! hc) → void?
  hc : http-conn?
```

Reconnects *hc* to the server, if it is *not* live but it is configured to auto-reconnect.

```
(http-conn-send!  hc
                  uri
                 [#:version version
                  #:method method
                  #:close? close?
                  #:headers headers
                  #:content-decode decodes
                  #:data data])            → void?
  hc : http-conn-liveable?
  uri : (or/c bytes? string?)
  version : (or/c bytes? string?) = #"1.1"
  method : (or/c bytes? string? symbol?) = #"GET"
  close? : boolean? = #f
  headers : (listof (or/c bytes? string?)) = empty
  decodes : (listof symbol?) = '(gzip)
  data : (or/c false/c bytes? string? data-procedure/c) = #f
```

Sends an HTTP request to *hc* to the URI *uri* using HTTP version *version*, the method *method*, and the additional headers given in *headers* and the additional data *data*. If

method is #"HEAD" (or "HEAD" or 'HEAD), provide the same *method* when calling http-conn-recv! to avoid attempting to receive content.

If *data* is a procedure, it will be called once with a procedure of one argument, which is a string or byte string to be written to the request body using chunked transfer encoding.

If *headers* does not contain an `Accept-Encoding` header, then a header indicating that encodings from *decodes* are accepted is automatically added.

If *close?* is #t and *headers* does not contain a `Connection` header, then a `Connection: close` header will be added.

This function does not support requests that expect `100 (Continue)` responses.

```
(http-conn-recv! hc
                 [#:content-decode decodes
                  #:method method
                  #:close? close?])
 → bytes? (listof bytes?) input-port?
  hc : http-conn-liveable?
  decodes : (listof symbol?) = '(gzip)
  method : (or/c bytes? string? symbol?) = #"GET"
  close? : boolean? = #f
```

Parses an HTTP response from *hc* for the method *method* while decoding the encodings listed in *decodes*.

Returns the status line, a list of headers, and an port which contains the contents of the response. The port's content must be consumed before the connection is used further.

If *close?* is #t, then the connection will be closed following the response parsing. If *close?* is #f, then the connection is only closed if the server instructs the client to do so.

Changed in version 6.1.1.6 of package `base`: Added the `#:method` argument.

```
(http-conn-sendrecv! hc
                     uri
                    [#:version version
                     #:method method
                     #:headers headers
                     #:data data
                     #:content-decode decodes
                     #:close? close?])
 → bytes? (listof bytes?) input-port?
  hc : http-conn-liveable?
  uri : (or/c bytes? string?)
```

```
  version : (or/c bytes? string?) = #"1.1"
  method : (or/c bytes? string? symbol?) = #"GET"
  headers : (listof (or/c bytes? string?)) = empty
  data : (or/c false/c bytes? string? data-procedure/c) = #f
  decodes : (listof symbol?) = '(gzip)
  close? : boolean? = #f
```

Calls `http-conn-send!` and `http-conn-recv!` in sequence.

```
(http-sendrecv  host
                uri
               [#:ssl? ssl?
                #:port port
                #:version version
                #:method method
                #:headers headers
                #:data data
                #:content-decode decodes])
 → bytes? (listof bytes?) input-port?
  host : (or/c bytes? string?)
  uri : (or/c bytes? string?)
  ssl? : base-ssl?-tnl/c = #f
  port : (between/c 1 65535) = (if ssl? 443 80)
  version : (or/c bytes? string?) = #"1.1"
  method : (or/c bytes? string? symbol?) = #"GET"
  headers : (listof (or/c bytes? string?)) = empty
  data : (or/c false/c bytes? string? data-procedure/c) = #f
  decodes : (listof symbol?) = '(gzip)
```

Calls `http-conn-send!` and `http-conn-recv!` in sequence on a fresh HTTP connection produced by `http-conn-open`.

The HTTP connection is not returned, so it is always closed after one response, which is why there is no `#:closed?` argument like `http-conn-recv!`.

```
(http-conn-CONNECT-tunnel  proxy-host
                           proxy-port
                           target-host
                           target-port
                          [#:ssl? ssl?]) → base-ssl?/c
                                           input-port?
                                           output-port?
                                           (-> port? void?)
  proxy-host : (or/c bytes? string?)
  proxy-port : (between/c 1 65535)
  target-host : (or/c bytes? string?)
```

```
target-port : (between/c 1 65535)
ssl? : base-ssl?/c = #f
```

Creates an HTTP connection to *proxy-host* (on port *proxy-port*) and invokes the HTTP "CONNECT" method to provide a tunnel to *target-host* (on port *target-port*).

The SSL context or symbol, if any, provided in *ssl?* is applied to the gateway ports using ports->ssl-ports (or ports->win32-ssl-ports).

The function returns four values:

- If *ssl?* was #f then #f. Otherwise an ssl-client-context? that has been negotiated with the target.

  If *ssl?* was a protocol symbol, then a new ssl-client-context? is created, otherwise the current value of *ssl?* is used
- An input-port? from the tunnelled service
- An output-port? to the tunnelled service
- An abandon function, which when applied either returned port, will abandon it, in a manner similar to tcp-abandon-port

The SSL context or symbol, if any, provided in *ssl?* is applied to the gateway ports using ports->ssl-ports (or ports->win32-ssl-ports) and the negotiated client context is returned

```
data-procedure/c : chaperone-contract?
```

Contract for a procedure that accepts a procedure of one argument, which is a string or byte string: (-> (-> (or/c bytes? string?) void?) any).

```
base-ssl?/c : contract?
```

Base contract for the definition of the SSL context (passed in ssl?) of an http-conn-CONNECT-tunnel:

```
(or/c boolean? ssl-client-context? symbol?).
```

If ssl? is not #f then ssl? is used as an argument to ssl-connect to, for example, check certificates.

```
base-ssl?-tnl/c : contract?
```

Contract for a base-ssl?/c that might have been applied to a tunnel. It is either a base-ssl?/c, or a base-ssl?/c consed onto a list of an input-port?, output-port?, and an abandon function (e.g. tcp-abandon-port):

```
(or/c base-ssl?/c (list/c base-ssl?/c input-port? output-port? (->
port? void?)))
```

1.1 Troubleshooting and Tips

1.1.1 How do I send properly formatted POST form requests?

You should send a `Content-Type` header with the value `application/x-www-form-urlencoded` and send the data formatted by `net/uri-codec`'s `form-urlencoded-encode` function. For example,

```
(http-conn-send!
   hc "/login"
   #:method "POST"
   #:data
   (alist->form-urlencoded
    (list (cons 'username "Ryu")
          (cons 'password "Sheng Long")))
   #:headers (list "Content-Type: application/x-www-form-
urlencoded"))
```

2 URLs and HTTP

```
(require net/url)
```
package: base

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", "https", and sometimes "file".

The net/url logs information and background-thread errors to a logger named 'net/url.

2.1 URL Structure

```
(require net/url-structs)
```
package: base

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

```
(struct url (scheme
             user
             host
             port
             path-absolute?
             path
             query
             fragment)
  #:extra-constructor-name make-url)
 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, which 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 Racket 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://racket-lang.org/a/")` has a `path` field with strings `"a"` and `""`, while the result of `(string->url "http://racket-lang.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)
    #:extra-constructor-name make-path/param)
  path : (or/c string? (or/c 'up 'same))
  param : (listof string?)
```

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

2.2 URL Parsing Functions

`(require net/url-string)` package: `base`

The functions used to convert strings and paths to from URL structure types and back again are provided by the `net/url-string` library, and re-exported by `net/url`.

```
url-regexp : regexp?
```

A regexp value that can be useful for matching url strings. Mostly follows RFC 3986 [RFC3986], Appendix B, except for using * instead of + for the scheme part (see `url`).

Added in version 6.4.0.7 of package `base`.

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

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.

The contract on *str* insists that, if the url has a scheme, then the scheme begins with a letter and consists only of letters, numbers, `+`, `-`, and `.` characters.

If *str* starts with `file:` (case-insensitively) and the value of the `file-url-path-convention-type` parameter is `'windows`, then special parsing rules apply to accommodate ill-formed but widely-recognized path encodings:

- If `file:` is followed by `//`, a letter, and `:`, then the `//` is stripped and the remainder parsed as a Windows path.
- If `file:` is followed by `\\`, then the `\\` is stripped and the remainder parsed as a Windows path.

In both of these cases, the host is `""`, the port is `#f`, and path-element decoding (which extract parameters or replaces `%20` with a space, for example) is not applied to the path.

Changed in version 6.3.0.1 of package `base`: Changed handling of `file:` URLs when the value of `file-url-path-convention-type` is `'windows`. Changed in version 6.4.0.7: Use more specific regexp for input contract. Changed in version 6.5.0.3: Support a host as an IPv6 literal address as written in `[...]`.

```
(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, and the result always starts `file://`. For a URL with a host, user, or port, its path must be either absolute or empty.

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 `&`.

The encoding of path segments and fragment is sensitive to the `current-url-encode-mode` parameter.

Changed in version 6.5.0.3 of package `base`: Support a host as an IPv6 literals addresses by writing the address in `[...]`.

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

Converts a path to a `url`.

With the `'unix` path convention, the host in the resulting URL is always `""`, and the path is absolute from the root.

With the `'windows` path convention and a UNC path, the machine part of the UNC root is used as the URL's host, and the drive part of the root is the first element of the URL's path.

Changed in version 6.3.0.1 of package `base`: Changed `'windows` encoding of UNC paths.

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

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

For the `'unix` path convention, the URL's host is ignored, and the URL's path is formed relative to the root.

For the `'windows` path convention:

- A non-`""` value for the URL's host field creates a UNC path, where the host is the UNC root's machine name, the URL's path must be non-empty, and the first element of the URL's path is used as the drive part of the UNC root.

- For legacy reasons, if the URL's host is `""`, the URL's path contains at least three elements, and and the first element of the URL's path is also `""`, then a UNC path is created by using the second and third elements of the path as the UNC root's machine and drive, respectively.
- Otherwise, the URL's path is converted to a Windows path. The result is an absolute path if the URL's first path element corresponds to a drive, otherwise the result is a relative path (even though URLs are not intended to represent relative paths).

Changed in version 6.3.0.1 of package `base`: Changed `'windows` treatment of a non-`""` host.

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

Converts *path* to a string that parses as a relative URL (with forward slashes). Each element of *path* is an element of the resulting URL path, and the string form of each element is encoded as needed. If *path* is syntactically a directory, then the resulting URL ends with `/`.

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

Determines the default conversion from strings for `"file"` URLs; see `string->url`.

```
(current-url-encode-mode) → (or/c 'recommended 'unreserved)
(current-url-encode-mode mode) → void?
  mode : (or/c 'recommended 'unreserved)
```

Determines how `url->string` encodes `!`, `*`, `'`, `(`, and `)` in path segments and fragments: `'recommended` leave them as-is, while `'unreserved` encodes them using `%`. The `'recommended` mode corresponds to the recommendations of RFC 2396 [RFC2396], but `'unreserved` avoids characters that are in some contexts mistaken for delimiters around URLs.

Internally, `'recommended` mode uses `uri-path-segment-encode` and `uri-encode`, while `'unreserved` mode uses `uri-path-segment-unreserved-encode` and `uri-unreserved-encode`.

2.3 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.

```
(get-pure-port  URL
               [header
                #:redirections redirections]) → input-port?
  URL : url?
  header : (listof string?) = null
  redirections : exact-nonnegative-integer? = 0
(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
(options-pure-port URL [header]) → input-port?
  URL : url?
  header : (listof string?) = null
```

Initiates a GET/HEAD/DELETE/OPTIONS 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*. If *redirections* is not `0`, then `get-pure-port` will follow redirections from the server, up to the limit given by *redirections*.

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*.

Beware: By default, `"https"` scheme handling does not verify a server's certificate (i.e., it's equivalent of clicking through a browser's warnings), so communication is safe, but the identity of the server is not verified. To validate the server's certificate, set `current-https-protocol` to `'secure` or a context created with `ssl-secure-client-context`.

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.

Changed in version 6.1.1.8 of package `base`: Added `options-pure-port`.

```
(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
(options-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.

Changed in version 6.1.1.8 of package base: Added options-impure-port.

```
(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.

Beware: See get-pure-port for warnings about "https" certificate validation.

```
(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)"
```

```
(get-pure-port/headers  url
                       [headers
                        #:redirections redirections
                        #:status? status?]
                        #:connection connection)
 → input-port? string?
  url : url?
  headers : (listof string?) = '()
  redirections : exact-nonnegative-integer? = 0
  status? : boolean? = #f
  connection : (or/c #f http-connection?)
```

This function is an alternative to calling `get-impure-port` and `purify-port` when needing to follow redirections. It also supports HTTP/1.1 connections, which are used when the *`connection`* argument is not `#f`.

The `get-pure-port/headers` function performs a GET request on *`url`*, follows up to *`redirections`* redirections and returns a port containing the data as well as the headers for the final connection. If *`status?`* is true, then the status line is included in the result string.

A given *`connection`* should be used for communication with a particular HTTP/1.1 server, unless *`connection`* is closed (via `http-connection-close`) between uses for different servers. If *`connection`* is provided, read all data from the result port before making a new request with the same *`connection`*. (Reusing a *`connection`* without reading all data may or may not work.)

```
(http-connection? v) → boolean?
  v : any/c
(make-http-connection) → http-connection?
(http-connection-close connection) → void?
  connection : http-connection?
```

A HTTP connection value represents a potentially persistent connection with a HTTP/1.1 server for use with `get-pure-port/headers`.

The `make-http-connection` creates a "connection" that is initially unconnected. Each call to `get-pure-port/headers` leaves a connection either connected or unconnected, depending on whether the server allows the connection to continue. The `http-connection-close` function unconnects, but it does not prevent further use of the connection value.

```
(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 interrupted 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)))
proxiable-url-schemes : (listof string?)
 = '("http" "https" "git")
```

The `current-proxy-servers` parameter determines a mapping of proxy servers used for connections. Each mapping is a list of three elements:

- the URL scheme, such as `"http"`, where `proxiable-url-schemes` lists the URL schemes that can be proxied
- the proxy server address; and
- the proxy server port number.

The initial value of `current-proxy-servers` is configured on demand from environment variables. Proxies for each URL scheme are configured from two variables each:

- `plt_http_proxy` and `http_proxy`, configure the HTTP proxy, where the former takes precedence over the latter. HTTP requests will be proxied using an HTTP proxy server connection
- `plt_https_proxy` and `https_proxy`, configure the HTTPS proxy, where the former takes precedence over the latter. HTTPS connections are proxied using an HTTP "CONNECT" tunnel
- `plt_git_proxy` and `git_proxy`, configure the GIT proxy, where the former takes precedence over the latter. GIT connections are proxied using an HTTP "CONNECT" tunnel

Each environment variable contains a single URL of the form `http://`⟨*hostname*⟩`:`⟨*portno*⟩. If any other components of the URL are provided, a warning will be logged to a `net/url` logger.

The default mapping is the empty list (i.e., no proxies).

```
(current-no-proxy-servers) → (listof (or/c string? regexp?))
(current-no-proxy-servers dest-hosts-list) → void?
  dest-hosts-list : (listof (or/c string? regexp?))
```

A parameter that determines which servers will be accessed directly i.e. without resort to `current-proxy-servers`. It is a list of

- strings that match host names exactly; and
- regexps that match host by pattern.

If a proxy server is defined for a URL scheme, then the destination host name is checked against `current-no-proxy-servers`. The proxy is used if and only if the host name does not match (by the definition above) any in the list.

The initial value of `current-no-proxy-servers` is configured on demand from the environment variables `plt_no_proxy` and `no_proxy`, where the former takes precedence over the latter. Each environment variable's value is parsed as a comma-separated list of "patterns," where a pattern is one of:

- a string beginning with a `.` (period): converted to a regexp that performs a suffix match on a destination host name; e.g. `.racket-lang.org` matches destinations of `doc.racket-lang.org`, `pkgs.racket-lang.org`, but neither `doc.bracket-lang.org` nor `pkgs.racket-lang.org.uk`;
- any other string: converted to a regexp that matches the string exactly.

This parsing is consistent with the `no_proxy` environment variable used by other software, albeit not consistent with the regexps stored in `current-no-proxy-servers`.

```
(proxy-server-for url-schm [dest-host-name])
 → (or/c (list/c string? string? (integer-in 0 65535)) #f)
  url-schm : string?
  dest-host-name : (or/c false/c string?) = #f
```

Returns the proxy server entry for the combination of *url-schm* and `host`, or `#f` if no proxy is to be used.

```
(url-exception? x) → boolean?
  x : any/c
```

Identifies an error thrown by URL functions.

```
(http-sendrecv/url  u
                    [#:method method
                     #:headers headers
                     #:data data
                     #:content-decode decodes])
 → bytes? (listof bytes?) input-port?
  u : url?
  method : (or/c bytes? string? symbol?) = #"GET"
  headers : (listof (or/c bytes? string?)) = empty
  data : (or/c false/c bytes? string? data-procedure/c) = #f
  decodes : (listof symbol?) = '(gzip)
```

Calls `http-sendrecv` using *u* to populate the host, URI, port, and SSL parameters.

This function does not support proxies.

```
(tcp-or-tunnel-connect scheme host port)
 → input-port? output-port?
  scheme : string?
  host : string?
  port : (between/c 1 65535)
```

If `(proxy-server-for` *scheme host*`)`, then the proxy is used to `http-conn-CONNECT-tunnel` to *host* (on port *port*).

Otherwise the call is equivalent to `(tcp-connect` *host port*`)`.

2.4 URL HTTPS mode

`(require net/url-connect)` package: `base`

These bindings are provided by the `net/url-connect` library, and used by `net/url`.

```
(current-https-protocol) → (or/c ssl-client-context? symbol?)
(current-https-protocol protocol) → void?
  protocol : (or/c ssl-client-context? symbol?)
```

A parameter that determines the connection mode for `"https"` connections; the parameter value is passed as the third argument to `ssl-connect` when creating an `"https"` connection. Set this parameter to validate a server's certificates, for example, as described with `get-pure-port`.

2.5 URL Unit

`(require net/url-unit)` package: `compatibility-lib`

`url@` deprecated

`url@`, `url^`, and `url+scheme^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/url` module.

```
url@ : unit?
```

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

The `url+scheme^` signature contains `current-connect-scheme`, which `url@` binds to a parameter. The parameter is set to the scheme of a URL when `tcp-connect` is called to create a connection. A `tcp-connect` variant linked to `url@` can check this parameter to choose the connection mode; in particular, `net/url` supplies a `tcp-connect` that actually uses `ssl-connect` when `(current-connect-scheme)` produces `"https"`.

Note that `net/url` does not provide the `current-connect-scheme` parameter.

2.6 URL Signature

`(require net/url-sig)` package: `compatibility-lib`

```
url^ : signature
```

Includes everything exported by the `net/url` module except `current-https-protocol` and `current-url-encode-mode`. Note that the exports of `net/url` and the `url^` signature do not include `current-connect-scheme`.

```
url+scheme^ : signature
```

Adds `current-connect-scheme` to `url^`.

3 URI Codec: Encoding and Decoding URIs

```
(require net/uri-codec)
```
package: base

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 library provides "unreserved" encoders that encode !, *, ', (, and) using their hex representation, which is not recommended by RFC 2396 but avoids problems with some contexts.

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 application/x-www-form-urlencoded 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 addition, since there appear to be some broken 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.

3.1 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.

```
(uri-path-segment-encode str) → string?
  str : string?
```

Encodes a string according to the rules in [RFC3986] for path segments.

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

Decodes a string according to the rules in [RFC3986] for path segments.

```
(uri-userinfo-encode str) → string?
  str : string?
```

Encodes a string according to the rules in [RFC3986] for the userinfo field.

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

Decodes a string according to the rules in [RFC3986] for the userinfo field.

```
(uri-unreserved-encode str) → string?
  str : string?
```

Encodes a string according to the rules in [RFC3986](section 2.3) for the unreserved characters.

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

Decodes a string according to the rules in [RFC3986](section 2.3) for the unreserved characters.

```
(uri-path-segment-unreserved-encode str) → string?
  str : string?
```

Encodes a string according to the rules in [RFC3986] for path segments, but also encodes characters that `uri-unreserved-encode` encodes and that `uri-encode` does not.

```
(uri-path-segment-unreserved-decode str) → string?
  str : string?
```

Decodes a string according to the rules in [RFC3986] for path segments.

```
(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-decode 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 `'semi-or-amp` mode is similar, but `;` is used when encoding. The other modes use/recognize only one 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"
```

3.2 URI Codec Unit

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

package: `compatibility-lib`

`uri-codec@` and `uri-codec^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/uri-codec` module.

```
uri-codec@ : unit?
```

Imports nothing, exports `uri-codec^`.

3.3 URI Codec Signature

```
(require net/uri-codec-sig)
```
package: `compatibility-lib`

```
uri-codec^ : signature
```

Includes everything exported by the `net/uri-codec` module except `uri-path-segment-unreserved-encode` and `uri-path-segment-unreserved-decode`.

4 FTP: Client

```
(require net/ftp)
```
package: net-lib

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

4.1 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 port-np argument usually should be 21.

```
(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; it must not contain a newline.

```
(ftp-directory-list ftp-conn [path])
 → (listof (list/c (one-of/c "-" "d" "l")
                   string?
                   string?))
  ftp-conn : ftp-connection?
  path : (or/c false/c string?) = #f
```

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. If a *path* argument is given, use it instead of the current directory.

Each file or directory is represented by a list of three or four 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`. The third string is the name of the file or directory. If the item is a file (the first string is `"-"`), and if the line that the server replied with has a size in the expected place, then a fourth string containing this size is included.

Warning: the FTP protocol has no specification for the reply format, so this information can be unreliable.

```
(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`).

Warning: the FTP protocol has no specification for the reply format, so this information can be unreliable.

```
(ftp-download-file  ftp-conn
                    local-dir
                    file
                   [#:progress progress-proc]) → void?
  ftp-conn : ftp-connection?
  local-dir : path-string?
  file : string?
  progress-proc : (or/c #f
                        (-> (-> (values exact-nonnegative-integer?
                                        evt?))
                            any))
                  = #f
```

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 in *local-dir*, then moved into place on success).

If *progress-proc* is not `#f`, then it is called with a function *get-count* that returns two values: the number of bytes transferred so far, and an event that becomes ready when the transferred-bye count changes. The *get-count* function can be called in any thread and any number of times. The *progress-proc* function should return immediately, perhaps starting

a thread that periodically polls *get-count*. Do not poll too frequently, or else polling will slow the transfer; the second argument from *get-count* is intended to limit polling.

```
(ftp-download-file
 ftp-conn "." "testfile"
 #:progress
 (lambda (get-count)
   (thread
    (lambda ()
      (let loop ()
        (define-values (count changed-evt) (get-count))
        (printf "~a bytes downloaded\n" count)
        (sync changed-evt)
        (loop))))))
```

```
(ftp-upload-file  ftp-conn
                  file-path
                 [#:progress progress-proc]) → void?
  ftp-conn : ftp-connection?
  file-path : path-string?
  progress-proc :  (or/c #f
                         (-> (-> (values exact-nonnegative-integer?
                                         evt?))
                             any))
                 = #f
```

Upload *file-path* to the server's current directory using the same name. If the file already exists in the local directory, it is replaced. The *progress-proc* argument is used in the same way as in ftp-download-file, but to report uploaded bytes instead of downloaded bytes.

```
(ftp-delete-file ftp-conn file-path) → void?
  ftp-conn : ftp-connection?
  file-path : path-string?
```

Delete the remote file use the *file-path* on the server.

```
(ftp-make-directory ftp-conn dir-name) → void?
  ftp-conn : ftp-connection?
  dir-name : string?
```

Make remote directory use the *dir-name*.

```
(ftp-delete-directory ftp-conn dir-name) → void?
  ftp-conn : ftp-connection?
  dir-name : string?
```

Delete remote directory use the *dir-name*.

```
(ftp-rename-file ftp-conn origin dest) → void?
  ftp-conn : ftp-connection?
  origin : string?
  dest : string?
```

Rename remote file name from *origin* to *dest*.

4.2 FTP Unit

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

package: compatibility-lib

```
ftp@ : unit?
```

Imports nothing, exports `ftp^`.

`ftp@` and `ftp^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/ftp` module.

4.3 FTP Signature

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

package: compatibility-lib

```
ftp^ : signature
```

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

5 Send URL: Opening a Web Browser

```
(require net/sendurl)
```
package: net-lib

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

See also browser/external, which requires racket/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.

On 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.)

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

On Unix, send-url uses a user-preference, or when none is set, it will look for a known browser. See the description of external-browser for details.

If *escape?* is true, then *str* is escaped (by UTF-8 encoding followed by "%" encoding) to avoid dangerous shell characters: single quotes, double quotes, backquotes, dollar signs, backslashes, non-ASCII characters, and non-graphic characters. Note that escaping does not affect already-encoded characters in *str*.

On all platforms, external-browser parameter can be set to a procedure to override the above behavior — the procedure will be called with the url string.

```
(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 (with #:escape? #t), but accepts a path to a file to be displayed by

the browser, along with optional *fragment* (with no leading #) and *query* (with no leading ?) strings. Use send-url/file to display a local file, since it takes care of the peculiarities of constructing the correct file:// URL.

The *path*, *fragment*, and *query* arguments are all encoded in the same way as a path provided to send-url, which means that already-encoded characters are used as-is.

```
(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 displays it from a temporary file.

When send-url/content is called, it scans old generated files (this happens randomly, not on every call) and removes them to avoid cluttering the temporary directory. If the #:delete-at argument is a number, then the temporary file is more eagerly removed after the specified number of seconds; the deletion happens in a thread, so if Racket exits earlier, the deletion will not happen. If the #:delete-at argument is #f, no eager deletion happens, but old temporary files are still deleted as described above.

```
(send-url/mac url [#:browser browser]) → void?
  url : string?
  browser : (or/c string? #f) = #f
```

Like send-url, but only for use on a Mac OS machine.

The optional *browser* argument, if present, should be the name of a browser installed on the system. For example,

```
(send-url/mac "http://www.google.com/" #:browser "Firefox")
```

would open the url in Firefox, even if that's not the default browser. Passing #f means to use the default browser.

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

A parameter that can hold a procedure to override how a browser is started, or #f to use the default platform-dependent command.

On Unix, the command that is used depends on the 'external-browser preference. If the preference is unset, send-url uses the first of the browsers from unix-browser-list for which the executable is found. Otherwise, the preference should hold a symbol indicating a known browser (from the unix-browser-list), or it a pair of a prefix and a suffix string that are concatenated around the url string to make up a shell command to run. In addition, the external-browser paremeter can be set to one of these values, and send-url will use it instead of the preference value.

Note that the URL is encoded to make it work inside shell double-quotes: URLs can still hold characters like #, ?, and &, so if the external-browser is set to a pair of prefix/suffix strings, they should use double quotes around the url.

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.

6 SMTP: Sending E-Mail

```
(require net/smtp)
```
package: net-lib

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.

6.1 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).

6.2 SMTP Unit

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

package: compatibility-lib

smtp@ : unit?

Imports nothing, exports smtp^.

smtp@ and smtp^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/smtp module.

6.3 SMTP Signature

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

package: compatibility-lib

smtp^ : signature

Includes everything exported by the net/smtp module.

7 `sendmail`: Sending E-Mail

```
(require net/sendmail)
```
package: `net-lib`

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.

7.1 Sendmail Functions

```
(send-mail-message/port from
                        subject
                        to
                        cc
                        bcc
                        extra-header ...) → output-port?
  from : (or/c string? false/c)
  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. All of these are quoted if they contain non-ASCII characters.

Note that passing already-quoted strings would be fine, since then there are no non-ASCII characters.

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. If it is `#f`, no "From:" header is generated, which usually means that your sendmail program will fill in the right value based on the user.

```
(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.

7.2 Sendmail Unit

`(require net/sendmail-unit)` package: `compatibility-lib`

```
sendmail@ : unit?
```

Imports nothing, exports `sendmail^`.

`sendmail@` and `sendmail^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/sendmail` module.

7.3 Sendmail Signature

`(require net/sendmail-sig)` package: `compatibility-lib`

```
sendmail^ : signature
```

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

8 Headers: Parsing and Constructing

```
(require net/head)
```
package: base

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).

8.1 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.

Example:

```
> (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.

```
(replace-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 string?)
  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.

Example:

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

8.2 Header Unit

(require net/head-unit) package: compatibility-lib

head@ : unit?

Imports nothing, exports head^.

head@ and head^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/head module.

8.3 Header Signature

(require net/head-sig) package: compatibility-lib

head^ : signature

Includes everything exported by the net/head module.

9 Header Field Encoding

```
(require net/unihead)
```
package: net-lib

The net/unihead module provides utilities for encoding and decoding header fields using the =?⟨*encoding*⟩?⟨*transport*⟩?...?= format.

```
(encode-for-header s) → string?
  s : string?
```

Encodes *s* for use in a header.

If *s* contains only ASCII characters, then the result string will have the same content as the given string. If *s* contains only Latin-1 characters, then on each CRLF-delimited line, the space-delimited sequence containing all non-ASCII characters in *s* is encoded with a =?ISO-8859-1?Q?...?= sequence. If *s* contains non-Latin-1 characters, then on each CRLF-delimited line, a space-delimited sequence containing all non-ASCII characters in *s* is encoded with a =?UTF-8?B?...?= sequence.

Examples:

```
> (encode-for-header "English")
"English"
> (encode-for-header "français")
"=?ISO-8859-1?Q?fran=E7ais?="
> (encode-for-header "→")
"=?UTF-8?B?4oaS?="
> (encode-for-header "→\r\nboth → and français here")
"=?UTF-8?B?4oaS?=\r\nboth =?UTF-8?B?4oaSIGFuZCBmcmFuw6dhaXM=?=
here"
```

```
(decode-for-header s) → string?
  s : string?
```

Decodes header fields that use the =?⟨*encoding*⟩?⟨*transport*⟩?...?= encoding format. The specified ⟨*encoding*⟩ is generalized via generalize-encoding before decoding content.

Examples:

```
> (decode-for-header "English")
"English"
> (decode-for-header "=?UTF-8?B?4oaS?= =?ISO-8859-
1?Q?fran=E7ais?=")
"→ français"
```

```
(generalize-encoding s) → (or string? bytes?)
  s : (or string? bytes?)
```

Generalizes the encoding name `s` to compensate for typical mailer bugs: Latin-1 and ASCII encodings are geenralized to WINDOWS-1252; GB and GB2312 are generalized to GBK; and KS_C_5601-1987 is generalized to CP949.

10 IMAP: Reading Mail

```
(require net/imap)
```
package: net-lib

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

10.1 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
              [#:tls? tls?
               #:try-tls? try-tls?]) → 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?)
  tls? : any/c = #f
  try-tls? : any/c = #t
```

Establishes an IMAP connection to the given server using the given username and password, and selects the specified mailbox. If *tls?* is true, a TLS connection is made to the server before communicating using the IMAP protocol. If *tls?* is #f but *try-tls?* is true, then after the IMAP connection is initially established, the connection is switched to a TLS connection if the server supports it.

The first result value represents 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
               [#:tls? tls?
                #:try-tls? try-tls?])
 → 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?)
  tls? : any/c = #f
  try-tls? : any/c = #t
```

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.

10.2 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? #f)
  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? #f)
  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? #f)
  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.

10.3 Manipulating Messages

```
(imap-get-messages imap msg-nums fields) → (listof list?)
  imap : imap-connection?
  msg-nums : (listof exact-nonnegative-integer?)
  fields : (listof (or/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`.

Example:

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

```
(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 Racket 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 : (or/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.

10.4 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 [flags]) → void?
  imap : imap-connection?
  mailbox : string?
  message : (or/c string? bytes?)
  flags : (listof (or/c 'seen 'answered 'flagged  = '(seen)
                        'deleted 'draft 'recent))
```

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? #f)
  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`.

10.5 IMAP Unit

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

package: `compatibility-lib`

```
imap@ : unit?
```

Imports nothing, exports `imap^`.

`imap@` and `imap^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/imap` module.

10.6 IMAP Signature

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

package: `compatibility-lib`

```
imap^ : signature
```

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

11 POP3: Reading Mail

```
(require net/pop3)
```
package: net-lib

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

```
(struct communicator (sender receiver server port state)
    #:extra-constructor-name make-communicator)
  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*.

11.1 Exceptions

```
(struct pop3 exn ()
    #:extra-constructor-name make-pop3)
```

The supertype of all POP3 exceptions.

```
(struct cannot-connect pop3 ()
    #:extra-constructor-name make-cannot-connect)
```

Raised when a connection to a server cannot be established.

```
(struct username-rejected pop3 ()
    #:extra-constructor-name make-username-rejected)
```

Raised if the username is rejected.

```
(struct password-rejected pop3 ()
    #:extra-constructor-name make-password-rejected)
```

Raised if the password is rejected.

```
(struct not-ready-for-transaction pop3 (communicator)
    #:extra-constructor-name make-not-ready-for-transaction)
  communicator : communicator?
```

Raised when the communicator is not in transaction mode.

```
(struct not-given-headers pop3 (communicator message)
    #:extra-constructor-name make-not-given-headers)
  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)
    #:extra-constructor-name make-illegal-message-number)
  communicator : communicator?
  message : exact-integer?
```

Raised when the client specifies an illegal message number.

```
(struct cannot-delete-message exn (communicator message)
    #:extra-constructor-name make-cannot-delete-message)
  communicator : communicator?
  message : exact-integer?
```

Raised when the server is unable to delete a message.

```
(struct disconnect-not-quiet pop3 (communicator)
    #:extra-constructor-name make-disconnect-not-quiet)
  communicator : communicator?
```

Raised when the server does not gracefully disconnect.

```
(struct malformed-server-response pop3 (communicator)
    #:extra-constructor-name make-malformed-server-response)
  communicator : communicator?
```

Raised when the server produces a malformed response.

11.2 Example Session

```
> (require net/pop3)
> (define c (connect-to-server "foo.bar.com"))
> (authenticate/plain-text "bob" "********" 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@foo.bar.com>"
 "From: Alice <alice@foo.bar.com>"
 ....
 "Status: RO")
> (get-message/complete  c 100)
```

```
("Date: Thu, 6 Nov 1997 12:34:18 -0600 (CST)"
 "Message-Id: <199711061834.MAA11961@foo.bar.com>"
 "From: Alice <alice@foo.bar.com>"
 ....
 "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)
```

11.3 POP3 Unit

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

package: compatibility-lib

pop3@ : unit?

pop3@ and pop3^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/pop3 module.

Imports nothing, exports pop3^.

11.4 POP3 Signature

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

package: compatibility-lib

pop3^ : signature

Includes everything exported by the net/pop3 module.

12 MIME: Decoding Internet Data

```
(require net/mime)
```
package: net-lib

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.

12.1 Message Decoding

```
(mime-analyze message-in [part?]) → message?
  message-in : (or/c bytes? input-port)
  part? : any/c = #f
```

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

If *part?* is #f, then *message-in* should start with the header for a full message; otherwise, *message-in* should start with the header for a part within a message.

```
(struct message (version entity fields)
    #:extra-constructor-name make-message)
  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)
    #:extra-constructor-name make-entity)
  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 : (or/c (output-port? . -> . void?) null?)
```

Represents the content of a message or a sub-part. The `mime-analyze` function chooses default values for fields when they are not specified in input.

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, but any `subtype` is allowed as a downcased version of the specification from the header.

Please note that RFC 3232 specifies that this list (taken from RFC 1700) is out-of-date, and that the IANA maintains a complete list, currently available at http://www.iana.org/assignments/media-types/media-types.xhtml

`'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. If `type` is `'multipart` or `'message`., then `body` is `'()`. 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)
    #:extra-constructor-name make-disposition)
  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.

12.2 Exceptions

```
(struct mime-error exn:fail ()
    #:extra-constructor-name make-mime-error)
```

The supertype of all MIME exceptions. Only the subtype missing-multipart-boundary-parameter is ever actually raised.

```
(struct unexpected-termination mime-error (msg)
    #:extra-constructor-name make-unexpected-termination)
  msg : string?
```

Originally raised when an end-of-file is reached while parsing the headers of a MIME entity, but currently a mere warning is logged.

```
(struct missing-multipart-boundary-parameter mime-error ()
    #:extra-constructor-name
    make-missing-multipart-boundary-parameter)
```

Raised when a multipart type is specified, but no "Boundary" parameter is given.

```
(struct malformed-multipart-entity mime-error (msg)
    #:extra-constructor-name make-malformed-multipart-entity)
  msg : string?
```

Never actually raised.

```
(struct empty-mechanism mime-error ()
    #:extra-constructor-name make-empty-mechanism)
```

Never actually raised.

```
(struct empty-type mime-error ()
    #:extra-constructor-name make-empty-type)
```

Never actually raised.

```
(struct empty-subtype mime-error ()
    #:extra-constructor-name make-empty-subtype)
```

Never actually raised.

```
(struct empty-disposition-type mime-error ()
    #:extra-constructor-name make-empty-disposition-type)
```

Never actually raised.

12.3 MIME Unit

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

package: compatibility-lib

```
mime@ : unit?
```

Imports nothing, exports `mime^`.

`mime@` and `mime^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/mime` module.

12.4 MIME Signature

```
(require net/mime-sig)
```
package: compatibility-lib

mime^ : signature

Includes everything exported by the net/mime module.

13 Base 64: Encoding and Decoding

```
(require net/base64)
```
package: base

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

13.1 Functions

```
(base64-encode bstr [newline-bstr]) → bytes?
  bstr : bytes?
  newline-bstr : bytes? = #"\r\n"
```

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 *newline-bstr*, which defaults to a CRLF combination, and the result always ends with a *newline-bstr* 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*.

13.2 Base64 Unit

```
(require net/base64-unit)
```
package: compatibility-lib

base64@ and base64^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/base64 module.

base64@ : unit?

Imports nothing, exports base64^.

13.3 Base64 Signature

(require net/base64-sig) package: compatibility-lib

base64^ : signature

Includes everything exported by the net/base64 module.

14 Quoted-Printable: Encoding and Decoding

```
(require net/qp)
```
package: net-lib

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.

14.1 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.

14.2 Exceptions

```
(struct qp-error ()
    #:extra-constructor-name make-qp-error)
(struct qp-wrong-input qp-error ()
    #:extra-constructor-name make-qp-wrong-input)
(struct qp-wrong-line-size qp-error ()
    #:extra-constructor-name make-qp-wrong-line-size)
```

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

14.3 Quoted-Printable Unit

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

package: compatibility-lib

```
qp@ : unit?
```

qp@ and qp^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/qp module.

Imports nothing, exports qp^.

14.4 -Printable Signature

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

package: compatibility-lib

```
qp^ : signature
```

Includes everything exported by the net/qp module.

15 DNS: Domain Name Service Queries

```
(require net/dns)
```
package: net-lib

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

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

15.1 Functions

```
(dns-get-address nameserver
                 address
                 [#:ipv6? ipv6?]) → string?
  nameserver : string?
  address : string?
  ipv6? : any/c = #f
```

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.

If *ipv6?* is a true value, then the numerical address that is returned will be an IPv6 address. If no AAAA record exists, an error will be raised.

```
(dns-get-srv nameserver name service [proto]) → (listof srv-rr?)
  nameserver : string?
  name : string?
  service : string?
  proto : string? = "tcp"
(struct srv-rr (priority weight port target)
    #:prefab)
  priority : (integer-in 0 65535)
  weight : (integer-in 0 65535)
  port : (integer-in 0 65535)
  target : string?
```

An SRV record is a particular kind of DNS resource record that maps an abstract service name onto a hostname and port combination. For more information, see the Wikipedia page on SRV records.

Consults the specified nameserver (normally a numerical address like "128.42.1.30") to retrieve the SRV records corresponding to the given name, service, and protocol. Returns a list of srv-rr structs if any corresponding SRV records are found; otherwise, returns '().

If *service* is "X", *proto* is "Y", and *name* is "example.com", then this will retrieve any SRV records at the domain name _X._Y.example.com.

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

Examples:

```
> (dns-get-srv (dns-find-nameserver) "racket-lang.org" "xmpp-
client")
'(#s(srv-rr 0 0 5222 "xmpp.racket-lang.org"))
> (dns-get-srv (dns-find-nameserver) "racket-
lang.org" "nonexistent-protocol")
'()
> (dns-get-srv (dns-find-nameserver) "racket-lang.org" "xmpp-
client" "tcp")
'(#s(srv-rr 0 0 5222 "xmpp.racket-lang.org"))
> (dns-get-srv (dns-find-nameserver) "racket-lang.org" "xmpp-
client" "udp")
'()
```

Added in version 6.4.0.8 of package net-lib.

```
(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. On Unix and Mac OS, this procedure parses "/etc/resolv.conf" to extract the first nameserver address. On Windows, it runs nslookup.exe.

15.2 DNS Unit

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

package: compatibility-lib

dns@ and dns^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/dns module.

```
dns@ : unit?
```

Imports nothing, exports `dns^`.

15.3 DNS Signature

```
(require net/dns-sig)
```
package: `compatibility-lib`

```
dns^ : signature
```

Includes `dns-get-address`, `dns-get-name`, `dns-get-mail-exchanger` and `dns-find-nameserver`.

16 NNTP: Newsgroup Protocol

```
(require net/nntp)
```
package: net-lib

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

16.1 Connection and Operations

```
(struct communicator (sender receiver server port)
    #:extra-constructor-name make-communicator)
  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*.

16.2 Exceptions

```
(struct nntp exn ()
    #:extra-constructor-name make-nntp)
```

The supertype of all NNTP exceptions.

```
(struct unexpected-response nntp (code text)
    #:extra-constructor-name make-unexpected-response)
  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)
    #:extra-constructor-name make-bad-status-line)
  line : string?
```

Raised for mal-formed status lines.

```
(struct premature-close nntp (communicator)
    #:extra-constructor-name make-premature-close)
  communicator : communicator?
```

Raised when a remote server closes its connection unexpectedly.

```
(struct bad-newsgroup-line nntp (line)
    #:extra-constructor-name make-bad-newsgroup-line)
  line : string?
```

Raised when the newsgroup line is improperly formatted.

```
(struct non-existent-group nntp (group)
    #:extra-constructor-name make-non-existent-group)
  group : string?
```

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

```
(struct article-not-in-group nntp (article)
    #:extra-constructor-name make-article-not-in-group)
  article : exact-integer?
```

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

```
(struct no-group-selected nntp ()
    #:extra-constructor-name make-no-group-selected)
```

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

```
(struct article-not-found nntp (article)
    #:extra-constructor-name make-article-not-found)
  article : exact-integer?
```

Raised when the server is unable to locate the article.

```
(struct authentication-rejected nntp ()
    #:extra-constructor-name make-authentication-rejected)
```

Raised when the server reject an authentication attempt.

16.3 NNTP Unit

```
(require net/nntp-unit)
```
package: compatibility-lib

```
nntp@ : unit?
```

Imports nothing, exports nntp^.

nntp@ and nntp^ are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net/nntp module.

16.4 NNTP Signature

```
(require net/nntp-sig)
```
package: compatibility-lib

```
nntp^ : signature
```

Includes everything exported by the net/nntp module.

17 TCP: Unit and Signature

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

17.1 TCP Signature

```
(require net/tcp-sig)
```
package: net-lib

tcp^ : signature

```
(tcp-listen  port-no
             [max-allow-wait
              reuse?
              hostname])      → tcp-listener?
  port-no : (and/c exact-nonnegative-integer?
                   (integer-in 0 65535))
  max-allow-wait : exact-nonnegative-integer? = 4
  reuse? : any/c = #f
  hostname : (or/c string? false/c) = #f
```

Like tcp-listen from racket/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 racket/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 racket/tcp.

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

Like tcp-accept from racket/tcp.

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

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

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

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

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

Like tcp-close from racket/tcp.

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

Like tcp-listener? from racket/tcp.

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

Like tcp-abandon-port from racket/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 racket/tcp.

17.2 TCP Unit

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

package: net-lib

```
tcp@ : unit?
```

Imports nothing and exports tcp^, implemented using racket/tcp.

18 TCP Redirect: `tcp^` via Channels

```
(require net/tcp-redirect)
```
package: net-lib

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

19 SSL Unit: tcp^ via SSL

```
(require net/ssl-tcp-unit)
```
package: net-lib

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

20 CGI Scripts

```
(require net/cgi)
```
package: net-lib

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.

20.1 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.

These functions respect current-alist-separator-mode.

```
(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.

20.2 CGI Unit

`(require net/cgi-unit)` package: `compatibility-lib`

```
cgi@ : unit?
```

Imports nothing, exports `cgi^`.

`cgi@` and `cgi^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the `net/cgi` module.

20.3 CGI Signature

`(require net/cgi-sig)` package: `compatibility-lib`

```
cgi^ : signature
```

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

21 Cookie: Legacy HTTP Client Storage

> **NOTE:** This library is deprecated; use the net-cookies package, instead. That package (source on GitHub) implements RFC 6265 [RFC6265] (which supersedes RFC 2109) and supports creating cookies on the server in an idiom more typical of Racket.

```
(require net/cookie)
```
package: net-lib

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

21.1 Functions

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

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

```
(valid-domain? v) → boolean?
  v : any/c
```

Returns #t if v represents a valid domain, #f otherwise.

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

Returns #t if v is a valid cookie name string, #f otherwise.

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

Returns #t if v is a valid cookie value string, #f otherwise.

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

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 : valid-domain?
```

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 : valid-path?
```

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

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

Modifies *cookie* with an expiration, 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 cookie-value?)
  name : cookie-name?
  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 cookie-value? false/c)
  name : cookie-name?
  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 ()
    #:extra-constructor-name make-cookie-error)
```

Raised for errors when handling cookies.

21.2 Examples

21.2.1 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
                    (list (make-header #"Set-Cookie" (string-
>bytes/utf-8 (print-cookie c))))
                    body)
```

21.2.2 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.

21.3 Cookie Unit

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

package: compatibility-lib

```
cookie@ : unit?
```

Imports nothing, exports `cookie^`.

`cookie@` and `cookie^` are deprecated. They exist for backward-compatibility and will likely be removed in the future. New code should use the net-cookies package.

21.4 Cookie Signature

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

package: compatibility-lib

`cookie^ : signature`

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

22 Git Repository Checkout

```
(require net/git-checkout)
```
package: base

The net/git-checkout library provides support for extracting a directory tree from a Git repository that is hosted by a server that implements the git:// protocol or its layering over HTTP(S). The net/git-checkout library does not rely on external binaries (such as a git client) or Git-specific native libraries (such as "libgit").

When run as a program, net/git-checkout accepts command-line arguments to drive the checkout. Use

```
racket -l- net/git-checkout -h
```

for information on command-line arguments and flags.

```
(git-checkout  hostname
               repository
               #:dest-dir dest-dir
              [#:ref ref
               #:transport transport
               #:depth depth
               #:status-printf status-printf
               #:initial-error initial-error
               #:tmp-dir given-tmp-dir
               #:clean-tmp-dir? clean-tmp-dir?
               #:verify-server? verify-server?
               #:port port
               #:strict-links? strict-links?
               #:username username
               #:password password])           → string?
  hostname : string?
  repository : string?
  dest-dir : (or/c path-string? #f)
  ref : string? = "master"
  transport : (or/c 'git 'http 'https) = 'git
  depth : (or/c #f exact-positive-integer?) = 1
  status-printf : (string? any/c ... . -> . void?)
                = (lambda args
                    (apply printf args)
                    (flush-output))
  initial-error : (or #f (-> any)) = #f
  given-tmp-dir : (or/c #f path-string?) = #f
  clean-tmp-dir? : any/c = (not given-tmp-dir)
  verify-server? : any/c = #t
```

```
port : (or/c #f (integer-in 1 65535)) = (case transport
                                          [(git) 9418]
                                          [(http) 80]
                                          [(https) 443])
strict-links? : any/c = #f
username : (or/c string? #f) = (current-git-username)
password : (or/c string? #f) = (current-git-password)
```

Contacts the server at *hostname* and *port* (where `#f` is replaced by the default) to download the repository whose name on the server is *repository* (normally ending in `".git"`). The tree within the repository that is identified by *ref* (which can be a branch, tag, commit ID, or tree ID) is extracted to *dest-dir*, and the result id an ID corresponding to *ref*.

If *transport* is `'git`, then the server is contacted using Git's native transport. If *transport* is `'http` or `'https`, then the server is contacted using HTTP(S). In the case of `'https`, the server's identity is verified unless *verify-server?* is false or the `GIT_SSL_NO_VERIFY` environment variable is set.

If *dest-dir* is `#f`, then the result is an ID determined for *ref* from just the server's report of the available branches and tags, or *ref* itself if it does not match a branch or tag name and looks like an ID.

A local clone of the repository is *not* preserved, but is instead discarded after the tree is extracted to *dest-dir*. If *dest-dir* does not exist, it is created. If *dest-dir* does exist, its existing content is left in place except as replaced by content from the Git repository.

If *ref* identifies a branch or tag by either name or by commit ID, then the `git://` protocol allows `git-checkout` to download only the commits and objects relevant to the branch or tag. Furthermore, the default *depth* argument allows `git-checkout` to obtain only the latest commit and its objects, instead of the entire history of the branch or commit. If *ref* is any other commit ID or tree ID, then the entire repository is downloaded, including all branches.

Status information is reported via *status-printf*. The same information is always logged with the name `'git-checkout` at the `'info` level.

If *initial-error* is not `#f`, then it is called (to raise an exception or otherwise escape) if initial communication with the server fails to match the expected protocol—perhaps indicating that the server does not provide a Git repository at the given address. If *initial-error* is `#f` or returns when called, an exception is raised.

If `tmp-dir` is not `#f`, then it is used to store a temporary clone of the repository, and the files are preserved unless *clean-tmp-dir?* is true. The clone does not currently match the shape that is recognized by other tools, such as `git`, and so a preserved temporary directory is useful mainly for debugging.

If *strict-links?* is true, then the checkout fails with an error if it would produce a sym-

bolic link that refers to an absolute path or to a relative path that contains up-directory elements.

If both *username* and *password* are non-#f *and* *transport* is 'http or 'https, then the provided credentials are passed to the remote server using HTTP Basic Authentication.

Added in version 6.1.1.1 of package base. Changed in version 6.2.900.17: Added the *strict-links?* argument. Changed in version 6.3: Added the *initial-error* argument. Changed in version 6.6.0.5: Added the *username* and *password* arguments. Changed in version 6.6.0.5: Changed to raise exn:fail:git exceptions instead of exn:fail.

```
(current-git-username) → (or/c string? #f)
(current-git-username username) → void?
  username : (or/c string? #f)
(current-git-password) → (or/c string? #f)
(current-git-password password) → void?
  password : (or/c string? #f)
```

Parameters used by git-checkout as the default values of the *username* and *password* arguments to control authentication with the remote server.

Added in version 6.6.0.5 of package base.

```
(struct exn:fail:git exn:fail ()
    #:extra-constructor-name make-exn:fail:git
    #:transparent)
```

Raised by git-checkout due to errors parsing or communicating with the git protocol.

Added in version 6.6.0.5 of package base.

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

[RFC6265] A. Barth, "HTTP State Management Mechanism," RFC, 2011. http://tools.ietf.org/html/rfc6265.html

Index